The use of Java Arrays in Matrix Computation

Geir Gundersen

Department of Informatics

University of Bergen

Norway

Cand Scient Thesis
April 2002

Acknowledgement

I want to thank my excellent advisor professor Trond Steihaug for a tremendous effort in the work on this thesis. Its been a real pleasure being your graduate student and it have been a positive learning experience. Thanks for all the insight you have shared with me I know it has been invaluable. These years have been both enjoyable and very demanding, they have been all in all very unforgettable. Last I want to thank you for having faith in me in many moments of frustration and despair.

I want to thank my mother for unbounded support over the years, I am forever grateful.

Last I want to thank my fiancée Trude Karlsen first of all for reading this thesis and correcting many writing errors and for being the most beautiful person, in many senses, that I have ever had the pleasure to encounter. Life would have been so less without you. Thanks for sharing life with me.

Great Hope
Great Doubt
Great Effort

Abstract

We will in this thesis show how we can utilize Java arrays for matrix computations. We will look at the disadvantages of Java arrays when used as two-dimensional array for dense matrix computation, and how we can improve upon its use when it comes to performance. We will also look at how we can create a dynamic data structure for sparse matrix computation also using Java arrays.

Contents

Chapter 1 Introduction

Chapter 2 The Java Language

Chapter 3 Implementation of Matrix Multiplication routines using Java Arrays

Chapter 4 The Computational Model

Chapter 5 Storage Schemes in Java for Large Sparse Matrices

List of Figures

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Chapter 5

List of Tables

Chapter 3

Chapter 4

Chapter 5

Appendix A

Chapter 1 Introduction

1.0 Java and Numerical Computing

Java has had an enormous success as a programming language and has made impact on graphical user interface development, network computing, server-side applications and recently the ideal coupling between Java and XML (extensible markup language) [69]. Java has the following features: platform independence, extensive libraries, a run-time system that enforces array-bounds checking, built in exception handling and an automated memory manager (supported by a garbage collector). Java is replacing existing software and in the process makes them better and inventing new software. Some of these features also make Java interesting for numerical computing [1].

Probably the single reason for Java's success is its platform independence. Java's architecture supports the platform independence of Java programs in several ways, but primarily through the Java Platform itself. The Java Platform acts as a buffer between a running Java program and the underlying hardware and operating system. Java programs are compiled to run on a Java virtual machine, with the assumption that the class files of the Java API (Application Programming Interface) will be available at run-time. The virtual machine runs the program; the API gives the program access the underlying computer's resources. No matter where a Java program run, it only need to interact with the Java Platform

Java was first intended as a platform independent dynamic web language, which was object-oriented, resembled C and C++ but easier to use. Java does not have features like pointers, templates, operator overloading and multiple inheritance. Powerful as these features may seem they tend to be the reasons for more complex and error-prone programs.

One area where Java has not been successful or been met with enthusiasm is numerical computation [1, 40, 41, 42, 43, 52, 56].

A definition of numerical analysis (computing) [73]:

Numerical analysis is the study of algorithms for the problems of continuous mathematics [73].

This means devising and analysing algorithms to solve a certain class of problems. "Continuous" means that real or complex variables are involved, real and complex numbers cannot be represented exactly on a computer. Creating algorithms that converge exceedingly fast and with small computational error is of considerable practical interest and is a goal of numerical computing.

The reason that most numerical routines are implemented in languages as FORTRAN, C and C++ are because of easy control of the memory model, performance and a rich built in numerical library [1, 40, 54, 61]. Object-oriented programming have been favored in the last decades because of its easy to understand paradigm and it is easy to build grand scale application designed in an object-oriented manner. Java has for students in many universities been the introductory language for many and is the first programming language they will encounter. They might find it natural to use the Java language for more then just an introductory course in basic programming principles and an object-oriented thinking, but also as a language for implementing numerical routines. Because of Java's considerable impact it will be used for

(limited) numerical computations. That the development is heading that way is [83] a clear indication on.

What kind of programming language one chooses to implements these numerical routines varies on the basis of the problem. Scientists and engineers have favoured FORTRAN and C for the last decades, but because of the introduction of object-oriented programming and the promise of platform independence makes Java interesting.

Java has not been regarded to become the next universal language for numerical computing despite its powerful features, the only reason is because of its poor performance compared to FORTRAN, C and C++ [54, 61].

There are several reasons for a Java application being slow, some of them are:

- The code is not optimised.
- Because of its virtual machine layer that 'hides' Java away from the underlying hardware.
- An unpredictable garbage collector.
- The object-oriented design.

The opinion of Java being is slow, seems to be changing. Progress in optimising compilers, just-in-time compilers (JIT) and HotSpot Virtual Machines has made it possible of Java to compete with FORTRAN, C and C++ on several kernels [54]. Java has still a long way to go before the whole numerical community embraces the Java language as the new universal language for scientific computing.

Java has poor performance in the following areas [58, 59, 60, 61, 62, 63]:

1. Floating point arithmetic.
2. Java arrays.
3. The memory model.
4. Automatic garbage collection.

The following areas are missing as an integral part of the Java language:

1. Complex numbers.
2. Data parallel programming.

The Java Grande Forum [41] is working extensively at adding key numerical ideas into the language.

Each new programming language that is released is apt to contain some new features or a combination of features that other languages do not have. Therefore it is interesting for the numerical community to find out if any of the features of Java are interesting when it comes to implementing numerical routines in that language. We are interested in finding if Java has features other languages do not have or if they are different from the ones of Java which makes it better for some numerical routines. Java has arrays as objects, that makes us capable of designing a new data structure for sparse matrices that have dynamic features compared to some of the usually storage schemes, see section 1.1.2 and Chapter 5, for sparse matrices.

It is expected from the reader of this thesis, that he has had introductory courses in Java as a programming language and numerical linear algebra.

1.1 Matrix Computation: Algorithms and Data Structures

Matrix computation is a large and important area in numerical computing. Solving for example large systems of linear equations on a computer is a fundamental operation for scientists and engineering in many cases. Developing efficient algorithms for working with matrices are therefore of considerable practical interest.

1.1.1 Matrix Algorithms

Dense and sparse matrix algebra is an important part of matrix computation. There are many examples of algorithms, which operates on both dense and sparse matrices.

1.1.2 Matrix Data Structures

A matrix is a rectangular array of numbers. The numbers in the array are called the entries in the matrix. The size of a matrix is described in terms of the numbers of rows (horizontal lines) and columns (vertical lines) it contains. A matrix with only one column is called a column vector or just a vector and a matrix with only one row is called row vector or just a vector. The entry that occurs in row *i* and column *j* of matrix ***A*** will be denoted ***A****ij*.

The definition of sparse matrices is that a matrix is sparse if it contains enough zero entries to be worth taking advantage of to reduce both the storage and work required in solving a linear system of equations and other basic linear algebra routines [70]. We have several different data structures for storing sparse matrices. In this thesis we will deal with the most fundamental and general data structures and implement basic linear algebra algorithms using these data structures.

Usually we choose and design the data structure based on how we intend to use the matrix, that is what routines we are implementing and how the matrix pattern is, for example symmetric and/or tri-diagonal. The most obvious difference between data structure for dense and sparse matrices is that for sparse matrices it has a more complex implementation and it only stores the nonzero structure. That is their values and their index positions that they have in the full matrix. Sparse algorithms also tend to be more complex than the same algorithms for dense matrices. The reasons for these rather complex structures and algorithms are to save space and time. The area of sparse matrix technology is large and it is not in the scope of this thesis to cover. We will present the basics that will help the reader understanding the issues and problems we deal with in this thesis.

Some of the storage schemes we have are compressed row storage (CRS), compressed column storage (CCS), coordinate storage (COO) and specialised versions of those depending on the matrix's properties. We will give an expanded definition of CRS, CCS and COO in Chapter 5. We are going to implement some basic linear algebra routines in Java using these storage schemes**,** to implement these storage schemes we use Java's native arrays. For an introduction to Java arrays in matrix computation, see section 1.2. Then we are going to implement some of

the same routines on Sparse Arrays, see section 1.2, and to some extent compare them to CRS, CCS and COO on the basis of performance (speed) and memory.

1.1.3 Matrix Operations on Different Storage Schemes

We have implemented several different matrix algorithms on the storage schemes we mentioned in the above section.

In Chapter 3 we will discuss the following elementary operations of dense matrix algebra:

-
- Multiplication of two dense matrices.
- Frobenius Norm.
- Product of a dense matrix by a vector.
- Product of a vector by a dense matrix.

We will implement these operations, the routines will be analysed on the basis of performance and the use of memory. We will discuss several different implementations for the same operations and compare them to each other.

In Chapter 5 we will discuss the elementary operations of sparse matrix algebra:

- Product of a full vector by a general sparse matrix on coordinate storage format sorted by rows and columns.
- Product of a general sparse matrix by a full vector on coordinate storage format sorted after rows and columns.
- Multiplication of two sparse matrices on coordinate storage format sorted after rows and columns.

- Product of a full vector by a general sparse matrix on compressed row storage format.
- Product of a general sparse matrix by a full vector on compressed row storage format.
- Multiplication of two sparse matrices, on compressed row storage format.

- Product of a full vector by a general sparse matrix, on compressed column storage format.
- Product of a general sparse matrix by a full vector on compressed column storage format.
- Multiplication of two sparse matrices, on compressed column storage format.

- Product of a full vector by a general sparse matrix on Sparse Arrays format.
- Product of a general sparse matrix by a full vector on Sparse Arrays format.
- Multiplication of two sparse matrices on compressed Sparse Arrays format.
- Addition of two sparse matrices on Sparse Arrays format.
- Permutation of rows and columns of matrices on Sparse Arrays format.
- Insertion and deletion of elements in a sparse matrix on Sparse Arrays format.
- Symbolic Factorisation of a matrix on Sparse Arrays format.

These operations will be implement in Java and the routines will be analysed on the basis of performance and memory. We will look at the properties of the sparse data structures (CRS, CCS, COO and Sparse Arrays) and the implementation of these routines and compare them to each other on the basis of performance (speed) and memory.

We will also briefly discuss an object-oriented graph implementation, and a symbolic factorisation routine implemented on that structure and on Sparse Arrays.

1.2 The use of Java Arrays in Matrix Computation

As we mentioned in the section above we will use Java's native arrays to implement the fundamental storage scheme both for dense and sparse matrices. We choose to use Java's native arrays instead of `java.util.Vector`, `java.util.ArrayList` and `java.util.LinkedList` classes because Java's native arrays have a better performance inserting and retrieving elements then the utility classes, see Chapter 5. Therefore it is important that the reader understands the features of Java's native arrays and it's underlying implementation. We are interested in finding the benefits of using Java arrays when it comes to performance and memory use, we will also look at drawbacks using Java's native arrays.

Java has no support for true two-dimensional arrays, as shown in Figure 1-0. We must therefore simulate a two-dimensional array with Java's arrays of arrays, as shown in Figure 1-1. This is not with out problems, but before we analyse this further we will look at how Java arrays are laid out in memory.

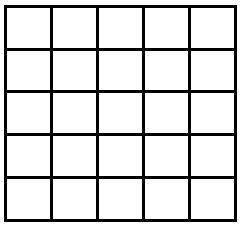

Figure 1-0
Shows the representation of a true two-dimensional array.

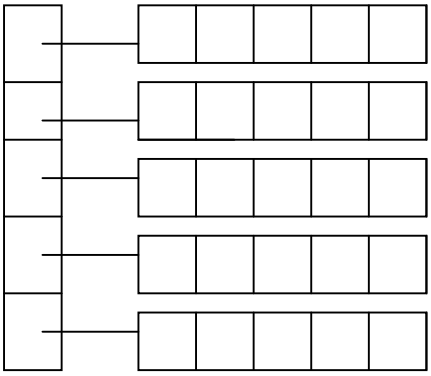

Figure 1-1
Shows the layout of a two-dimensional Java arrays.

We can for example declare and initialize a Java array like this:

```
double[][] a = {{1.0,0.0,0.0,0.0,0.0},
                {0.0,1.0,0.0,0.0,0.0},
                {0.0,0.0,1.0,0.0,0.0},
                {0.0,0.0,0.0,1.0,0.0},
                {0.0,0.0,1.0,0.0,1.0},
};
```

An element of a `double[][]` is a `double[]`, that is Java arrays are arrays of arrays. The `double[][]` is an object and its elements, `double[],` is objects. So Java arrays are objects of objects. When an object is created and gets heap allocated, see Chapter 2, the object can be placed anywhere in the memory. That means the elements of a `double[][], double[],` may be scattered throughout the memory space. The consequence is that traversing columns in a consecutive order is more time consuming than traversing the rows in a consecutive order, as shown in Figure 1-2. Figure 1-2 illustrates a Frobenius norm operation, this operation and the column versus row traversin will be further discussed in Chapter 3 and 4.

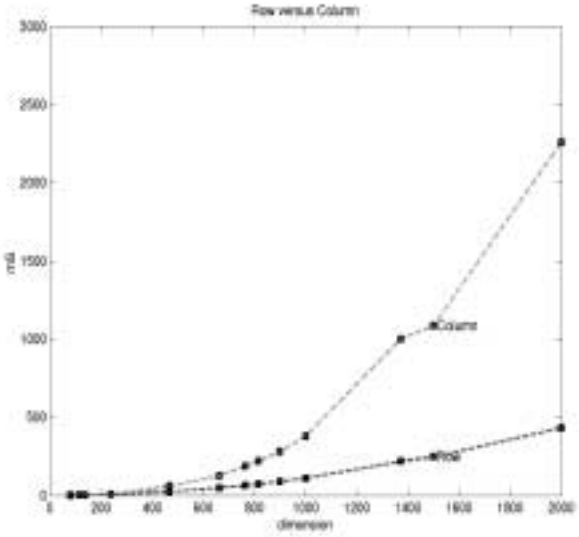

Figure 1-2
The time accessing the two-dimensional Java array of type double row-wise and column-wise.

Another observations is that the elements of a `double[][]` can have different lengths, as shown in Figure 1-3. This may cause problems for storing dense matrices on that structure where we "demand" that we work on rectangular. As we shall see in Chapter 5 we are able to create a new data structure for sparse matrices that have dynamic features, which is special for the Java. That structure is Java arrays, where we only store the nonzero elements in each row this is possible since we can have different lengths on Java arrays. We declare a Java array for values and one for their indexes. We have one of type `int[][]` for storing indexes and one of type `double[][]` for storing values. We can now work on the rows (`double[]` and `int[]`) independently of the rest of the structure. The goal here is to find out if this storage scheme is more suitable for matrix computations then other storage schemes. Performance will always be an important issue. It is therefore important to find out if this new structure can compete with CRS, CCS and COO, which does not have the possibility to work on the rows

independently from the rest of the nonzero structure (for example insertion and deletion of elements), when it comes to performance.

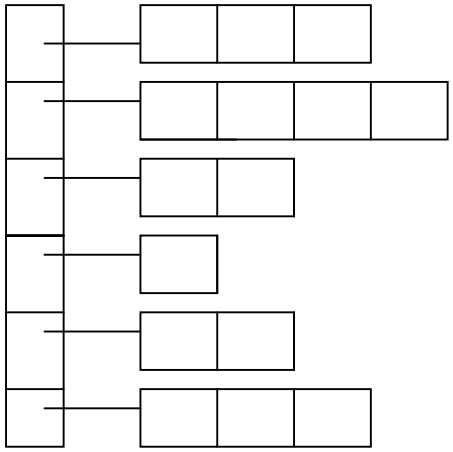

Figure 1-3
The layout of a two-dimensional Java arrays where we have different row lengths.

In Chapter 3 we analyze how we can implement a matrix multiplication on Java arrays for dense matrices, considering the row-wise orientation. We also analyze one package implementation of the matrix multiplication routine and based on that routine's performance, we implement another matrix multiplication routine that takes the row-orientation into fully consideration. These two routines are compared against each other on the basis of performance and memory use.

In Chapter 5 we will introduce the use of Java arrays for storing sparse matrices, we will also look at several different storage formats and implement them in Java for analyzing both performance and memory. Then we will introduce an alternative storage scheme, Sparse Arrays, for storing sparse matrices based on the features of Java arrays.

1.2.1 Optimisation of Java Code in Matrix Computation routines

For most developers, optimisation is a dreaded subject. This is because it can make the code more complex, harder to understand or harder to maintain and it is not certain we achieve the performance we hoped for. In developing numerical routines, optimisation maybe crucial for achieving a competitive performance [1, 40, 42]. In Chapter 2 we will discuss some of the performance issue concerning Java. We will also show some optimising techniques that are relevant for implementing the routines in Chapter 3.

Examples of optimising:

- Alias multidimensional arrays in loops , that is turn `A[i][j]` into `Ai[j].`
- Declare local (scalar) variables in inner-most scope, `for(int i;---).`
- Use `+=` instead of `+` semantics for methods to reduce the number of temporaries created.
- Do not create and destroy lots of little objects in innermost loops. Java's garbage collector can slow things down. Instead we can reuse objects if that is possible.

- Do not use `java.util.Vector` class for numerics, this is designed for heterogeneous list of objects not scalars. You will need to cast each element when accessing it.

In Chapter 2 we will discuss several other optimising techniques and give a brief introduction to the performance of the Java language.

Another important subject that is crucial for performance is the implementation of the Java compiler and the Java Virtual Machine, these will be discussed in Chapter 2. Compilers and JVMs will not have a central part in this thesis, but they will have an important implication on how we conclude and explain the experimental data. To really get under the hood of the impact of compilers and JVMs we need more than this thesis. What we are looking for is general knowledge, which is not totally dependent on the compiler or Java's Virtual Machine. When we cannot give 'reasonable' explanations to experimental data and the reason is most likely the impact of the compiler or the JVM we will mention it but not discuss it in great detail. This will without a doubt limit this thesis, but still there are enough interesting experimental data and brilliant ideas presented here that will give a real contribution to the field of numerical computing in Java.

Considering the construction `double[][]` it is 'hidden' from the programmer that column traversing is handled differently then row traversing in the JVM. Column traversing means that we are going from one array-object to another array-object. This means the JVM extensively have to access different references and their objects. Row traversing we means that we are only accessing elements in one array-object. This means the JVM do not have to extensively work with different references and objects accessing elements. An example of the Java Virtual Machine impact is the Frobenius norm example, see Figure 1-2. In Chapter 3 we will show the consequence of this column traversing cost when we compare different implementations and in Chapter 4 we will explain how much impact this has when we explain the differences for different implementations. Since this is handled in the JVM we will not try to explain this in more details, but we will take the consequence of this when implementing a matrix multiplication routine to see how much we can achieve (performance) taking the row-wise structure into fully consideration.

An example of the compiler's impact is 'dead code elimination' [72]. If the compiler discovers that code is not used in the executing program it can be eliminated. In Chapter 4 we will see that this may have an impact on benchmarking Java constructions.

The numerical community has considered other alternatives then the use of Java arrays as a structure for storing matrices for high performance computing [58, 62]. There are many convincing arguments for that decision [58, 62]. But still there is no documented alternatives to Java arrays, and since it is an integral part of the Java language with an elegant notion (`int[]`, `double[][]`) and its performance for retrieving and inserting elements we will use Java arrays to implement data structure for storing both dense and sparse matrices. Designing high performance routines using Java arrays, the knowledge of Java arrays memory layout and good knowledge on how to optimise the Java code will be in the centre of all these experiments.

In Chapter 3 we will use optimising techniques to utilise the Java arrays more efficient and we will use additional optimising techniques for generally making the routines faster. Optimising Java arrays, as we will see in Chapter 3 and 4 have a real impact on the performance.

Chapter 2 The Java Language

2.0 Introduction

This chapter is meant as a supplement for the reader to understand all aspect of this thesis, especially the performance and memory aspect of the Java languages. We do not introduce any new theory in this chapter, but we are trying to explain some parts of the Java language that is essential for making an efficient application. We are trying to explain those mechanisms that are special for the Java the language and how it handles different operations. In this process we will introduce some fundamental terminology.

We give a short description of Java's Virtual Machine, garbage collector, heap allocator, primitives, objects/references. And finally we will give some advice on how to make Java code faster.

2.1 The Basics of the Java Platform

Java's architecture comprises four distinct and interrelated technologies: language, classfile format, Java API (Application Programming Interface) libraries and Java Virtual Machine (JVM) [48, 49, 50]. When executing a Java application, the source code is written in the Java language, which is compiled into the classfile format and in turn executed in the JVM. Additionally, Java applications call methods from the Java API libraries that provide access to system resources for example graphics, networking and disk I/O. Together, the JVM and Java API libraries form a compile and runtime execution environment, which is also known as the Java platform.

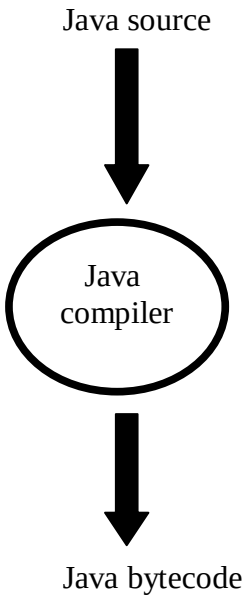

Figure 2-0

Connection between the Java source, Java compilers and Java bytecodes.

Java achieves architectural independence by compiling its source code into its own intermediate representation, not the typical machine-specific op-codes of a particular hardware platform. The intermediate representation is called *byte code*. The byte code produced is designed to run in a

stack-based architecture. Unlike the assembly-language representation of most modern machines that are register based, a stack-based machine has no registers. The JVM is an abstract computer architecture based on a dynamic stack that provides operations to push, pop, and manipulate data. The JVMs main functions are to load classfiles and execute the resulting byte codes.

2.2 The Java Virtual Machine

As stated in the above section the JVM is an abstract computer that runs compiled Java programs. The JVM is "virtual" because it is generally implemented in software on top of a "real" hardware platform and operating system [48, 49, 50]. All Java programs are compiled for the JVM. Therefore, the JVM must be implemented on a particular platform before compiled Java programs will run on that platform.

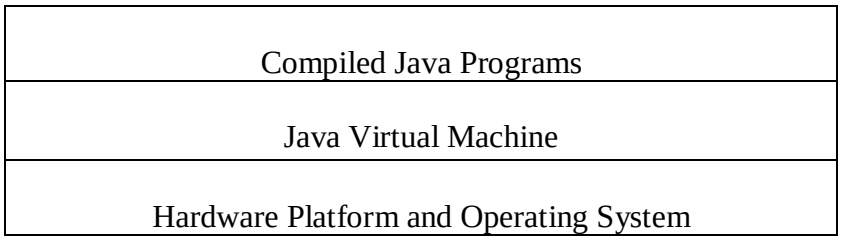

Figure 2-1
Connection between compiled Java programs, JVM and hardware platforms and operating systems.

The JVM plays a central role in making Java portable. It provides a layer of abstraction between the compiled Java program and the underlying hardware platform and operating system, as shown in Figure 2-1.

The JVM works with data in three forms: objects, object references and primitive types [51]. Objects reside on the garbage-collected heap. Object references and primitive types reside either on the Java stack as local variables, on the heap as instance variables of objects or in the method area as class variables.

In the JVM, memory is allocated on the garbage-collected heap only as objects. There is no way to allocate memory for a primitive type on the heap, except as part of an object. If we want to use a primitive type where an `Object` reference is needed, we can allocate a wrapper object for the type from the `java.lang` package. For example, there is an `Integer` class that wraps an `int` type with an object. Only object references and primitive types can reside on the Java stack as local variables. Objects can never reside on the Java stack.

The architectural separation of objects and primitive types in the JVM is reflected in the Java programming language in which objects cannot be declared as local variables. Only object references can be declared as such. Upon declaration, an object reference refers to nothing. Only after the reference has been explicitly initialized, either with a reference to an existing object or with a call to `new`, does the reference refer to an actual object.

In the JVM instruction set, all objects are instantiated and accessed with the same set of opcodes, except for arrays. In Java, arrays are full-fledged objects and like any other object in a Java program, are created dynamically. Array references can be used anywhere a reference to type `Object` is called for and any method of `Object` can be invoked on an array. Yet, in the JVM, arrays are handled with special bytecodes.

As with any other object, arrays cannot be declared as local variables, only array references can. Array objects themselves always contain either an array of primitive types or an array of object references. If we declare an array of objects, we get an array of object references. The objects themselves must be explicitly created with `new` and assigned to the elements of the array. An object is a class instance or an array. The reference values (often just references) are pointers to these objects, and a special `null` reference, which refers to no object.

There may be many references to the same object. Most objects have state, stored in the fields of objects that are instances of classes or in the variables that are the components of an array object. If two variables contain references to the same object, the state of the object can be modified using one variable's reference to the object and then the altered state can be observed through the reference in the other variable.

2.3 The Garbage Collected Heap

The heap is where the objects of a Java program lives [55]. Every time we allocate memory with the `new` operator, that memory comes from the heap. The Java language does not allow us to free allocated memory directly. Instead, the runtime environment keeps track of the references to each object on the heap and automatically frees the memory occupied by objects that are no longer referenced, a process called garbage collection. Automatic garbage collection is a central aspect and a key feature in Java, which makes it different from C and C++. The garbage collector usually kicks in when we do memory-demanding operations in the program. It is not only objects that gets heap allocated primitives as well (`double`, `int` and `boolean`), as instance variables. The programmer has no control of where in the memory the objects gets heap allocated.

2.4 Improving Byte Code Execution: HotSpot and JITs

In Chapter 4 some of the routines we implement have different running characteristics on different JVMs. It has not been in the scope of this thesis to get under the hood of Java’s JVM to explain its underlying workings. We are only observing the different characteristics and concluding what impact it will have on using different implementations of the routines.

The development of fast JVMs has become an important goal, this because Java is considered used in many more areas then it was first thought of. For numerical computations it is crucial that it can have a satisfying performance and it is important that it can deliver a consistent performance and result across hardware platforms. But things are under development and some improvements are called JIT Compiler and HotSpots [44, 49].

The Java HotSpot performance engine is Sun's next-generation compilation technology that compiles only performance-critical code. The latest version of HotSpot provides both client- and server-side solutions [45, 61, 72].

A JIT compiler simply compiles byte code as it executes. JIT refers to the fact that it compiles byte code only when it's ready to be executed, just in time in other words. JIT compilers are very powerful because only code that actually executes is compiled. Methods that are executed frequently can be retrieved from the native method cache at execution time and executed on the fly. There is one big issue with JIT compilers: once a method has been compiled, it loses the power of portability because it has been compiled to the native code of the hardware and consequently it is not portable any longer.

HotSpot has a dynamic compiler, but it differs from a JIT because it uses heuristics to determine which section of the code should be executed and it is also more aggressive when optimizing the selected code than a typical JIT compiler.

2.5 Benchmarking

There are two basic types of benchmarking. Benchmarking to discover relative speeds or bottlenecks and benchmarking to understand performance characteristics and thus where to expect bottlenecks. This information can help when we look for performance bottlenecks and avoid accidentally building bottlenecks into the applications. It also helps us avoid the sin of premature optimization, if a simple, clean algorithm is expected to run fast enough, wait to optimize until after the need is clear. In Chapter 3 we will benchmark the routines to find their relative speeds for various inputs. In Chapter 4 we will do benchmarking to understand the performance characteristics and locate bottlenecks in those routines we present in Chapter 3.

2.6 Method Calls

The ability to write and call methods is a critical tool for building and maintaining large systems. Methods allow programs to be broken into smaller, more easily handled chunks. However, if method calls slow down a running program, programmers will design systems with bigger parts and fewer method calls.

Object-oriented programming increases the number of method calls when compared to equivalent procedural programs because it encourages more data encapsulation. Compare these two lines of code and notice the extra method call in the line showing encapsulation:

Without encapsulation: `double x = anObject.Z;`

With encaspulation: `double x = anObject.getZ();`

Encapsulation increases the number of method calls in a program, so it is essential that those method calls execute quickly. If method calls do not execute quickly, programmers often attempt to speed up their programs by avoiding encapsulating the data in their programs. We have avoided method calls for all the routines in Chapter 3 and for some of the routines in Chapter 5, but that also mean we have broken data encapsulation. The ‘hidding’ of data is a fundamental object-oriented principle, so we have gained performance by breaking this principle. Because the early Java environments shipped without JIT compilers, method calls were much slower than current Java environments. The encapsulation shown above may have been unacceptably slow to run in those early environments, with the result that the data is public. Fortunately, today's JIT-enabled Java environments perform method calls much faster

than earlier non-JIT-enabled environments. There is less of a need to make speed-versus-encapsulation tradeoffs in these environments. When running in a Java environment without a JIT, method calls take anywhere between 280 and 500 clock cycles [57]. A good JIT can speed up method calls by a factor of more than 100 -- so in target environments with a good JIT, we can have both encapsulation and speed [57]. In environments without a JIT or with a poor JIT, programmers must decide on a case-by-case basis whether speed or encapsulation is more important. A good JIT can make this decision unnecessary.

If we expect to run with a reasonable JIT, method calls are no more expensive in Java than they are in C or C++ [57]. If we expect to run on a system without a JIT or without a very good one, this is something we'll have to pay attention to in the speed-critical portions of the application.

2.7 Object Creation: Java versus C and C++

Object creation in a Java program is fundamentally different than that in an equivalent C or C++ program. Many of the small temporary objects that C or C++ would create on the stack are created on the heap in Java [57]. In C and C++, when these objects are discarded at the end of a method call, the space is available for more temporary objects. In Java, the objects are discarded, but the space is not reclaimed until the next garbage collection, which usually does not happen until the heap memory is exhausted. The space for the *next* temporary object in a Java program always comes off the heap. The space for the new temporary object is rarely in a cache, and so the initial use of the temporary object should run slower than the initial use of a temporary object in a C or C++ program [57].

With the current Java runtimes, performance is also affected because creating a new object is roughly as costly as a `malloc` in C or a `new` operation in C++. Creating a new object in any of these ways takes hundreds or even thousands of clock cycles. In C and C++, creating an object on the stack takes about 20 clock cycles. C and C++ programs create many temporary objects on the stack, but Java programs do not have this option.

One important thing to notice is that a JIT does not speed up object creation time nearly as much as it speeds up method calls [57]. The bottlenecks for the same program running under an JIT-enabled environment and running under a non-JIT-enabled environment might well be different, if only subtly, because of this. While profiling a Java application, we should bear in mind whether or not the profiler has performance characteristics similar to the expected runtime environment or environments.

Object creation is something we must pay attention to for all Java programs. This does not mean we should avoid object creation at all costs, but simply be aware of where we are creating many objects and watch for bottlenecks there.

Object creation time is much more expensive in Java than it is in C or C++ [57]. With a good JIT and runtime, it is 10 to 12 times as expensive and with a poor runtime, it is about 25 times as expensive. JITs improve this situation, but not nearly as much as they improve things like method-call speed [57]. For performance in Java programming, the most important thing to pay attention to, usually is the amount of object creation. We have not extensive object creation in the routines of Chapter 3 and 4. In Chapter 5 we have extensive object creation in some of the routines.

2.8 Java Arrays

Java, unlike C and C++, implements arrays as true objects with defined behavior, including range checking. This language difference imposes overhead on a Java application using arrays that are missing in equivalent C and C++ programs. Creating an array is object creation. Further, using an array involves the overhead of checking the array object for `null` and then range-checking the array index. C and C++ omit the checks for `null` and the range checks, and therefore should run faster. We will in Chapter 5, see that we can in fact utilize the object behavior to design a data structure with dynamic features.

In general, array access is 2 or 3 times as expensive as accessing non-array elements, even for large arrays when the range checking can be amortized over many accesses [57, 39]. Array access for non-JIT-enabled environments is about one-tenth as fast as JIT-enabled environment [57]. Array access is slower than accessing atomic variables. If we expect to access a few elements in an array many times each, copying them to atomic variables can speed things up.

In Chapter 3 we will give a more formal definition of Java arrays.

2.9 Optimizing Techniques

In this section we present some optimizing techniques that we used in Chapter 3 and some that we recommend:

- Calling `System.gc()` does not necessarily force a synchronous garbage collection. Instead `gc()` call is really a hint to runtime that now is a good time to run the garbage collector. The runtime decide whether to execute the garbage collection at that time and what kind of garbage collection to run [31].

- Using `javac -O` we achieve the following; it makes the class file smaller and quicker to load and it can inline methods of modifiers like private, final and static [74].

- Creating Objects costs time and CPU effort for an application. Therefore avoid creating objects in frequently used routines. Instead of creating new objects it may be more efficient to reuse old ones.

- Access instance variables directly rather than through accessor methods.

- Most of the applications in this thesis spend their times in loops therefore we give some optimization techniques that can speed up the loops:

    1. Take out any code that does not need to be executed on every pass.
    2. Move any method calls out of the loop, even if it requires rewriting.
    3. Array access (and assignment) always has more overhead then temporary variable access because the JVM performs bounds checking for array-elements access.

4. Comparison to 0 is faster than comparison too most other numbers. This alternation typically reverses the iteration order of the loop from counting up (0 to max) to counting down (max to 0).

   Instead of `for(int i = 0;i<N;i++)`
   We use `for(int i = N-1;i>=0;i--)`
   Or `for(int i = N-1;--i>=0;)`

- `System.arraycopy()` is faster than using a loop for copying arrays in a destination JVM except where we are guaranteed that the JVM has a JIT. In the latter case the own loop may be faster [36], see Chapter 5.

- The fastest ordered collection in Java are plain arrays (`int[], Object[],` etc) [38]. The drawback using these directly is lack of object-oriented methodology we can apply. Arrays are not proper classes that can be extended.

Summary, Discussions and Conclusions

In this chapter we have introduced some issues and subjects that are central to understand how to develop efficient routines in Java. This was only a brief introduction to a much more complex and larger subject. But it is essential to understand the concept of the garbage collector, JVMs, object creations, method calls and Java arrays to appreciate what we are trying to achieve in this thesis. And better understand the consequences of certain operations we implement.

The *relative* speed of different activities varies a lot from platform to platform and between JIT-enabled and non-JIT-enabled systems. To find the bottlenecks in the program we must profile the application on a system with performance characteristics similar to the systems on which we expect the application to run. It is a very complex issue coming up with a general '*how to do it approach*' when creating efficient routines in Java. Optimizing for one platform may cause it to be slower then other platforms. Maybe the best conclusion is that most of the optimization must be performed in the compiler. We must have an efficient runtime environment for all platforms, we cannot have grave inconsistency running a routine on several different platforms. How to approach such a problem is not in the scope of this thesis to elaborate, only to say that there is a need for better compilers and JVM. We will instead focus on how we can utilize the constructions in the Java language, as Java arrays, to create efficient routines.

Chapter 3 Implementation of Matrix Multiplication routines using Java Arrays

3.0 Introduction

In languages like FORTRAN, C and C++ there are extensive libraries when it comes to numerical linear algebra routines. LAPACK, LINPACK, SPARSKIT, Sparselib++ and NIST Sparse BLAS are some of them. Ever since Java emerged on the scene there has been development of numerical libraries in that language, JAMA and Jampack are two of these packages developed in Java. They both offer fundamental linear algebra operations (matrix multiplication, norm1-maximum column sum, transpose, the infinity norm, frobenius norm, addition of two matrices, subtraction of two matrices, scalar by matrix and more), decompositions (LU, QR and Cholesky Decompositions). What they both have in common is that they use Java arrays to store their matrices. In this chapter we will give a short introduction to these packages.

Unlike addition and subtraction of matrices it is difficult to give a general machine independent rule for the optimal algorithm for matrix multiplication. With machine in this context we mean the Java Virtual Machine. In fact matrix multiplication is a classic example of an operation, which is very dependent on the details of the architecture of the machine. We will therefore discuss several different implementation of that operation using Java arrays as the underlying data structure.

We will implement a rather straightforward matrix multiplication routine, see section 3.6. Then we will focus on JAMA's implementation of the matrix multiplication routine and compare it to the straightforward routine on the basis of performance and memory. Last we will implement a matrix multiplication routine that takes the row-wise layout of a two-dimensional array into consideration. This implementation will be compared to JAMA's matrix multiplication routine on the basis of performance and memory. We are interested in finding out how much impact the layout of Java arrays has when it comes to performance for the routines we implement and the memory demand.

3.1 Java Arrays

In this section we will give a more formal definition of Java arrays and give an example of the impact column versus row traversing has on performance.

An array is a sequence of indexed components with the following general properties [68]:

- The length of the array (its number of components) is fixed when an array is constructed.
- Each component has a fixed and unique index. The indices range from a lower index bound through a higher index bound.
- Any component of the array can be accessed (inspected or updated) using its index.

Java arrays has the following specific properties [68]:

- For an array of length n, the index bounds are 0 and n – 1.
- Every array is homogeneous, the type of its component being stated in the program. The components may be values of some stated primitive type or objects of some stated class. (Java actually allows a measure of heterogeneity. An array of type `C[]`, where `C` is an object class, may contain objects of any subclass of `C`).
- An array itself is an object. Consequently, an array is allocated dynamically (by means of '`new`', is manipulated by reference and is automatically deallocated when no longer reference to.)
- The notation for accessing the component with index *i* in array `a` is '`a[i]`' and the notation for inspecting the length of array `a` is '`a.length`'.
- Arrays are objects that store a set of elements in an index-accessible order. Those elements can either be a primitive data type, such as an `int` or `float` or any type of `Object`. To declare an array of a particular type, just add brackets (`[]`) to the declaration.
- Elements of a primitive declared type arrays are initialized to `zero` or `false.`
- Elements of an object declared type are initialized to `null.`

3.1.1 Java Arrays of Primitives and Objects

When creating an array of primitive elements, the array holds the actual values for those elements. Unlike an array of primitives, when we create an array of objects, they are not stored in the actual array. The array only stores references to the actual objects and initially each reference is `null` unless explicitly initialized. The objects are not in the array; only *references* to the objects are in the array.

3.1.2 Multidimensional Arrays

Because arrays are handled through references, there is nothing that stops us from having an array element refer to another array. When one array refers to another, we get a multidimensional array, as shown in Figure 3-0. This requires an extra set of square brackets for each added dimension on the declaration line. For instance, if we wish to define a rectangular two-dimensional array, we might use the following line:

```
double[][] values;
```

As with one-dimensional arrays, if an array is one of primitives, we can immediately store values in it once we create the array. Just declaring it is not sufficient. For instance, the following two lines will result in a compilation-time error because the array variable is never initialized:

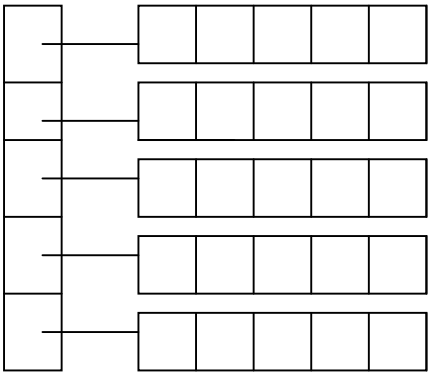

Figure 3-0
The layout of a two-dimensional Java array.

```
double[][] values;

values[1][1] = 10.00;
```

If however, we created the array between these two source lines (with something like the next code), the last line would become valid.

```
values = new double[3][4];
```

In the case of an array of objects, creating the multidimensional array produces an array full of `null` object references. We still need to create the objects to store in the arrays.

Because each elements in the outermost array of a multidimensional array is an object reference, there is nothing that requires were arrays shall be rectangular (or cubic for three-dimensional arrays). Each inner array can have its own size.

We might expect the elements of a row to be stored continuously, but we cannot expect the rows to be stored continuously. A consequence of this is if the two-dimensional array is large, then accessing the consecutive elements in a row will be faster then accessing consecutive elements in a column. This is one of the basic observations in this chapter.

Other observations concerning Java arrays are:

- The compiler must take into account array aliasing, two rows are the same in a matrix and arrays can have different lengths.
- Garbage collection overhead for Java arrays is large, since all rows of the array are dealt with independently.
- Extensive run-time arrays bounds checking.

Array aliasing means that it is possible for a two-dimensional array to have the same element in two places. For example `double[][]` can have the same row, `double[]`, in two places, as shown in Figure 3-1.

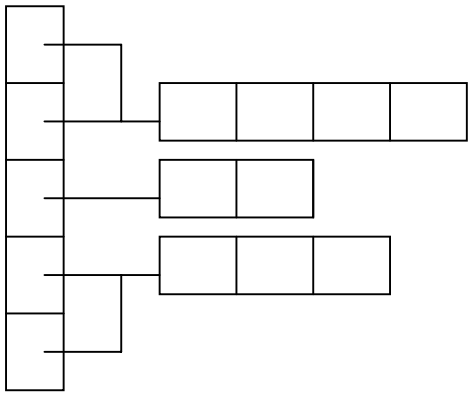

Figure 3-1
A two-dimensional Java array where we have array aliasing and different row lengths.

When we garbage collect a two-dimensional Java array, for example `double[][]`, we are actually garbage collection all the elements of `double[][]` independently, that is `double[]`. Instead of garbage collecting one object we actually garbage collecting a bunch of objects. This is time consuming because this leads to a lot of work for the garbage collector or the garbage collector can kick in anytime, which can lead to a halt in the program that slows down the routine [75].

Java has extensive array bounds checking this may be necessary for security reasons, but it adds extra overhead.

In this next example we return to the Frobenius norm example first introduced in Chapter 1, see section 3.4 for a full definition of the Frobenius norm. This example illustrates the difference in performance between consecutive row traversing and consecutive column traversing.

The mathematical definition of the operation is:

$$s = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} Aij \qquad (3\text{-}0)$$

These two next code examples shows the implementation of *(3-0)* in Java:

Frobenius norm example

Loop-order (i,j):

```
double s = 0;
double[] array = new double[m][n];
for(int i = 0;i<m;i++){
     for(int j = 0;j<n; j++){
          s+=array[i][j];
     }
}
```

Loop-order (j,i):

```
double s = 0;
double[] array = new double[m][n];
for(int j = 0;j<n;j++){
     for(int i = 0;i<m; i++){
          s+=array[i][j];
     }
}
```

So given an algorithm description, in this case the mathematical definition the loop-order (i, j): means that we are traversing the matrix row-by-row and loop-order (j, i) means that we are traversing the matrix column-by-column.

These routines finds the sum of all the elements in a matrix and stores the sum in `s`. The sum `s` are the same either we traverse the matrix, column-by-column with the loop-order (j,i) or row-by-row with the loop-order (i,j). This would be an essential operation finding the Frobenius norm of a matrix with non-negative numbers. Traversing consecutive elements of a row or a column in a matrix is a common operation in almost all-numerical linear algebra routines. To do that we can either traverse the array row-by-row or column-by-column, as shown in Figure 3-2.

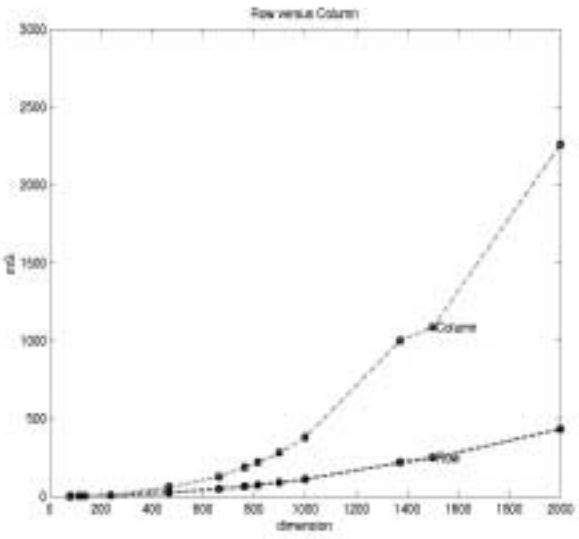

Figure 3-2
Time accessing the array matrix with loop-order (i,j) (Row), and loop-order (j,i) (Column).

Figure 3-2 shows that traversing columns is much less efficient then traversing rows when the array gets larger. Traversing the Java array as we did here is an essential operation in many basic linear algebra routines not only the Frobenius norm but also for a matrix multiplication operation. We are therefore interested in finding out if this has an impact on the matrix multiplication routine.

3.2 Matrix Multiplication

In this section we give the formal definition matrix multiplication.

If ***A*** is an *mxn* matrix and ***B*** is an *nxp* matrix, then the product ***AB*** is the *mxp* matrix. The definition of matrix multiplication requires that the number of columns of the first factor ***A*** must be the same as the number of rows of the second factor ***B*** in order to form the product ***AB***. If this condition is not satisfied, the problem is undefined.

Multiplication of two matrices is defined as follows, when ***A*** is *mxn* and ***B*** is *nxp*:

$$Cij = \sum_{k=0}^{n-1} AikBkj \qquad i = 0,1,2,3,\ldots,m\text{-}1 \quad j=0,1,2,3,\ldots p\text{-}1 \qquad (3\text{-}1)$$

In Java `A[i][j]` is the same element as ***A****ij*. When we say we are accessing a column of a matrix we are looping through a matrix `A[i][j]` the index `i` is running. The same goes for accessing the rows of a matrix `A[i][j]` the index `j` is running.

We use the following abbreviations in this chapter:

- ***AB***: multiplication of two matrices.
- ***Ab***: product of a matrix by a vector.
- $\boldsymbol{b^T A}$: product of a vector by a matrix.

These abbreviations are also used in Chapter 4 and 5 with the same meaning.

3.3 JAMA and Jampack

JAMA [2] and Jampack [3] are two numerical linear algebra packages implemented in Java. JAMA is based on a single matrix class within a strictly object-oriented framework, Jampack uses a more open approach that lends itself to extension by the user. Both packages store their matrices using Java arrays.

Even if we only deal with JAMA's matrix multiplication routine in this chapter, JAMA itself is interesting and deserves a presentation.

JAMA is a basic linear algebra package for Java. It provides user-level classes for constructing and manipulating real dense matrices. It is meant to provide sufficient functionality for routine problems, packaged in a way that is natural and understandable to non-experts. It is intended to serve as *the* standard matrix class for Java.

JAMA is comprised of six Java classes: `Matrix`, `CholeskyDecomposition`, `LUDecomposition`, `QRDecomposition`, `SingularValueDecomposition` and `EigenvalueDecomposition`.

The `Matrix` class provides the fundamental operations of numerical linear algebra. Various constructors create matrices from two-dimensional arrays of double precision floating point numbers. Various *gets* and *sets* provide access to sub matrices and matrix elements. The basic arithmetic operations include matrix addition and multiplication, matrix norms and selected element-by-element array operations. A convenient matrix print method is also included.

Five fundamental matrix decompositions, which consist of pairs or triples of matrices, permutation vectors, and the like, produce results in five decomposition classes. The `Matrix` class to compute solutions of simultaneous linear equations, determinants, inverses and other matrix functions, accesses these decompositions.

The five decompositions are:

1. Cholesky Decomposition of symmetric, positive definite matrices.
2. LU Decomposition (Gaussian elimination) of rectangular matrices.
3. QR Decomposition of rectangular matrices.
4. Eigenvalue Decomposition of both symmetric and nonsymmetrical square matrices.
5. Singular Value Decomposition of rectangular matrices.

The designers of JAMA claims that it represents a compromise between the need for pure and elegant object-oriented design and the need to enable high performance implementation [2].

Next we present the 'skeleton' of the `Matrix` class, for more information on this class and the rest of the methods in it see [2]:

```
public class Matrix{
     private int m;//the number of rows in the Java array
     private int n;//the number of columns in the Java array
     private double[][] A;
     public Matrix(double[][] A){
          this.A = A;
          this.m = A.length;
          this.n = A[0].length;
    }
     public Matrix times(Matrix B){…}
     public double normF(){…}
}
```

A JAMA `Matrix` object is constructed like this:

We first construct a two-dimensional Java array of type double:

```
double[][] array = {{1.0,0.0,0.0,0.0,0.0},
                    {0.0,1.0,0.0,0.0,0.0},
                    {0.0,0.0,1.0,0.0,0.0},
                    {0.0,0.0,0.0,1.0,0.0},
                    {0.0,0.0,1.0,0.0,1.0},
                };
```

Then we construct the `Matrix` object `A`, with the `array` reference as a parameter to the object:

```
Matrix A = new Matrix(array);
```

We do a matrix multiplication in JAMA like this:

First we create a new two-dimensional Java array of type double:

```
double[][] array1 = {{1.0,0.0,0.0,0.0,0.0},
                     {0.0,1.0,0.0,0.0,0.0},
                     {0.0,0.0,1.0,0.0,0.0},
                     {0.0,0.0,0.0,1.0,0.0},
                     {0.0,0.0,1.0,0.0,1.0},
                   };
```

Then we construct the `Matrix` object `B`, with the `array` reference as a parameter to the object

```
Matrix B = new Matrix(array1);
```

Then we perform the actual matrix multiplication:

```
Matrix C = A.times(B);
```

As we can see JAMA's `times()` routine takes a reference of a `Matrix` object as a parameter and it is returning a reference of the resulting `Matrix` object. The routine works directly on the Java arrays instances. The `Matrix` class also demands that the row length in a `double[][]` have the same length. It does extensive checking to see if this is true, so if the rows of a two-dimensional array have different lengths. The other instance variables are for the dimension *mxn*, where `m` is the row length and `n` is the column length. The `times()` method will be further explained in section 3.7.

Jampack (JAva Matrix PACKage) is a collection of classes and methods for manipulating matrices.

Jampack includes classes for several types of matrices (e.g., diagonal and triangular), methods implementing the common matrix operations and methods to compute the most important matrix decompositions.

The guiding principles of Jampack are openness and extensibility. Jampack makes no attempt to hide the implementation of its classes and methods

Jampack is also open-ended. Matrix operations, transformations and functions are organized into classes of methods called suites. New features to Jampack can be added by placing new methods in the appropriate suite. It will often turn out that methods already in a suite can be used as templates to guide the coding of new features.

The classes of Jampack falls into three broad categories: classes that define matrices, classes that manipulate matrices, and classes that generate matrix decompositions. In addition, there is a class defining complex numbers and their operations and a class defining global parameters for the package. Let's look briefly at each in turn.

There are currently two matrix types implemented in Jampack--complex general matrices (`Zmat`) and complex diagonal matrices (`Zdiagmat`). (Eventually the core package will include real versions of these classes and real and complex classes for band matrices.) In addition, there are three subclasses of `Zmat` for upper triangular, lower triangular, and positive semidefinite matrices.

The classes for manipulating matrices are called suites. The most important suites are the ones that implement the basic matrix operations: `Plus`, `Minus`, `Times`, `Solve` and `H` (for conjugate transpose). Generally a suite will have several methods. For example, the `Times`

suite contains, among others, methods for multiplying two `Zmats`, multiplying a `Zmat` by its conjugate transpose or multiplying a `Zmat` by a `Zdiagmat`. The generic method in any suite is named "`o`". For example, `Times.o(A,B)` will produce the product of `A` and `B` for any combination of `Zmats` and `Zdiagmats`. Special methods have longer names. For example, `Times.aha(A)` computes $A^H A$.

Jampack has classes implementing the six basic decompositions of matrix computations: the partially pivoted LU decomposition (`Zludpp`), the Cholesky decomposition (`Zchol`), the QR decomposition (`Zqrd, Zhqrd`), the Schur decomposition (`Schur`), the spectral decomposition of a Hermitian matrix (`Zspec`) and the singular value decomposition (`Zsvd`). In addition it has an eigenvalue-vector decomposition (`Eig`).

The class Z, which provides useful constructors and the usual operations, implements complex numbers. The operations are written in the form `c.op(a,b)` which causes c to be overwritten by `a.op.b`. This convention allows computations with `Zs` to proceed without the creation of new `Zs`, which is very important in inner loops.

A `Zmat` is implemented as two arrays of type double, one representing the real part of the matrix, the other the imaginary part. The indexing of these arrays begins at zero, which is normal for Java arrays. In Jampack we say that a primitive array has a *base index* of zero. Matrices, as opposed to primitive arrays, usually have nonzero base indices, which vary from discipline to discipline. Consequently, Jampack allows the user to set the base index of the matrices in a Java application. To keep things from becoming too complicated, the base may be set only once per application.

An example of doing matrix multiplication using Jampack's `Times` suite:

Creates the imaginary part of the matrix:
```
double[][] im = new double[n][n];
```

Creates the real part of the matrix:
```
double[][] re =  new double[n][n];
```

Creates the `Zmat` object, with the imaginary and the real part as instances:
```
Zmat A = new Zmat(im,re);
```

Creates the imaginary part of the matrix:
```
double[][] im1 = new double[n][n];
```

Creates the real part of the matrix:
```
double[][] re1 =  new double[n][n];
```

Creates the `Zmat` object, with the imaginary and the real part as instances:
```
Zmat A = new Zmat(im1,re1);
```

Performs the actual multiplication:
```
Zmat C = Times.o(A, B);
```

In Jampack the matrix multiplication method belongs to a `Times` class not a matrix class, the benefit from this is that the methods in the `Times` class can be `final` and/or `static`. If the

methods are `final` or `static` they are more efficient to access then non-final or non-static methods. If this is the right object-oriented design compared to JAMA's design were the matrix multiplication method belongs to the `Matrix` class will not be discussed here. But because an object-oriented design says that a class can have features (methods), we will in Chapter 5 use the JAMA's design approach when we create the classes.

In this thesis we only look at operations on real matrices, therefore Jampack's package will not be analysed any further, but in the Concluding Remarks we will discuss that the results presented in Chapter 3 and 4 also would apply to the Jampack package.

3.4 Notation and Terminology

In this section we will describe some notation we use and what test platform we do the benchmarking on.

The full definition of a Frobenius norm is:

$$\sqrt{\sum_{i=0}^{m-1}\sum_{j=0}^{n-1} Aij^2}$$

`A.times(B)` routine: matrix multiplication routine, where `A` and `B` are objects of the `Matrix` class.

The pure row-oriented `A.times(B)` routine: matrix multiplication routine, where `A` and `B` are objects of the `Matrix` class.

`A.times(B)` on ***Ab*** means that the `Matrix` object `A` contains a square matrix ***A,*** which is stored with Java arrays. And the `Matrix` object `B` contains the column vector ***b,*** which is stored with Java arrays. We then perform a product between the matrix ***A*** in the times method of A with the row vector ***b.***

`A.times(B)` on $\boldsymbol{b^T A}$ means that the `Matrix` object `A` contains a column vector $\boldsymbol{b^T}$, which is stored with Java arrays. And the `Matrix` object ***B*** contains the square matrix ***A,*** which is stored with Java arrays. We then perform a product between the column vector $\boldsymbol{b^T}$ in the times method of A with the matrix ***A.***

`A.times(B)` on ***AB*** means that the `Matrix` object `A` contains a square matrix ***A,*** which is stored with Java arrays. And the `Matrix` object `B` contains the square matrix ***B,*** which is stored with Java arrays. We then perform a product between the matrix ***A*** in the times method of `A` with the matrix ***B.***

When we say that a routine is pure row-oriented we mean that it does not traverse any column of `double[][]`, this does not apply to matrices with dimensions like *nx1* or *1xn*, see section 3.7.1.

When we say that a routine is pure column-oriented we mean that it does not traverse any rows of `double[][]`, this does not apply to matrices with dimensions like *nx1* or *1xn* , see section 3.7.1.

All the matrices in this chapter are square, except for the break even test in the last section. When doing a matrix multiplication, ***AB*** operation, both matrices are squared and have the same dimensions. That is if ***A*** has the dimension *mxn* and ***B*** has the dimension *nxp,* then we have that *m=n=p*.

`ArrayIndexOutOfBoundsException` is thrown to indicate that an array has been accessed with an illegal index. The index is either negative or greater than or equal to the size of the array [67].

`OutOfMemory` is thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no memory could be made amiable by the garbage collector [67].

3.5 Benchmarking

Measuring the execution time of a routine or a piece of code in the intention of comparing it to other routines or the same routine on different platforms under different conditions is called benchmarking. We measured the benchmarks under the same conditions. With no running operations in the background. The benchmarks were run multiple times checking for abnormalities.

The timing itself did not alter the original performance of the applications we measured. We measured the routines using `System.currentTimeMillis()` because we are interested in measuring the "wall-clock time". `System.currentTimeMillis()` returns the current time in milliseconds [67]. Further we have compiled all programs using `javac –O`, see Chapter 2, section 2.9.

All the benchmark results only consider the timings of the for-loops from where they begin to where they as the next code example show. There the `start` variable contains the time when we start the outermost for-loop and the `stop` variable contains the time where the outermost for-loops stops, the `executiontime` variable contains the time for executing the whole nested for-loop construction:

```
long start = System.currentTimeMillis();
for(int i = 0;i<m;i++){
     for(int j = 0;j<n;j++){
          for(int k = 0;k<p;k++){
               C[i][j] += A[i][k]*B[k][j];
          }
     }
}
long stop = System.currentTimeMillis();
long executiontime = start – stop;
```

The initialisation time is not considered in the benchmarks times, but we have always measured the initialisation times when we compared routines against each other. The initialisation time

cases, the initialisation times of the routines we compared were similar. When there was a difference, there was only a small difference that did not have any impact on the overall performance. The technical data of the platform we performed the benchmarking on is shown in Table 3-0.

Technical data for the testplatform	
Machine type:	SUN's, Ultra-5_10;sparc;sun4u
Operating System:	SunOS Release 5.8 Generic_108528-09
JDK version:	Solaris VM(build Solaris_JDK_1.2.2_09c, native threads, sunwjit)
Ram:	128 MB
CPU:	300MHz

Table 3-0
Technical data for the test platform.

This platform was choosen because it was available over the extensive time period the testing would be performed, without it being altered (upgraded or Java Development Kit (JDK)) version changing).

3.6 A Straightforward Matrix Multiplication routine

In this section we present a straightforward matrix multiplication routine. As stated in the introduction the matrix multiplication routine is sensitive to the underlying architecture for finding the optimal implementation, we therefore pay special attention to the implementations that does extensive column traversing versus those that do extensive row traversing. We expect on the basis of the Frobenius norm example that we will see a performance difference between such implementations.

A straightforward way to do a matrix multiplication using Java arrays is shown here [1]:

```
for(int i = 0; i<m;i++){
     for(int j = 0;j<n;j++){
          for(int k = 0;k<p;k++){
               C[i][j] += A[i][k]*B[k][j];
          }
     }
}
```

We will in some cases use SMM as an abbreviation for the Straightforward Matrix Multiplication routine.

There are six distinct ways of doing matrix multiplication, by interchanging the for loops in this routine we get:
Partial Column and Partial Row:

1. Loop-order (i,j,k) means that for each row of the matrix ***A*** we traverse all of the columns of matrix ***B***. We build matrix ***C*** row-by-row.
2. Loop-order (j,i,k) means that for each column of the matrix ***B*** we traverse all of the rows of matrix ***A***. We build matrix ***C*** column by column.

Pure Row:

3. Loop-order (k,i,j) means that we sweeps through ***C*** N times row-wise, accumulating one term of the inner product in each pass of ***C***'s elements. But here we traverse the columns of matrix ***A***, but not in the innermost for-loop, see section 3.8.
4. Loop-order (i,k,j) means that for each row of matrix ***A*** we traverses all of the rows of matrix ***B***, we build up matrix ***C*** row-by-row.

Pure Column:

5. Loop-order (j,k,i) means that for each column of matrix ***B*** we traverses all of the columns of ***A***, we build up matrix ***C*** column-by-column.
6. Loop-order (k,j,i) means that we sweeps through ***C*** N times column-wise, accumulating on term of the inner product in each pass.

In Table 3-1 we have the result of performing the straightforward matrix multiplication routine on ***AB***. From that table we see that a pure column matrix multiplication is the least efficient routine, the pure row matrix multiplication is most efficient. It is not acceptable in any case to use a pure column matrix multiplication. Partial row and partial column matrix multiplication has a performance, which is in between pure row and pure column matrix multiplication routines when it comes to performance. As with the Frobenius norm example, see section 3.1.2, we see that routines where we traverse a two-dimensional array pure column are much less efficient then routines where we traverse a two-dimensional array pure row.

Since the pure column routine with the loop-order (i,k,j) is more efficient on the operation ***AB*** then with loop-order (k,i,j) we choose to compare the straightforward matrix multiplication with JAMA's `A.times(B)` routine on the operations ***Ab, b^TA*** and ***AB***.

Matrix Multiplication						
Matrix Dimension	Partial Row/Column		Pure Row		Pure Column	
	(i,j,k)	(j,i,k)	(k,i,j)	(i,k,j)	(j,k,i)	(k,j,i)
80	66	72	66	63	100	99
115	208	233	178	174	295	299
138	331	341	298	257	468	474
240	2491	2617	1630	1538	4458	4457
468	27655	28804	13690	13175	56805	58351

Table 3-1
The straightforward matrix multiplication routine on input *AB* with different loop-orders. The time is in milliseconds (mS).

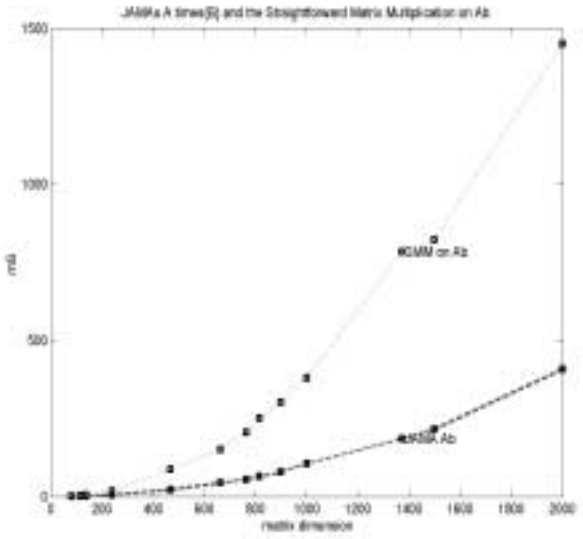

Figure 3-3
JAMA's `A.times(B)` and the straightforward matrix multiplication on input *Ab*.

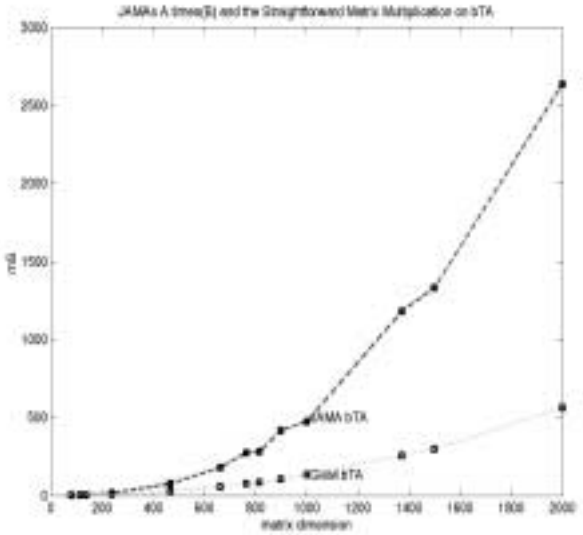

Figure 3-4
JAMA's `A.times(B)` and the straightforward matrix multiplication on input $b^T A$.

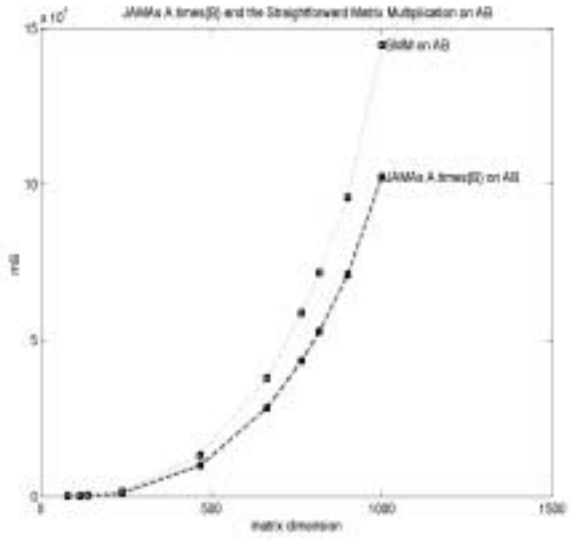

Figure 3-5
JAMA's `A.times(B)` and the straightforward matrix multiplication on input *AB*.

Figure 3-3 shows that JAMA's `A.times(B)` routine is more efficient then SMM on ***Ab***, the reason for that is because the for-loops are executed {m{n{1}}} for SMM compared to {1{n{p}}} for JAMA's routine, remember that $m=n=p$ and see section 4.3.3 for details on the {{{}}} notation, and JAMA's routine traverse its arrays `double[]` instead of `double[][]`, see section 3.7. Figure 3-4 shows that JAMA's routine is less efficient then SMM on $\boldsymbol{b}^T\boldsymbol{A}$, JAMA's routine is considered pure column and SMM's routine is considered pure row, that gives a clear indication on this difference. Figure 3-5 shows that JAMA's routine is more efficient then SMM on ***AB***, the for-loops are executed and initialised the same number of times for both routines, {m{n{p}}}, so the difference may be caused by that SMM traverses `double[][]` in the innermost for-loop, while JAMA's routine traverses `double[]`, see section 3.7.

We can see that SMM on input $\boldsymbol{b}^T\boldsymbol{A}$ has nearly the same efficiency as JAMA's `A.times(B)` routine on ***Ab*** for matrix dimension less then or equal to 1500. The reason for this is that they both are row-oriented routine and JAMA's routine execute and initialise the for-loops {1{n{p}}} and SMM execute and initialise the for-loops {1{n{p}}}.

In the next section we shall see how the designers of JAMA's `A.times(B)` routine has implemented its routine to see what optimisations they have done. That is how they have used the Java array construction and other techniques to make JAMA's routine more efficient on certain inputs then the SMM routine.

3.7 JAMA's `A.times(B)` implementation

In this section we give the complete code of JAMA's `A.times(B)` routine as it is in the `Matrix` class and then we explain both the algorithm and the techniques used in this routine to make it fast.

The implementation of the `times()` method in the `Matrix` class:

```
public Matrix times(Matrix B) {
      Matrix X = new Matrix(m,B.n);
      double[][] C = X.getArray();
      double[] Bcolj = new double[n];
      for (int j = 0; j < B.n; j++) {
          for (int k = 0; k < n; k++) {
               Bcolj[k] = B.A[k][j];
          }
          for (int i = 0; i < m; i++) {
               double[] Arowi = A[i];
               double s = 0;
               for (int k = 0; k<n; k++) {
                    s += Arowi[k]*Bcolj[k];
               }
               C[i][j] = s;
         }
      }
      return X;
}
```

In this routine the result matrix is constructed column-by-column, loop-order (j,i,k). Therefore the first element in the resulting matrix position `C[0][0]` is constructed by multiplying the first column of the matrix ***B*** with the first row of matrix ***A***. The second element in the resulting matrix, position `C[1][0]` is constructed by multiplying the second row of matrix ***A*** with the first column of matrix ***B***. So the first column of ***C*** is constructed by multiplying all of the rows of ***A*** with first column of ***B***, then the second column of ***C*** is created by multiplying all of the rows of ***A*** with the second column of ***B***. To create all of the columns of matrix ***C,*** we do this for all of the columns of matrix ***B***.

Optimising techniques used in JAMA's `A.times(B)` routine:

- The most obvious difference between JAMA's `A.times(B)` routine and the straightforward matrix multiplication routine, is that it does not traverse `double[][]` in its innermost for-loop, it traverse `double[]` instead. Traversing `double[][]` instead of `double[]` will in some cases be twice as expensive [36]. In the following code example we traverse the array with `double[]`. Instead of a `double[][]` indexing we do a `double[]` indexing. We get a more efficient traversing, with an average difference of a factor of 30%, as shown in Figure 3-6. For larger matrices the difference is even greater. When tested matrix dimension *m=n=p=3000*, we achieved a difference of 1400% (a factor of 14) row `double[][]` had 11075 mS and row `double[]` had 782 mS, which is significant.

This next code example is the Forbenius norm but we traverses `double[]` in the innermost for - loop:

```
for(int i = 0;i<m;i++){
        double[] a = array[i];
        for(int j = 0;j<n; j++){
             s+=a[j];
        }
}
```

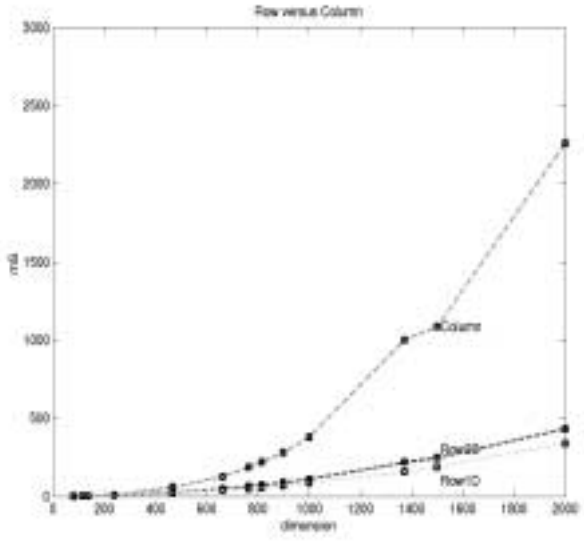

Figure 3-6
The time for traversing row-wise and column-wise `double[][]` (Row 2D) and `double[] (Row 1D)`.

There are of course a cost of getting the arrays from `double[][]` to a `double[]`. For the rows of a `double[][]` we only get the reference to a row array object but for getting a column we have to copy all the elements from `double[][]` to a new array `double[]` object.

Getting a row, `double[]`, from a `double[][]` is like this:

```
double[] Arowi = A[i];
```

This can be done since a two dimensional array consists of arrays of arrays where each array is an object with its own unique reference. The `double[] Arowi` is only holding the same reference to the same array object that is stored in `A[i].` We are not creating any new array objects only declaring and initialising a reference type of `double[]` which is much less time consuming then creating a new array object. Then we traverse `Arowi[i]` instead of `A[j][i].`

- Getting a column, `double[]`, from a `double[][]` is like this:

```
 int i = 0;
double[] Bcolj = new double[n];
for(int j = 0;j<n;j++){
     Bcolj[j] = B[j][i];
}
```

This is clearly more time consuming then getting a row from a `double[][]`.

- Another thing that boost the performance is that the Java array of `Matrix` object ***B*** is stored in a temporary local variable and is not accessed by a method call.

Altering a code in this manner to achieve a more efficient code is called optimisation. Optimisation comes with a price, a more complex code that is clearly illustrated with the implementation of the JAMA's `A.times(B)` routine compared to the implementation of the straightforward matrix multiplication routine.

Compared to the straightforward matrix multiplication routine we are only creating and heap allocating one object besides the resulting matrix, this is an array object that we store the columns of the ***B*** matrix. We do not however create any new objects inside the for-loops.

3.7.1 JAMA's `A.times(B)` routine on input Ab versus b^TA

We are interested in finding how JAMA's `A.times(B)`works for all inputs, but we have no way of testing this routine for all inputs. Therefore we test JAMA's `A.times(B)` routine on four cases $\boldsymbol{Ab}$, $\boldsymbol{b^TA}$, $\boldsymbol{AB}$ (where both matrices are square and have the same dimensions) and a break even test. The reason for testing $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$ as inputs on JAMA's routine is because we are expecting consistent time performance running these two inputs, since both $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$ have the same complexity O($m \cdot n$) and the same number of arithmetic operations. Having knowledge of the layout of Java arrays and of the `A.times(B)` routine of JAMA we know that $\boldsymbol{Ab}$ is as pure row-oriented routine traversing only the rows of the matrices involved. And that $\boldsymbol{b^TA}$ is a pure column-oriented operation. We see that pure row traversing was more efficient then pure column traversing when comparing $\boldsymbol{Ab}$ to $\boldsymbol{b^TA}$, as shown in Figure 3-7. We will in Chapter 4, give a detailed explanation on this difference.

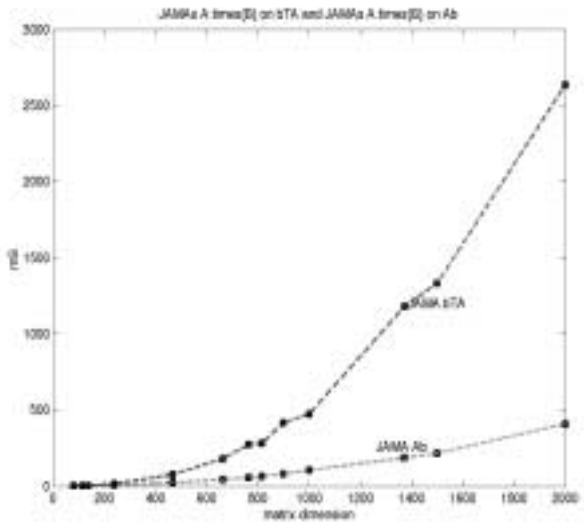

Figure 3-7
JAMA's `A.times(B)` routine on input Ab and b^TA.

For matrix dimension *n = 1500*, we have that the operation $\boldsymbol{b}^T\boldsymbol{A}$ is 5 times slower then the $\boldsymbol{Ab}$ operation, while for *n = 2000* we have that the $\boldsymbol{b}^T\boldsymbol{A}$ operation is 7 times slower then the $\boldsymbol{Ab}$ operation.

JAMA's `A.times(B)` routine on input $\boldsymbol{Ab}$ the $\boldsymbol{b}$-vector is still a `double[][]`:

```
-double[][] b = new double[n][1];
```

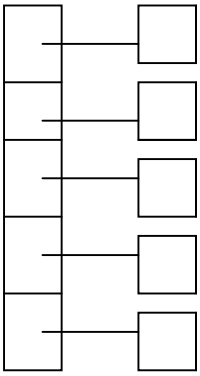

Figure 3-8
A Java array with dimension *nx1*.

JAMA's `A.times(B)` on input $\boldsymbol{b}^T\boldsymbol{A}$ the $\boldsymbol{b}$-vector is still a `double[][]`:

```
-double[][] b = new double[1][n];
```

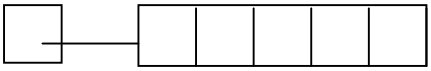

Figure 3-9
A Java array with dimension *1xn*.

We tested the constructions in Figure 3-8 and Figure 3-9 to see if one of them was more time consuming then the other. The largest matrix has the dimension *2000x2000* that leads us to a *2000x1* matrix and a *1x2000* matrix. We measured no time differences in traversing these two arrays. Further we tested *n* up till we caught an `OutOfMemoryError`, see section 3.4, but we did not measure any time or differences between them. This shows that for the testing environment we can consider them equal when it comes to time performance.

3.8 The pure row-oriented `A.times(B)` implementation

It is clear from the Frobenius norm example, Figure 3-2, and JAMA's `A.times(B)` routine on input $\boldsymbol{Ab}$ versus $\boldsymbol{b}^T\boldsymbol{A}$, Figure 3-7, that traversing the rows of Java arrays is less time consuming then traversing the columns of the Java arrays. The next step is therefore to consider a pure row-oriented `A.times(B)` routine to see if the difference between the operations $\boldsymbol{Ab}$ and $\boldsymbol{b}^T\boldsymbol{A}$ is

getting smaller and to see if we achieve an overall better performance than that of JAMA's `A.times(B)` routine. JAMA's `A.times(B)` still traverses the columns of matrix ***B*** this has an impact doing $\boldsymbol{b^TA}$ on JAMA's `A.times(B)`. If we eliminate all column traversing and just traverse the rows we hope we will achieve a total better performance.

Since we have two ways of performing a matrix multiplication pure row-wise, we implemented both of them using the same optimisation techniques used in JAMA's `A.times(B)` routine and we also used some additional optimising techniques. The result of testing the loop-order (i,k,j) against loop-order (k,j,i) on input ***Ab***, $\boldsymbol{b^TA}$ and ***AB*** is in Table 3-2.

The pure row-oriented `A.times(B)` routine						
Matrix Dimension	***Ab***		$\boldsymbol{b^TA}$		***AB***	
	(i,k,j)	(k,i,j)	(i,k,j)	(k,i,j)	(i,k,j)	(k,i,j)
80	1	1	0	0	30	33
115	3	3	1	2	101	109
138	3	4	1	2	166	176
240	12	18	4	5	848	962
468	44	88	18	17	7875	8741
663	102	207	35	34	23059	24909
765	138	300	46	59	35863	38528
817	162	364	64	64	43839	46964
900	196	476	74	79	56127	62190
1000	246	574	89	95	78606	86321
1374	454	1358	182	157	NA	NA
1500	550	1644	191	206	NA	NA
2000	966	3235	333	396	NA	NA

Table 3-2
The pure row-oriented `A.times(B)` routine on input *Ab*, b^TA and *AB* with loop-orders (i,k.j) and (k,i,j), the time is in miliseconds (mS).

NA means Not Applicable which means that the routines took so long to measure that we chose to stop/abondon these measurements.

The routine with loop-order (j,k,i) was more efficient then the routine with loop-order (k,j,i) on the ***Ab***, $\boldsymbol{b^TA}$ and ***AB*** operations.

This is the pure row-oriented implementation with loop-order (k,i,j), without the initialisation:

```
for(int k = 0;k<n;k++){
     Browk = B[k];
     for(int j = 0;j<m;j++){
          a = A[j][k];
          Crowj = C[j];
          for(int i =0;i<Bn;i++)
               Crowi[j] +=a*Browk[i];
          }
          C[j] = Crowi;
     }
}
```

We will not give a detailed explanation of this difference, but indicate that the difference between the routine with loop-order (i,k,j) and (k,i,j) may be caused by this routine is not totally row-wise. It actually traverse the columns of matrix ***A,*** with this construction in the loop-body of the second for-loop: `a = A[j][k];`, see the code example above. The difference between loop-order (k,i,j) versus loop-order (i,k,j) on input ***Ab*** is caused by the column traversing for loop-order (k,i,j) and that the for-loops are executed {m{n{1}}} compared to {1{n{p}}} for loop-order (i,k,j), see Chapter 4 and section 4.3.3. The difference between loop-order (k,i,j) versus loop-order (i,k,j) on input $\boldsymbol{b^TA}$ is caused by the column traversing for loop-order (k,i,j). The difference between loop-order (k,i,j) versus loop-order (i,k,j) on input ***AB*** is caused by the column traversing of loop-order (k,i,j).
This is the implemented routine of the pure row-oriented `A.times(B)` routine with loop-order (i,k,j):

```
public Matrix times(Matrix B){
          Matrix X = new Matrix(m,B.n);
          double[][] C = X.getArray();
          double[][] BA = B.A;
          double[] Arowi, Crowi, Browi;
          int Bn = B.n, Bm = B.m;
          double a = 0.0;
          int i = 0, j = 0, k = 0;
          for(i = 0;i<m;i++){
               Arowi = A[i];
               Crowi = C[i];
               for(j = 0;j<Bm;j++){
                    Browi = BA[j];
                    a = Arowi[j];
                    for(k = Bn;--k>=0;){
                         Crowi[k] += a*Browi[k];
                    }
               }
          }
          return X;
}
```

In this routine we use partial storage which means that we cannot in *n* iterations, accessing only rows construct a complete value for an element in the resulting matrix. We must multiply the value in position `A[0][0]` in the first row of ***A*** with all of the elements of the first row of ***B***,

accumulating the result into the first row of the resulting matrix ***C*** in the same position as the elements of the row of matrix ***B***. Then we multiply the values in position `A[0][1]` in the first row of ***A*** with the second row of ***B***, again accumulating the result in the first row of the result matrix ***C***. When we accumulate the result into the column of the resulting column, we perform an addition with the values already accumulated into the result matrix ***C***. We have created the first row of ***C,*** when we have done the same for each element in the first row of ***A***. Then we do the same for the second row of the matrix ***B***, to create the second row of the matrix ***C***. Finally we would have the whole resulting matrices. We only create one object in this routine and that is the resulting `Matrix` object.

Optimizing techniques used in the pure row-oriented `A.times(B)` routine:

- JAMA's `A.times(B)` and this pure row-oriented `A.times(B)` routine have three nested for-loops and both only traverses `double[]` in the innermost for-loop.

- If we take a closer look at this pure row-oriented `A.times(B)` we can see that we have no declarations of variables, neither in the for-loop nor in the loop-bodies. We declare all the locale variables outside the loops and we only do it once.
- We can also see that we have different loop-body constructions. How much impact these differences will have when we compare it to JAMA's `A.times(B)` routine when it comes to performance is important and will be further analysed in Chapter 4.

- We only initialise local variables here, so we do not heap allocate anything (except the resulting `Matrix` object), neither do we declare any new variables in the for-loops to be placed on the stack. Since JAMA's `A.times(B)` routine uses one more array object than this pure row-oriented `A.times(B)` routine it is not so memory demanding as JAMA's routine.

All these modifications that we have made is for the sake of consistent results on inputs (***Ab*** and $\boldsymbol{b^TA}$), see section 3.8.1 and achieving a overall better performance then JAMA's routine, see section 3.9.

We now have a routine that does not traverse the columns of any matrices involved and we have eliminated all unnecessarily declarations and initialisations. Before we have done any time measuring it may seem that we have a more efficient routine.

3.8.1 The pure row-oriented `A.times(B)` on input *Ab* versus b^TA

The pure row-oriented `A.times(B)` on input ***Ab*** and $\boldsymbol{b^TA}$ only traverse the rows of the underlying structure of Java arrays. In Figure 3-10 we see that there is still a difference between them. In Chapter 4 we will give a more detailed explanation on this difference.

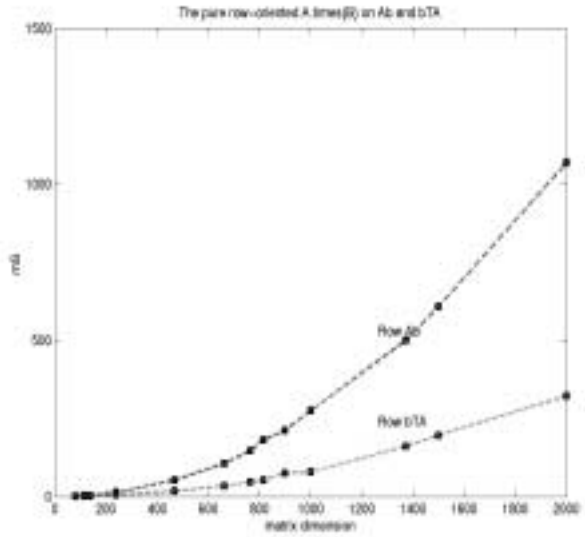

Figure 3-10
The pure row-oriented `A.times(B)` routine on input *Ab* and b^TA.

For matrix dimension *n = 1500*, we have that the b^TA operation is 3.33 times slower then the ***Ab*** operation, while for *n = 2000* we have that the b^TA operation is 4 times slower then the ***Ab*** operation. Compared to the difference between JAMA's `A.times(B)` routine on input ***Ab*** versus b^TA, we have narrowed the gap between the ***Ab*** and b^TA operations for the pure row-orinted `A.times(B)` routine, see section 3.7.1.

3.9 The pure row-oriented `A.times(B)` versus JAMA's `A.times(B)`

After creating the pure row-oriented `A.times(B)` routine we compare it to JAMA's `A.times(B)` routine. We compare the pure row-oriented routine to JAMA's routine on input ***Ab***, b^TA and ***AB***.

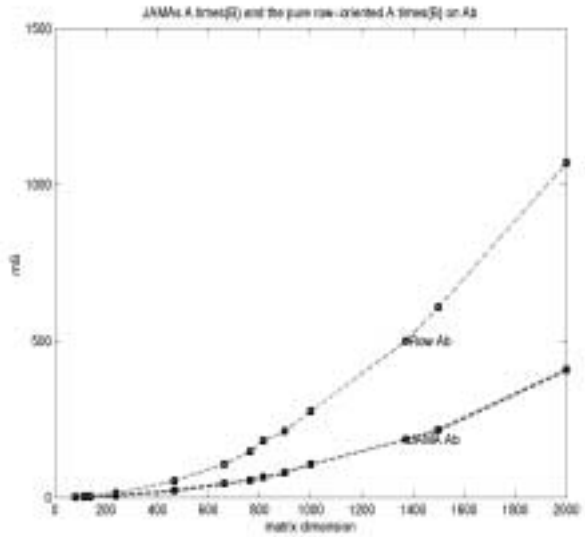

Figure 3-11
JAMA's `A.times(B)` and the pure row-oriented `A.times(B)` on input *Ab*.

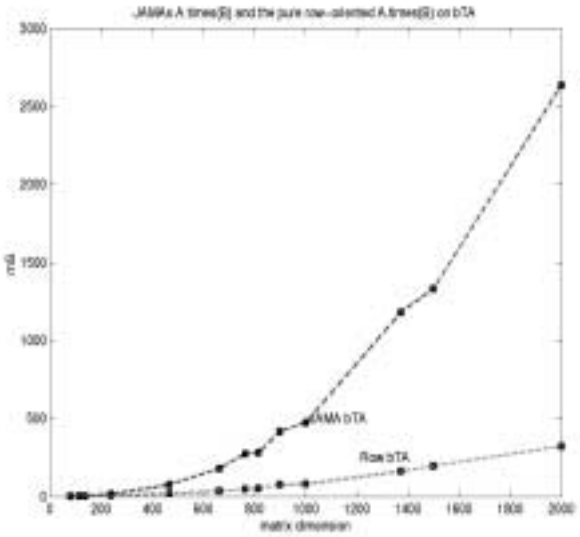

Figure 3-12
JAMA's `A.times(B)` versus the pure row-oriented `A.times(B)` on input b^TA.

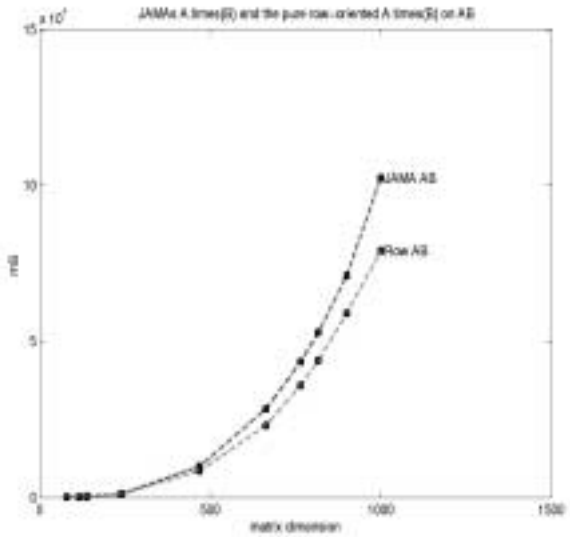

Figure 3-13
JAMA's `A.times(B)` versus the pure row-oriented `A.times(B)` routine on input *AB*.

First we compared JAMA's `A.times(B)` routine with the pure row-oriented `A.times(B)` routine on input ***Ab,*** as shown in Figure 3-11. JAMA's routine on input ***Ab*** is more efficient then the pure row-oriented routine on input ***Ab*** with an average factor of two**.** There is a significant difference between JAMA's `A.times(B)` routine versus the pure row-oriented `A.times(B)` routine on $\boldsymbol{b^TA}$ as shown in Figure 3-12. It is a difference with an average factor of 7. The third comparison is on input ***AB***, here the pure row-oriented `A.times(B)` routine on input ***AB*** as slightly better than JAMA's routine on input ***AB*** and with an average of 30% better performance, as shown in Figure 3-13. The first comparison on the ***Ab*** operation is comparing two pure row-oriented routines, the difference can be explained that the for-loops of JAMA's `A.times(B)` is executed {1{n{p}}} compared to {m{n{1}}} for the pure row-oriented `A.times(B)` routine, again see section 4.3.3. The next comparison was between a pure column routine (JAMA) and a pure row routine (the pure row-oriented) and again we see a

clear difference between the execution time for such routines and again the pure column routine was much less efficient then the pure row routine. The third and last comparison is more difficult to give a short explanation on, one reason might be that we actually traverses the columns of the $\boldsymbol{B}$ matrix p times, for a detailed explanation see section 4.3.4.

The reason for the above differences will be explained in greater detail in Chapter 4.

It is also interesting finding when the pure row-oriented `A.times(B)` routine starting on $\boldsymbol{AB}$, is achieving a better performance then JAMA's `A.times(B)` routine starting on $\boldsymbol{AB_p}$. Where $\boldsymbol{B_p}$ has the dimension nxp where $p = 1,2,3,\ldots,n$. The reason is that we get a clearer idea on the overall performance of the pure row-oriented routine compared to JAMA's routine and as we shall see we have an early break even. If we increase p from *1*, for both the routines, we are interested in when the pure row-oriented routine is more efficient then JAMA's routine on $\boldsymbol{AB_p}$. We are expecting a break even because by adding columns to the $\boldsymbol{B_p}$ matrix we are getting closer to a $\boldsymbol{AB}$ operation and for the $\boldsymbol{AB}$ operation the pure row-oriented routine is more efficient then JAMA's routine.

The Break Even results													
Matrix Dimension	80	115	138	240	468	663	765	817	900	1000	1374	1500	2000
p	13	9	7	5	3	3	5	7	4	6	5	4	6

Table 3-3
Shows for what p we have a break even.

Table 3-3 shows us that we had an early break even for when the pure row-oriented `A.times(B)` routine was more efficient and that gives us an indication that the pure row-oriented routine has a better overall performance then JAMA's `A.times(B)` routine on inputs different from the cases: $\boldsymbol{Ab}$, $\boldsymbol{b^TA}$ and $\boldsymbol{AB}$.

What we have not done here is to experiment with all kinds of dimensions, that matrices in a matrix multiplication could have. We cannot say that JAMA's routine has the best overall performance or that the pure row-oriented routine has the best overall performance. We only compared them on these cases, but based on the result we got from them, we tried to indicate an overall performance. After we reached break even, we tested both routines with an increasing p til we had input $\boldsymbol{AB}$, the pure row-oriented `A.times(B)` routine had a better performance then JAMA's `A.times(B)` routine.

It is clear for the timings done in this chapter that column traversing is time consuming, therefore we are trying to find a those implementations that takes the row-wise layout into consideration. And we can see from those implementations that we achieve more efficient routines from those that do not take the row-wise layout into consideration. We focused on JAMA's `A.times(B)` routine because of the simple fact that it is a method in a package that is into consideration to become *the* Matrix class in Java, it is therefore natural to consider its implementations of different routines using Java's native arrays. But how much the row-wise layout plays a part in all the routines in JAMA's package we considered a too big effort, and in analyzing the matrix multiplication routine we hope to unveil all the parts of the use of Java's native arrays. In the next chapter we will establish a model that clearly shows the impact of the row-wise layout but it will also show the impact of the different constructions used in the pure row-oriented `A.times(B)` routine versus the constructions used in JAMA's `A.times(B)`

routine. Some of the things we will unveil is particular for the matrix multiplication routine other things are more general and are the same for all routines that uses Java's native arrays as its underlying storage format.

Chapter 4 The Computational Model

4.0 Introduction

In this chapter we establish the model we use for explaining the differences between JAMA's `A.times(B)` routine and the pure row-oriented `A.times(B)` routine on the inputs ***Ab***, $\boldsymbol{b^TA}$ and ***AB***, as we showed in Chapter 3. We experimented with two models before we established the one we used in this chapter, which was surprisingly difficult to establish. The model we implemented still has its weaknesses. The model we implemented only shows the qualitative behavior of the routines. But we cannot predict the quantitative time component, the time component that are missing is significant, a factor of 1.5 to 2. This will undoubtedly limit this model, but still it will unveil some of the reasons for the differences in performance.

We are explaining the following comparisons of the `A.times(B)`routine:

- The pure row-oriented on input ***Ab*** versus $\boldsymbol{b^TA}$.
- The pure row-oriented versus JAMA's on input ***Ab***.
- The pure row-oriented on input $\boldsymbol{b^TA}$ versus JAMA's on input ***Ab***.
- The pure row-oriented versus JAMA's on input $\boldsymbol{b^TA}$.
- The pure row-oriented on input ***Ab*** versus JAMA's on input $\boldsymbol{b^TA}$.
- JAMA's on input ***Ab*** versus $\boldsymbol{b^TA}$**.**
- JAMA's versus the pure row-oriented on input ***AB***.

These comparisons are interesting because the routines have the same number of arithmetic operations and complexity, we therefore expect (from a 'users' view) that they have the same execution time, but as we saw in Chapter 3 we did not have similar execution time for these routines.

4.1 The First Model

The first idea for a model, explaining the differences, was to split all the constructions in the for-loops and loop-bodies for both (JAMA's and the pure row-oriented) routines. Then we would measure their relative performance on the test platform and summarize them multiplied with the number of times they got executed for various inputs and then compare them. In this process we localize those constructions that made a significant difference, either executed so many times that they make the difference or have a high execution cost.

We split all the constructions like this example `int i = 0;`is split up in `int i;` and `i=0;` that is one declaration and one initialization.

The idea for splitting the constructions that appear inside the for-loops was because we wanted to say something about how much a declaration for a primitive type or a reference type costs on one platform and how much they would cost on another. Also how much a multiplication costs one platform compared to another. If the cost of an `int`, is larger on one platform then another. We would also see how much impact each construction has on the overall performance when we compare routines against each other. In that process we also had to make a benchmark suite

that could be run on multiple platforms. Measuring the relative cost of a construction is called micro benchmarking and micro benchmarking seems to be a real problem. The main reason for this maybe the compiler or that we did not find a reliable way to do micro benchmarking. Next we show an example to illustrate this.

In this example we will try to measure the cost of an `int i = 0`; We will therefore measure *n* `int i = 0;` and the time we get we divide by the number *n*.

```
long start = System.currentTimeMillis();
int i1 = 0;
int i2 =0;
.
.
.
int in = 0;
long stop = System.currentTimeMillis();
long executiontime = stop – start;
```

The value of *n* must be very large to measure any time at all. When *n* gets large there is also a possibility that we get a memory problem, because we have to store all the variables that is declared and initialize in the stack. The memory handling can have an impact on the time performance. Finding the right balance is not easy and we have to find the balance for each construction since they usually have different costs. We would probably also have to find the balance for each software and hardware.

We can state the above code example as the function:

$$constructiontime(n) \qquad (4\text{-}0)$$

(4-0) gives the time of executing a Java construction like `int i = 0;` *n* times. And we define the relative cost of executing that particular construction one time as:

$$constructiontime(n)/n \qquad (4\text{-}1)$$

The time given by *(4-1)* should be similar for *all n,* which we did not achieve. The relative difference was in some cases 100% to 200%. Therefore benchmarking constructions like we did above was not found reliable.

4.2 The Second Model

In this next approach we did not split up the constructions like we did in the for-loops of both routines, we measured the constructions as they appear in the routine. For example the construction `double s = 0;` was not split up in one declaration and one initialization, we tried to find the relative cost of the whole expression `double s = 0;`. This may lead to some loss of information, but it may give the impact of these constructions for various inputs.

We also abandon this model, the main reason for that was that we measured the two loop-bodies for JAMA's `A.times(B)` and the pure row-oriented `A.times(B)` in their innermost for-loops to be equal. This means they had the same relative cost on the test platform, but on input $\boldsymbol{AB}$ there was an average different between JAMA's and the pure row-oriented routine of about 30%, which means the pure row-oriented routine are more efficient. Since most of the work is done in the innermost for-loop, this difference in time must also be because there is a difference in the innermost constructions, contradicting that they should have the same execution time. That there is a difference between the innermost constructions is clearly illustrated in section 4.4.1 (Table 4-1).

4.3 The Concept of the Implemented Model

In this section we illustrate the concept of the implemented model. We are interested in finding how much impact the construction differences has for the two routines, we must also see how many times they are executed for various inputs. For example the innermost for-loop for both $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$, and the pure row-oriented `A.times(B)` routine, will be executed the same number of times which is O ($m \cdot n$). But that is not the case for the first and second for-loop for $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$. This leads us to the implemented model, for explaining the performance differences.

This next code example gives us the motivation for implementing the model:

```
for(int i = 0;i<m;i++){A;
     for(int j = 0;j<n;j++){B;
          for(int k = 0;k<p){C;}
     }
}
```

The code above we define in this example as the *whole* routine. Next we split the whole routine into three benchmark components. The benchmark components are:

Benchmark X:
```
for(int i = 0;i<m;i++){A;}
```

Benchmark Y:
```
for(int i = 0;i<m;i++){
     for(int j = 0;j<n;j++){B;}
}
```

Benchmark Z:
```
for(int i = 0;i<m;i++){
     for(int j = 0;j<n;j++){
          for(int k = 0;k<p;k++){C;}
     }
}
```

We executed these three benchmarks as many times as they would be executed in the whole routine, but of course subtracting the extra overhead we get from executing the loops multiple times. To do this subtracting, we measured these three benchmarks without the body-loops. In the theoretical model we expect an empty for-loop to be executed as many times, as it would be if it were not empty. The constructions `A`, `B` and `C` are *dummy* constructions and simulate any real Java code.

It is also important to notice that we embedded the constructions (`A`, `B` and `C`) in for-loops and that they get executed the same way as in the whole routine, because if we replace `A`, `B` and `C` with the constructions of JAMA's and the pure row-oriented `A.times(B)` routines we will introduce Java arrays, and with the same indexing of arrays. This is important because if they were not, we would loose the impact of row-wise traversing compared to column-wise traversing of arrays.

First we have to look at the different implementations of JAMA's `A.times(B)` routine and the pure row-oriented `A.times(B)` routine. They use different constructions in their implementation, see Chapter 3, section 3.7 and 3.8 respectively. We have to find out if these construction differences have anything to say when it comes to the difference in performance between the two routines.

4.3.1 The Cost of Executing the Benchmarks

What we show here, is that if we split the whole routine into minor parts and measure those parts it is the same as if we had measured the whole routine. We take an arithmetic look at this. The theoretical model will predict the time to execute the for-loops.

First definitions:

- *m, n and p* is the number of executions.
- b_1 is the cost for initialisation of the for-loop (`int i = 0;` or `i = 0;`).
- b_2 is the cost of looping the loop (`i<m`, `i++` and `--i>=0`).
- *A* is the cost of executing the innermost loop-body of Benchmark X.
- *B* is the cost of executing the innermost loop-body of Benchmark Y.
- *C* is the cost of executing the innermost loop-body of Benchmark Z.

TimeX gives us the cost of executing Benchmark X and we define it as follows:

$$TimeX = b_1 + m*(b_2 + A) \qquad (4\text{-}2)$$

TimeY gives us the cost of executing Benchmark Y and we define it as follows:

$$TimeY = b_1 + m*[b_2 + b_1 + n*(b_2 + B)] \qquad (4\text{-}3)$$

TimeZ gives us the cost of executing Benchmark Z and we define it as follows:

$$TimeZ = b_1 + m*[b_2 + b_1 + n*(b_2 + b_1 + p*(b_2 + C))] \qquad (4\text{-}4)$$

BenchmarksTime is the sum of the estimated cost of executing these three benchmarks consecutively:

$$BenchmarksTime = TimeX + TimeY + TimeZ \quad (4\text{-}5)$$

Time executing the whole routine is:

$$TotalTime = b_1 + m*[b_2 + A + b_1 + n*(b_2 + B + b_1 + p*(b_2 + C))] \quad (4\text{-}6)$$

LoopTime contains the components in the *BenchmarksTime* that is executed multiple times

$$LoopTime = b_1 + m*[b_2 + b_1 + n*(b_2 + b_1 + b_2 + b_1 + b_2 + b_1 + p*b_2)] \quad (4\text{-}7)$$

ModelTime we get from subtracting the *LoopTime* from the *BenchmarksTime:*

$$ModelTime = BenchmarksTime – LoopTime = b_1 + m*[b_2+A+b_1+n*(b_2+B+b_1+p*(b_2 + C))] \quad (4\text{-}8)$$

We have established the 'right' theoretical model since we are executing the same constructions the same amount of time in the benchmarks routines, *(4-8)*, as we would in the whole routine, *(4-6)*. The problem is how the compiler deals with less complex for-loops. The problem by doing the benchmarks is if the compiler sees that many of these constructions are executed multiple times but not used either inside or outside the for-loops, it may decide not to execute them at all in the Java Virtual Machine, which is called dead code elimination [72]. The reason for describing this is that the sum of the benchmarks, that we establish in section 4.3.2, is lower then the cost of the whole routines, the pure row-oriented and JAMA's `A.times(B)` see Chapter 3, section 3.8 and 3.7 respectively, with a factor of 1.5 to 2. We then had to normalise the benchmark values to see if the benchmark timings and the whole routine had the same *behaviour*. Next we explain more on the concept of normalisation:

We want to explain the two routines behaviour for various inputs, to do that we split up the routines in several benchmark components and we measured their performance and compared them to the whole routine. The most important thing to notice is that there is an uncertainty measuring a whole routine compared to the benchmark components. Because we have measured them under different conditions we are not expecting that they behave *exactly* the same as the whole routine, but we expect them to have approximately the same behaviour as the whole routine. With under different condition we mean that they are not measured as a part of the whole routine.

An example of the impact of normalisation is to see if a given routine have the same behaviour on different platforms, because one of the platforms are considered faster then the other, we get a timing difference between the values for the routines. We can still see if they have the same behaviour, that is the same increase/decrease in speed for the same inputs. We discover that by eliminating the time difference, by normalisation all the values for one platform. So if we compare the normalised value for one platform and the values for the other platform, we see if those values follow each other, that they are approximately equal. If that is so, we can conclude that they responds to the same input the same way, but that the underlying software and hardware effects forces us normalise to see that.

This is what we have when we split up the whole routine into different benchmarks, here the sum of the benchmarks was less then the sum of the whole routine, but after normalisation we saw that the benchmarks and the whole routine had nearly same behaviour for the same inputs.

This next example illustrates how we normalise the benchmark time (*ModeTime*):
Matrix Dimension is 765. We choose this Matrix Dimension for all the normalised values we present in this chapter.
JAMA's `A.times(B)` input on ***Ab*** routine is 200mS for matrix dimension 765.
The benchmark time on input ***Ab*** is 175 mS.
We then calculate the ratio:
Normvalue = 200/175 = 1.143 (rounding off)
We multiply this ratio value with all the other values for the benchmarks results. What we then get is a set of normalised values that we compare with the whole routines measured values. If those sets of values follow each other closely we would have done the right benchmarking.

In Appendix A we present the loop-overhead and the benchmark timings, these times are *not* normalised, but the actual benchmark timings. We present the normalised values in the Figures 4-0 to 4-5 and 4-12 to 4-13 when we compare the set of normalised values to the values of the whole routine (JAMA's and the pure row-oriented `A.times(B)` routine).

What we then get is a qualitative explanation not a quantitative explanation. That means the benchmark timings show that same behaviour, which is the same as the whole routine but we do not have the similar relative speed. We have created a theoretical model, but that does not mean it is a strong model that can be carried out with practical examples. We still have enough information to pinpoint the bottlenecks in the benchmarks.

The reason for this difference might be one of the following:

- We have missed a significant time component.
- We have done poor benchmarking, which means we have done the benchmarking under wrong conditions compared to the whole routine.
- The compiler may deal with less complex benchmark in another way then the whole routine. It may be easier for the compiler to optimise the benchmarks then the whole routine.

We have shown with the theoretical model, that we have not missed any significant time component. We have done *all* the benchmarking under the same conditions (hardware and software) and tested several times. So the only reason could be that the compiler and Java Virtual Machine deals with less complex routines, different then more complex routines, what we call optimising compiler [72, 31]. An optimising compiler can perform dead code elimination for example to speed up class load time [79].

This next example shows the problem of optimizing compilers:

```
long start = System.currentTimeMillis();
for(int i = 0;i<n;i++){
     int ii = 0;
}
long stop = System.currentTimeMillis();
long executiontime = stop – start;
```

We are also interested in finding the relative cost of `int ii = 0;` not the whole construction. Therefore we must also measure the for-loop construction with an empty for-loop, subtracting that cost from the whole cost of both the for-loop construction and the `int ii = 0;` construction. The main problem with this benchmarking are that we cannot be sure that an empty for-loop is executed *n* times. There might be dead code elimination and we cannot be certain that since the variable `ii` is not used anywhere that it is actually executed *n* times in the for-loop. Dead code elimination is just one of many optimizing techniques a compiler could use, but we will not in this thesis dwell on that any further.

We have shown the measured time for the whole routine and by summarising the three benchmarks for either JAMA's `A.times(B)` routine or the pure row-oriented `A.times(B)`routine we should have the same time as the whole routine. We have already stated that it is not the case, therefore we must see if the times of the summarised benchmarks are consistent with the whole routine.

4.3.2 The Benchmarks

In this section we present the benchmarks we are comparing:

The Benchmarks

Pure Row-Oriented

Benchmark X:
```
for(i = 0;i<m;i++){
     Arowi = A[i];
     Crowi = C[i];
}
```

Benchmark Y:
```
for(i=0;i<m;i++){
     for(j=0;j<n;j++){
          Browi = B[j];
          A = Arowi[j];
     }
}
```

Benchmark Z:
```
for(i = 0;i<m;i++){
 for(j = 0;j<n;j++){
  for(k=p;--k>=0;){
       Crowi[k] += a*Browi[k];
          }
     }
}
```

JAMA

Benchmark X:
```
for(int j = 0;j<m;j++){
     for(int k = 0;k<n;k++){
          Bcolj[k] = B.A[k][j];
     }
}
```

Benchmark Y:
```
for(int j=0;j<m;j++){
     for(int i=0;i<n;i++){
         double[] Arowi = A[i];
         double s = 0;
         C[i][j] = s;
     }
}
```

Benchmark Z:
```
for(int j = 0;j<m;j++){
 for(int i = 0;i<n;i++){
     for(int k = 0;k<p;k++){
          s+=Arowi[k]*Bcolj[k];
          }
     }
}
```

The obvious difference between these two benchmarks suites for JAMA's `A.times(B)` routine and the pure row-oriented `A.times(B)` routine, is that the for-loops are optimised for the pure row-oriented routine. We do not declare the running index variable in the for-loop and in the innermost for loop we have another declaration of the for-loop. The other difference is that they have different constructions in their loop-bodies. These differences will have an impact on performance for JAMA's routine and the pure row-oriented routine when compared for equal inputs.

4.3.3 Comparing the Benchmarks

In this section we use the established model to indicate the differences between JAMA's `A.times(B)` routine and the pure row-oriented `A.times(B)` routine.

We do the following benchmarks comparisons:

- For JAMA's `A.times(B)` we compare Benchmark X for $\boldsymbol{Ab}$ against Benchmark X for $\boldsymbol{b^TA}$, then we compare Benchmark Y for $\boldsymbol{Ab}$ against Benchmark Y for $\boldsymbol{b^TA}$ and finally we compare Benchmark Z for $\boldsymbol{Ab}$ against Benchmark Z for $\boldsymbol{b^TA}$.

- For the pure row-oriented `A.times(B)` we compare Benchmark X for $\boldsymbol{Ab}$ against Benchmark X for $\boldsymbol{b^TA}$, then we compare Benchmark X for $\boldsymbol{Ab}$ against Benchmark X for $\boldsymbol{b^TA}$ and finally we compare Benchmark X for $\boldsymbol{Ab}$ against Benchmark X for $\boldsymbol{b^TA}$.

- For each input we compare Benchmark X for the pure row-oriented routine against Benchmark X for JAMA's `A.times(B)` routine and Benchmark Y for the pure row-wise routine against Benchmark Y for JAMA's routine and finally Benchmark Z for the pure row-wise routine against Benchmark Z for JAMA's routine.

The idea behind choosing just these comparisons is that for an $\boldsymbol{AB}$ operation we have the possibility to measure the O(m) (Benchmark X)operations, O($m \cdot n$) (Benchmark Y) and finally O($m \cdot n \cdot p$) (Benchmark Z) up against each other. It is now possible to see which constructions that makes the difference (each benchmark are executed the same amount of time). This is not exactly what we do for comparing $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$ operations but we still get the information we need, now we must beware that for some of the benchmarks we compare, the constructions may be executed different amounts of time.

We must also compare the impact of the loop-overhead to see if optimising the for-loops, has any impact for the overall performance for various inputs. When we compare the benchmarks we are looking for which one are most efficient, why it is most efficient and also vaguely estimate how large the difference is.

Next we introduce the notation we use in the following sections:

1. {} means we have one for-loop.
2. {{}} means we have two nested for-loops.
3. {{{}}} means we have three nested for-loops.

Some words on this notation: {1}, {1{n}}, {1{n {p}}}. This notation tells us how many times the for-loops in the benchmarks are executed and initialised for various inputs.

- {1}: Tells us that the first for-loop is initialised *1*-time and executed *1*-time.
- {1{n}}: Tells us that the first for-loop is initialised *1*-time and executed *1*-time, the second for-loop is initialised *1*-time and executed *n*-times.
- {1{n {p}}}: Tells us that the third for-loop is initialised *n*-times and executed *np*-times.

This notation is important because it gives information on how many times we initialise and execute each for-loops and how many times each loop-body is executed. Understanding this notation is crucial for understanding the following comparisons. When we say execution we mean how many times the for-loop iterates before it terminates. When we say initialisation we mean how many times a for-loop gets initialised. This is the cost of starting and terminating a for-loop not the cost of iterating it. Table 4-0 shows the for-loops executions and initialisations for each benchmark for each operation.

Another thing to notice is that in JAMA's `A.times(B)` routine we have a `for`-loop that iterates through the columns of matrix ***B***, this for-loop is considered a part of the loop-body of Benchmark X.

Executions and Initialisation of the for-loops					
	JAMA		ROW		JAMA/ROW
	Ab	***b^TA***	***Ab***	***b^TA***	***AB***
Benchmark X	{1}	{m}	{m}	{1}	{m}
Benchmark Y	{1 {n}}	{m {1}}	{m {n}}	{1 {n}}	{m {n}}
Benchmark Z	{1 {n {p}}}	{m {1{p}}}	{m {n {1}}}	{1 {n {p}}}	{m {n {p}}}

Table 4-0
The for-loops executions and initialisations for each benchmark and operation.

ROW means the pure row-oriented `A.times(B)` routine.
JAMA means JAMA's `A.times(B)` routine.

4.3.4 JAMA's versus the pure row-oriented `A.times(B)` on *AB*.

In this section we compare JAMA's `A.times(B)` routine on input ***AB*** versus the pure row-oriented `A.times(B)` routine on input ***AB***. First we present the benchmark timings for the empty for-loops (loop-overhead) then we present the benchmark timings for the benchmarks presented in section 4.3.2. In Appendix A, we have all the timing results.

Appendix A [Table 1-12] shows the loop-overhead:

- Benchmark X: We measured no difference.
- Benchmark Y: This small difference is caused by the loop-optimisation of the for-loops of the pure row-oriented `A.times(B)` routine on input ***AB***.
- Benchmark Z: This difference is caused by optimised for-loops of the pure row-oriented `A.times(B)` routine on input ***AB***. Here we see that the loop-optimisation is significant.

Appendix A [Table 1-13] shows the benchmark-timings:

- Benchmark X: Here we see a clear difference because the cost of the loop-body of JAMA's `A.times(B)` on input ***AB*** compared to the cost of the loop-body of the pure row-oriented `A.times(B)` routine on input ***AB***.
- Benchmark Y: Here we see a small difference caused by the difference in loop-body cost and optimised for-loops.
- Benchmark Z: Here we see a clear difference because the cost of the loop-body of JAMA's `A.times(B)` on input ***AB*** is slightly more expensive then the cost of the loop-body of the pure row-oriented `A.times(B)` routine on input ***AB***. The optimised for-loops of the pure row-oriented routine on input ***AB*** also adds to the difference.

The for-loops are executed and initialised the same amount of time as for the pure row-oriented `A.times(B)` routine on input ***AB*** and JAMA's `A.times(B)` routine on input ***AB***. The difference is caused mainly by the constructions in the innermost for-loop in both routines. Since these constructions are executed *mnp*-times they will overshadow all the other differences. And since the cost of executing the construction in the innermost for-loop of JAMA's routine on input ***AB*** is more time consuming then the cost of executing the construction in the innermost for-loop of the pure row-oriented routine on input ***AB***, see section 4.4.1.

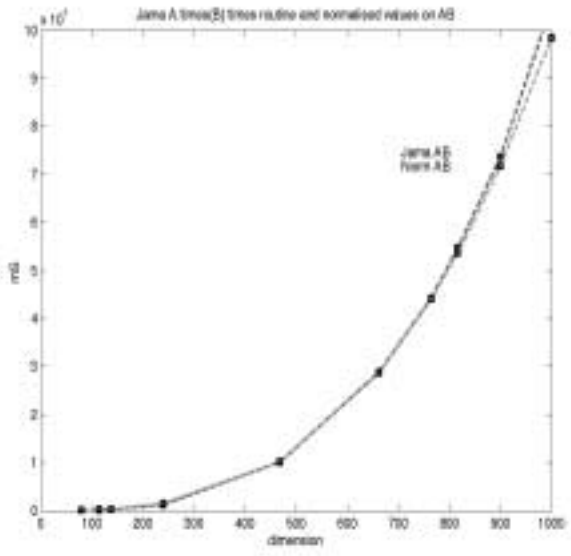

Figure 4-0
The normalised values and JAMA's `A.times(B)` for input *AB*.

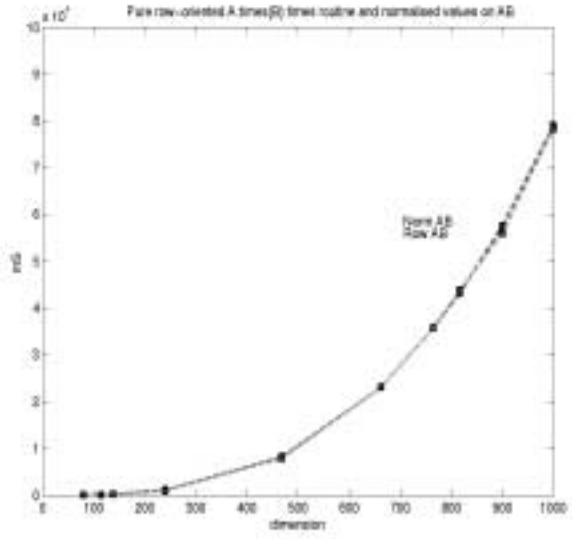

Figure 4-1
The normalised values and the pure row-oriented `A.times(B)` routine on input *AB*.

From Figure 4-0 and Figure 4-1, it is clear that we have consistent results comparing the whole routine with the normalised benchmark results. That means we have similar results for the normalised benchmark values and the values for the whole routine that of course also implies that they have the same behaviour for the same inputs.

When we state that we have consistent results between the normalised values and the values for the whole (JAMA and the pure row-oriented) routine we mean that they are approximately equal and have the same behaviour. This is also true for all such comparisons in this chapter.

4.3.5 The pure row-oriented `A.times(B)` on Ab versus b^TA

In this section we compare the pure row-oriented `A.times(B)` routine on input $\boldsymbol{Ab}$ versus $\boldsymbol{b^TA}$. First we present the benchmark timings for the empty for-loops (loop-overhead) then we present the benchmark timings for the benchmarks presented in section 4.3.2. In Appendix A, we have all the timing results.

Appendix A [Table 1-2] shows the loop-overhead:

- Benchmark X: We measured no loop-overhead.
- Benchmark Y: Here we see a clear difference between $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$. That is because the first and second for-loop of $\boldsymbol{Ab}$ are executed $(m + mn)$-times compared to $(1+n)$-times for $\boldsymbol{b^TA}$.
- Benchmark Z: Here we see a significant difference between $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$. This is because the for-loops are executed $(m + mn + mn)$-times compared to $(1+n+np)$-times for $\boldsymbol{b^TA}$.

Appendix A [Table 1-3] shows the benchmark-timings:

- Benchmark X: We measured no difference.
- Benchmark Y: Here we see a significant difference between $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$. That is because the loop-body of $\boldsymbol{Ab}$ is being executed mn-times compared to n-times for $\boldsymbol{b^TA}$.

- Benchmark Z: Here we see a clear difference between $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$. Since the innermost loop-body is executed the same number of times, we must see how many times the for-loops are being initialised. The for-loops of $\boldsymbol{Ab}$ are initialised $(1+m+mn)$-times for $\boldsymbol{Ab}$ compared to $(1+1+n)$-times for $\boldsymbol{b^TA}$, these loop overheads causes the difference.

Since both routines are pure row-oriented, we would not expect such a significant difference, but because the second and third for-loop is executed mn-times for $\boldsymbol{Ab}$ compared to $(n+np)$-times for $\boldsymbol{b^TA}$, we have a clear difference.

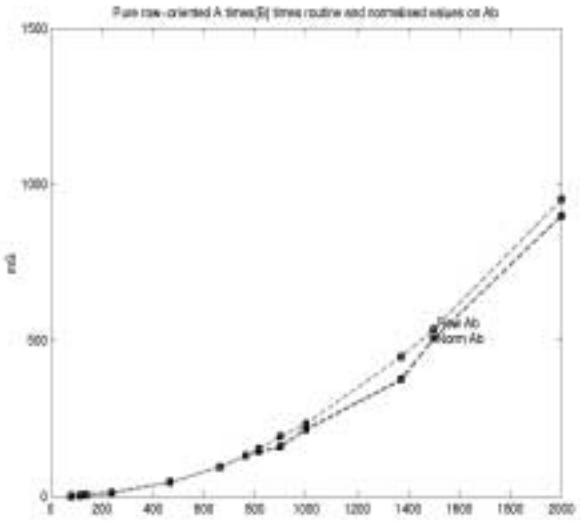

Figure 4-2
The normalised values and the pure row-oriented `A.times(B)` for Ab.

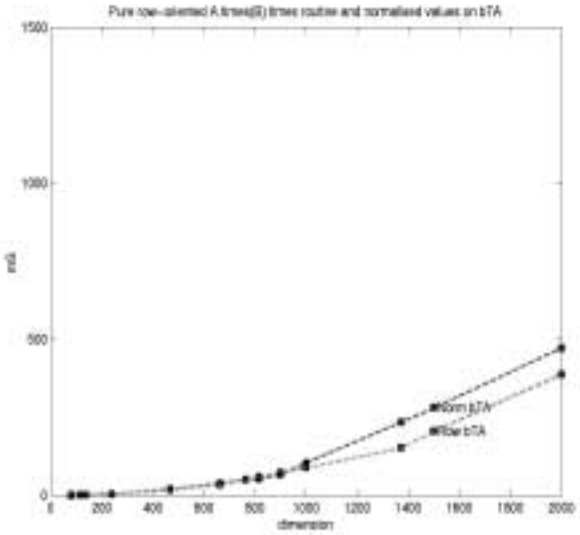

Figure 4-3
The normalised value and the pure row-oriented `A.times(B)` for b^TA.

From Figure 4-2 and Figure 4-3 it is clear that we have consistent results comparing the whole routine with the normalised benchmark results. Even if we have a 'clitch' for matrix dimension larger then 800, the difference is constant and the 'clitch' is not getting larger for increasing

matrix dimensions. So even if we do not have the similar results, we see that we have a constant difference which also implies that we have the same behaviour for the same inputs.

4.3.6 JAMA's `A.times(B)` on Ab versus b^TA

In this section we compare JAMA's `A.times(B)` on input $\boldsymbol{Ab}$ versus $\boldsymbol{b^TA}$. First we present the benchmark timings for the empty for-loops (loop-overhead) then we present the benchmark timings for the benchmarks presented in section 4.3.2. In Appendix A, we have all the timing results.

Appendix A [Table 1-0] shows the loop-overhead:

- Benchmark X: We measured no loop-overhead.
- Benchmark Y: We measured no loop-overhead
- Benchmark Z: Here we measured a loop-overhead but there was not any significant difference.

Appendix A [Table 1-1] shows the benchmarks timings:

- Benchmark X: Here we see a significant difference between $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$. The for-loop for $\boldsymbol{b^TA}$ is executed m-times and has a very time consuming loop-body compared to the $\boldsymbol{Ab}$ operation that is only executed 1-time.
- Benchmark Y: Here we see a small difference between $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$, this is because the second for-loop in $\boldsymbol{b^TA}$ is initialised m-times compared to 1-time for $\boldsymbol{Ab}$ both loop-bodies are executed mn-times.
- Benchmark Z: Here there is no significant difference between $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$, both loop-bodies are being executed same amount of time, remember that $m=n=p$.

The Benchmark X for JAMA's `A.times(B)` on $\boldsymbol{b^TA}$ is a very expensive operation. In fact it is that operation which gets a column from `double[][]` to a `double[]`, see Chapter 3, section 3.7. This operation is the most significant time component when comparing these two benchmarks.

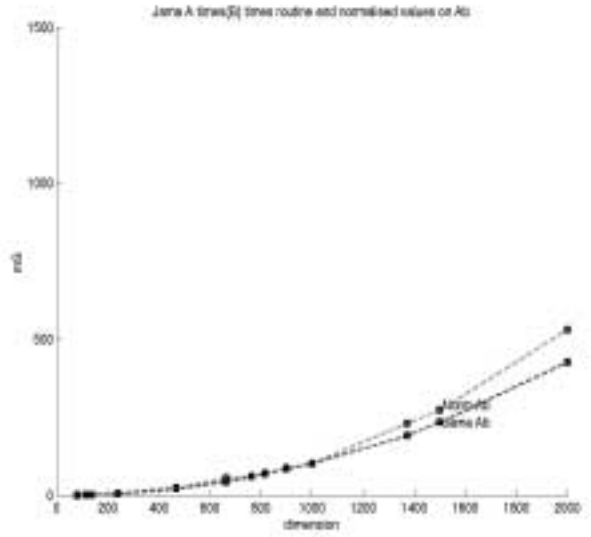

Figure 4-4
The normalised values and JAMA's `A.times(B)` values for input *Ab*.

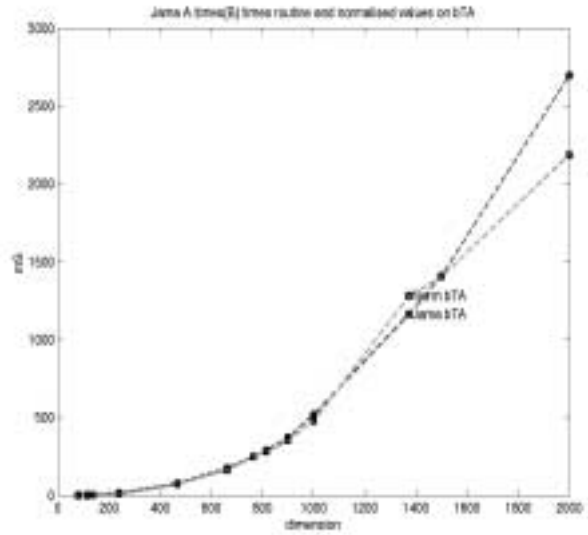

Figure 4-5
The normalised values and JAMA's `A.times(B)` values for input b^TA.

From Figure 4-4 and 4-5 it is clear that we have consistent results comparing the whole routine with the normalised benchmark results for matrix dimension less then 1500. For larger matrix dimension we have a 'clitch' that is getting wider between the values for the whole routine and the normalised values. The reason for that maybe uncertainty in the benchmarks timings, we have for JAMA's `A.times(B)` on input $\boldsymbol{b^TA}$ the loop-body of the first for-loop a construction that is also a for-loop. This latter for-loop gets the columns from `double[][]` to a `double[]` see Chapter 3, section 3.7. Next we will explain this in more detail.

The reason that we have more accurate normalised set of values for the $\boldsymbol{AB}$ operation then for the $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$ operations, is that Benchmark Z is an O(n^3) operation and the O(n^2) and O(n) operations add little to the overall performance. The O(n^3) operation is the dominating time factor, see Table 1-13 [Appendix A]. The fact that Benchmark Z is very similar in construction to the whole routine (three nested for-loops), makes us expect that it is less sensitive to compiler

effects, like dead code elimination. The ***Ab*** and $\boldsymbol{b^TA}$ operations have O(n^2) operations for Benchmark Z, but for benchmark Y for the pure row-oriented `A.times(B)` routine on input ***Ab*** we also have an O(n^2) operation. Both these benchmarks add significantly to the overall performance, that is all the constructions that are executed O(n^2) times. But Benchmark Y and Benchmark Z could be handled very differently in the compiler and JVM, because Benchmark Y is a 'simpler' construction then Benchmark Z, and is more sensitive for an optimising compiler. We have a nearly similar scenario with the $\boldsymbol{b^TA}$ operation for JAMA's `A.times(B)` routine, where the loop-body to the first for-loop is a for-loop that is executed O(n^2)-times. This will also be the case for all the O(n) operations for the ***Ab*** and $\boldsymbol{b^TA}$ operations. We will conclude that ***Ab*** and $\boldsymbol{b^TA}$ are more sensitive to compiler effects then ***AB***. A consequence of is that we get more uncertainty in the normalised values and it may explain that we "only" have consistent results for the *AB* operations.

4.3.7 JAMA's `A.times(B)` versus the pure row-oriented `A.times(B)`

When comparing JAMA's `A.times(B)` with the pure row-oriented `A.times(B)`, it is not as straightforward as it was with JAMA's routine on ***Ab*** versus $\boldsymbol{b^TA}$ and the pure row-oriented on ***Ab*** versus $\boldsymbol{b^TA}$. The reason for this is the difference in cost of the loop-bodies. There is not only a difference in the loop-body cost, but also the cost of the for-loops. The for-loops of the pure row-oriented routine are optimised, see Chapter 2, section 2.9. Even if there is a difference in the constructions, analysing and finding those time components that make the difference is still possible.

4.3.8 The pure row-oriented versus JAMA's `A.times(B)` on *Ab*.

In this section we compare JAMA's `A.times(B)` on input ***Ab*** versus the pure row-oriented `A.times(B)` on input ***Ab***. First we present the benchmark timings for the empty for-loops (loop-overhead) then we present the benchmark timings for the benchmarks presented in section 4.3.2. In Appendix A, we have all the timing results.

Appendix A [Table 1-4] shows the loop-overhead:

- Benchmark X: We measured no loop-overhead.
- Benchmark Y: Here we see a clear difference between JAMA's `A.times(B)` on input ***Ab*** and the pure row-oriented `A.times(B)` on input ***Ab***. This is because the first and second for-loop of the pure row-oriented routine on ***Ab*** is being executed ($m+mn$)-times compared to ($1+n$)-times for JAMA's routine on ***Ab***.
- Benchmark Z: Here we see a significant difference between JAMA's `A.times(B)` on ***Ab*** and the pure row-oriented on ***Ab***. That because the first, second and third for-loop is being executed ($m+mn+mn$)-times for the pure row-oriented `A.times(B)` on ***Ab*** compared to ($1+n+np$)-times for JAMA's routine on ***Ab***.

Appendix A [Table 1-5] shows the benchmarks timings:

- Benchmark X: We measured no loop-overhead.
- Benchmark Y: Here we see a significant difference between JAMA's `A.times(B)` routine on input ***Ab*** and the pure row-oriented `A.times(B)` routine on input ***Ab***. This because the loop-body is being executed *mn*-times for the pure row-oriented routine on input ***Ab*** compared to *n*-times for JAMA's routine on input ***Ab.***
- Benchmark Z: Here we see a clear difference between JAMA's `A.times(B)` routine on input ***Ab*** and the pure row-oriented `A.times(B)` routine on input ***Ab***. Both are being executed the same amount of time, but because the loop-overhead is significant for the pure row-oriented routine on input ***Ab,*** it will cause JAMA's routine on input ***Ab*** to be more efficient.

The pure row-oriented `A.times(B)` routine is less efficient then JAMA's `A.times(B)` routine on ***Ab***. This is because we initialise and execute the for-loops more times for the pure row-oriented routine on input ***Ab*** then we did for the JAMA's routine on input ***Ab***.

4.3.9 The pure row-oriented on *Ab* versus JAMA's `A.times(B)` on b^TA.

In this section we compare JAMA's `A.times(B)` on input $\boldsymbol{b^TA}$ versus the pure row-oriented `A.times(B)` on input ***Ab***. First we present the benchmark timings for the empty for-loops (loop-overhead) then we present the benchmark timings for the benchmarks presented in section 4.3.2. In Appendix A, we have all the timing results.

Appendix A [Table 1-6] shows the loop-overhead:

- Benchmark X: We measured no difference.
- Benchmark Y: We measured no difference.
- Benchmark Z: Here we see a clear difference between JAMA's `A.times(B)` routine on input $\boldsymbol{b^TA}$ versus the pure row-oriented `A.times(B)` on input ***Ab***. This because the first, second and third for-loop is being executed (*m*+*mn*+*mn*)-times for the pure row-oriented routine on input ***Ab*** compared to (*m*+*m*+*mp*)-times for JAMA's routine on input $\boldsymbol{b^TA}$.

Appendix A [Table 1-7] shows the benchmark-timings:

- Benchmark X: Here we see a significant difference between JAMA's `A.times(B)` routine on input $\boldsymbol{b^TA}$ and the pure row-oriented `A.times(B)` on input ***Ab***. That because the difference in executing the loop-body for JAMA's routine on input $\boldsymbol{b^TA}$, which is significantly more expensive then the loop-body for the pure row-oriented routine on input ***Ab***.
- Benchmark Y: Here we see a clear difference between JAMA's `A.times(B)` routine on input $\boldsymbol{b^TA}$ and row ***Ab***. That because the loop-body of the pure row-oriented `A.times(B)` on input ***Ab*** is executed *mn*-times compared to *m*-times for JAMA's routine on input $\boldsymbol{b^TA}$.
- Benchmark Z: Here we see a clear difference between JAMA's `A.times(B)` routine on input $\boldsymbol{b^TA}$ and the pure row-oriented `A.times(B)` on ***Ab***. This because we initialise the for-loops of the pure row-oriented routine on input ***Ab*** (*1*+*mn*+*mn*)-times compared to (*1*+*m*+*m*)-times for JAMA's routine on input $\boldsymbol{b^TA}$.

The difference of Benchmark X is more significant then the differences between Benchmark Y and Benchmark Z the other benchmarks. The cost of executing the loop-body of Benchmark X of JAMA's `A.times(B)` on input $\boldsymbol{b}^T\boldsymbol{A}$, *m*-times, makes it the significant time component.

4.3.10 The pure row-oriented on $\boldsymbol{b}^T\boldsymbol{A}$ versus JAMA's `A.times(B)` on $\boldsymbol{Ab}$.

In this section we compare JAMA's `A.times(B)` on input $\boldsymbol{Ab}$ versus the pure row-oriented `A.times(B)` on input $\boldsymbol{b}^T\boldsymbol{A}$. First we present the benchmark timings for the empty for-loops (loop-overhead) then we present the benchmark timings for the benchmarks presented in section 4.3.2. In Appendix A, we have all the timing results.

Appendix A [Table 1-8] shows the loop-overhead:

- Benchmark X: We measured no difference.
- Benchmark Y: We measured no difference.
- Benchmark Z: Here we see a small difference, because we have highly optimised for-loops in the pure row-oriented `A.times(B)` routine compared to the for-loops of JAMA's `A.times(B)` routine. Since the for-loops are being executed the same amount of time this optimisation makes the difference.

Appendix A [Table 1-9] shows the benchmark-timings:

- Benchmark X: We measured no difference.
- Benchmark Y: We measured no difference.
- Benchmark Z: Here we see a clear difference between JAMA's `A.times(B)` on input $\boldsymbol{Ab}$ and the pure row-oriented `A.times(B)` $\boldsymbol{b}^T\boldsymbol{A}$.

The reason for this is two factors:

1. The loop-overhead for JAMA's routine on input $\boldsymbol{Ab}$ is more expensive then the loop-overhead for the pure row-oriented routine on $\boldsymbol{b}^T\boldsymbol{A}$.
2. The loop-body of JAMA's routine on input $\boldsymbol{Ab}$ is more time consuming then the loop-body of the pure row-oriented routine on $\boldsymbol{b}^T\boldsymbol{A}$, see section 4.4.1.

The difference here is not so obvious, all the for-loops are executed and initialised the same amount of time, so we have to focus on the cost of the for-loops and the cost of the loop-bodies. Since the cost of the pure row-oriented routine constructions (for-loops and loop-bodies) is less time consuming then JAMA's routine it will cause the difference.

4.3.11 The pure row-oriented versus JAMA's `A.times(B)` on $\boldsymbol{b}^T\boldsymbol{A}$

In this section we compare JAMA's `A.times(B)` on input $\boldsymbol{b}^T\boldsymbol{A}$ versus the pure row-oriented `A.times(B)` on input $\boldsymbol{b}^T\boldsymbol{A}$. First we present the benchmark timings for the empty for-loops (loop-overhead) then we present the benchmark timings for the benchmarks presented in section 4.3.2. In Appendix A, we have all the timing results.

Appendix A [Table 1-10] shows the loop-overhead:

- Benchmark X: We measured no difference.
- Benchmark Y: We measured no difference.
- Benchmark Z: Here we see a small difference which is caused by the optimised for-loops of the pure row-oriented `A.times(B)` routine on input $\boldsymbol{b^T A}$ and that the for-loops of JAMA's `A.times(B)` on input $\boldsymbol{b^T A}$ is initialised ($1+m+m$)-times compared to ($1+1+n$)-times for the pure row-oriented routine on input $\boldsymbol{b^T A}$.

Appendix A [Table 1-11] shows the benchmark-timings:

- Benchmark X: Here we can see a significant difference between JAMA's `A.times(B)` on input $\boldsymbol{b^T A}$ and the pure row-oriented `A.times(B)` routine on input $\boldsymbol{b^T A}$. This is because the time consuming loop-body of JAMA's routine on $\boldsymbol{b^T A}$ is executed m-times compared to 1-time for the pure row-oriented routine on input $\boldsymbol{b^T A}$.
- Benchmark Y: Here there is a small difference between JAMA's `A.times(B)` on input $\boldsymbol{b^T A}$ and the pure row-oriented `A.times(B)` on $\boldsymbol{b^T A}$. That because the loop-body of JAMA's routine on input $\boldsymbol{b^T A}$ is slightly more expensive then the loop-body of the pure row-orientedroutine on input $\boldsymbol{b^T A}$ and the for-loops of the pure row-oriented routine on input $\boldsymbol{b^T A}$ are optimised.
- Benchmark Z: Here we see a clear difference between JAMA's `A.times(B)` on input $\boldsymbol{b^T A}$ and the pure row-oriented `A.times(B)` routine on input $\boldsymbol{b^T A}$. The loop-body of JAMA's routine on input $\boldsymbol{b^T A}$ is slightly more expensive then the loop-body of the pure row-oriented routine on input $\boldsymbol{b^T A}$ and because we have optimised the for-loops of the pure row-oriented routine on input $\boldsymbol{b^T A}$.

The significant time component is the time consuming loop-body of JAMA's `A.times(B)` on input $\boldsymbol{b^T A}$, that is Benchmark X.

4.4 Testing on Different Platforms

In this section we will run the same comparisons on other platforms then the testplatforms, the reason for that is to see if we also tested the following inputs on JAMA's `A.times(B)` routine and the pure row-oriented `A.times(B)` routine on other platforms. We tested the routines on the following inputs $\boldsymbol{Ab}$, $\boldsymbol{b^T A}$ and $\boldsymbol{AB}$.

The platforms are:

1. Sun Solaris UltraSparc JDK 1.3 (Sun)
2. Dell Inspiron 8000 Windows 2000 JDK 1.3 (Sun)
3. PC Windows NT JDK 1.2 (Sun)

Dell Inspiron 8000 Windows 2000 JDK 1.3 (Sun):

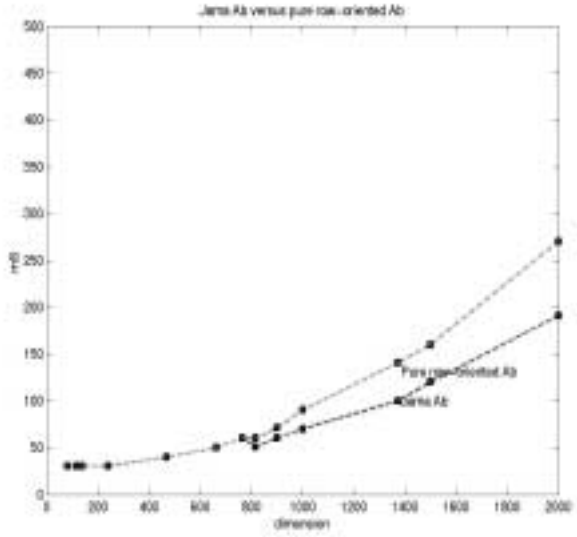

Figure 4-6
JAMA's and the pure row-oriented `A.times(B)` routine on input *Ab*.

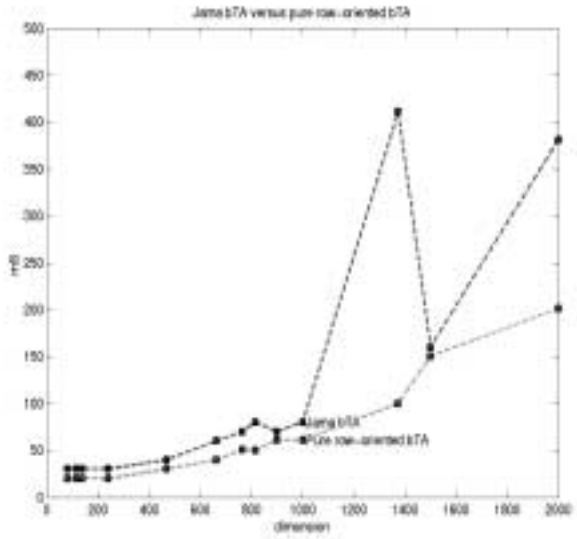

Figure 4-7
JAMA's and the pure row-oriented `A.times(B)` routine on input b^TA.

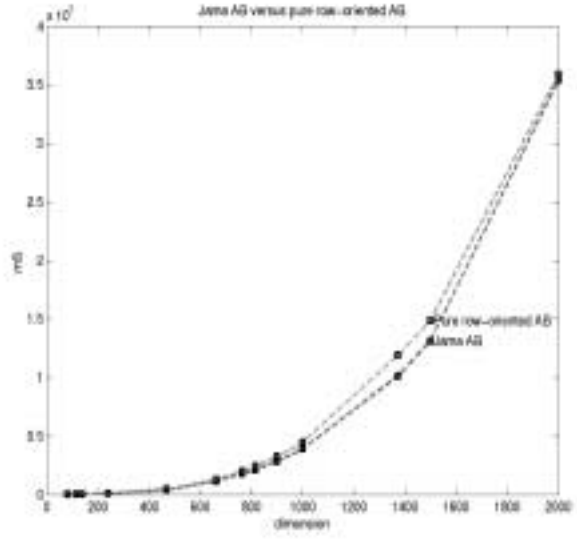

Figure 4-8
JAMA's and the pure row-oriented `A.times(B)` routine on input *AB*.

Figure 4-6 shows that there still is a difference between JAMA's `A.times(B)` and the pure row-oriented `A.times(B)` on input ***Ab.*** Figure 4-7 shows that there still is a difference between JAMA's routine and the pure row-oriented routine on input $\boldsymbol{b^T A}$**.** These results are consistent with the test platform. Figure 4-8 shows that JAMA's routine and the pure row-oriented routine on input ***AB*** and here we have only a slight difference in performance, that is here the pure row-oriented routine on ***AB*** is less efficient then JAMA's routine on the same input. We have here the reversed result from the testplatform.

PC Windows NT JDK 1.2 (Sun):

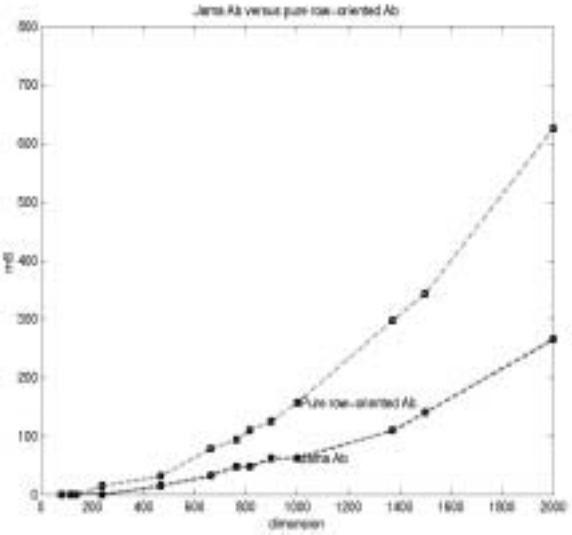

Figure 4-9
JAMA's and the pure row-oriented `A.times(B)` routine on input *Ab*.

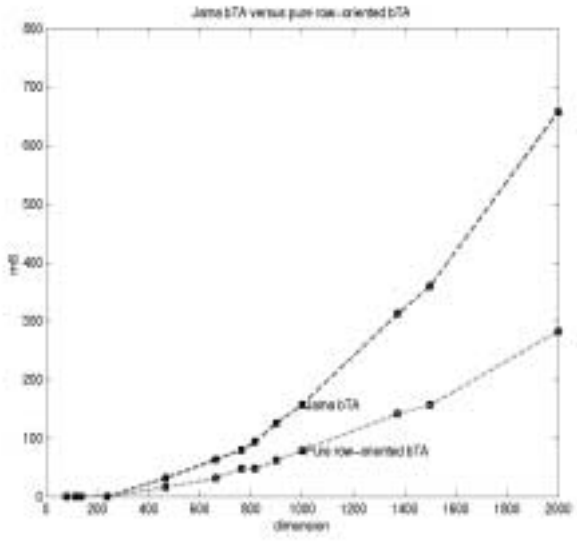

Figure 4-10
JAMA's and the pure row-oriented `A.times(B)` routine on input b^TA..

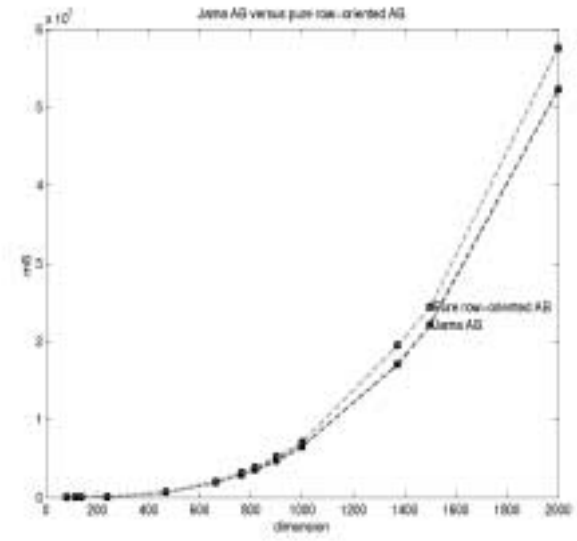

Figure 4-11
JAMA's and the pure row-oriented `A.times(B)` routine on input *AB*.

Figure 4-9 shows that there still is a difference between JAMA's `A.times(B)` and the pure row-oriented `A.times(B)` on input ***Ab.*** Figure 4-10 shows that there is still a difference between JAMA's routine and the pure row-oriented routine on input $\boldsymbol{b^TA}$**.** These results are consistent with the test platform. Figure 4-11 shows that JAMA's routine and the pure row-oriented routine on input ***AB*** and

These two platforms had the same consistent results as the test platform for the ***Ab*** and $\boldsymbol{b^TA}$ inputs, but for the ***AB*** input the results were reversed compared to the testplatform.

In the next section we shall see that going from JDK (Java Development Kit) to another on the same platform causes an inconsistency on input ***AB***. We will also indicate a reason for this, that is the cost of executing constructions will vary from one JDK to another.

4.4.1 *AB* on JDK 1.2 versus *AB* on JDK 1.3

In this section we compare JAMA's `A.times(B)` on input ***AB*** versus the pure row-oriented `A.times(B)`on input ***AB***. First we present the benchmark timings for the empty for-loops (loop-overhead) then we present the benchmark timings for the benchmarks presented in section 4.3.2. In Appendix A, we have all the timing results.

Appendix A [Table 1-14] shows the loop-overhead:

- Benchmark X: We measured no differences.
- Benchmark Y: This difference is caused by the optimisation of the pure row-oriented `A.times(B)` on input ***AB*** for-loops.
- Benchmark Z: This difference is caused by the optimisation of the for-loops of the pure row-oriented `A.times(B)` on input ***AB***.

Appendix A [Table 1-15] shows the benchmark-timings:

- Benchmark X: Here there is a clear difference. The loop-body of JAMA's `A.times(B)` routine on input ***AB*** is more expensive then the loop-body of the pure row-oriented `A.times(B)` on input ***AB***.
- Benchmark Y: Here there is a clear difference. The loop-body of JAMA's `A.times(B)` routine on input ***AB*** is more expensive then the loop-body of the pure row-oriented `A.times(B)` on input ***AB***
- Benchmark Z: Here there is a clear difference. The loop-body of the pure row-oriented `A.times(B)` on input ***AB*** is more expensive then the loop-body of JAMA's `A.times(B)` routine on input ***AB,*** which is the opposite from JDK 1.2

We went from JDK 1.2 to JDK 1.3 and got the reversed result. The reason is the cost of executing the innermost for-loop construction for JAMA's `A.times(B)` routine on input ***AB*** is much less then for the pure row-oriented `A.times(B)` routine on ***AB***.

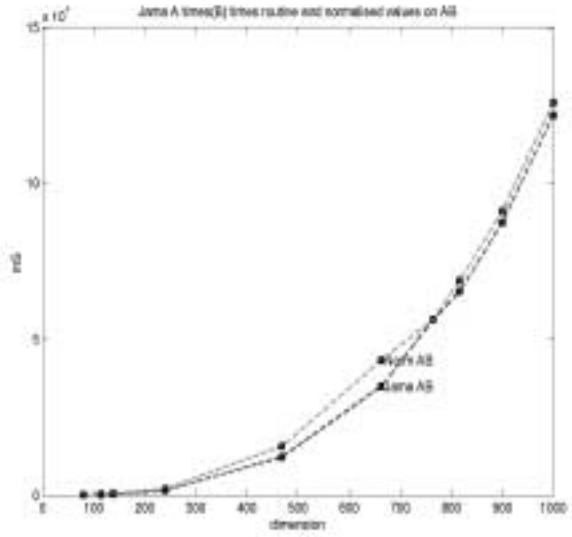

Figure 4-12
The normalised values and JAMA's `A.times(B)` for input *AB*.

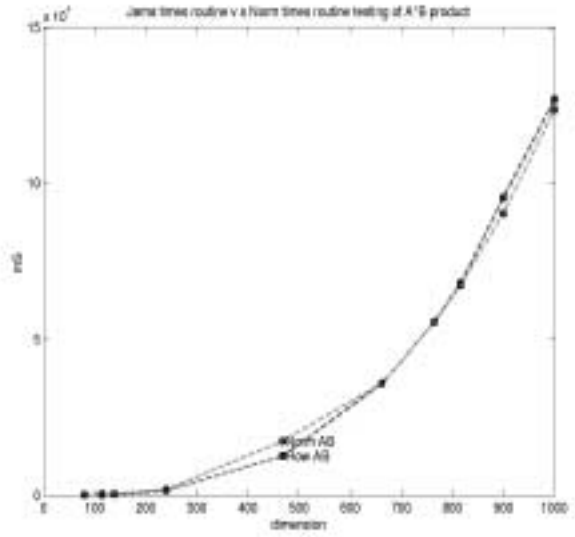

Figure 4-13
The normalised values and the pure row-oriented `A.times(B)` on input *AB*.

From Figure 4-12 and Figure 4-13 it is clear that we have consistent results comparing the whole routine with the normalised benchmark results. By this we mean we have approximately the similar results for the normalised benchmarks timings and the values for the whole routine, that of course also implies that they have the same behaviour for the same inputs.

There are two important issues when measuring these inputs on JAMA's `A.times(B)` routine and the pure row-oriented `A.times(B)` routine. That is that we have the same consistent result concerning the inputs ***Ab*** and $\boldsymbol{b^TA}$**.** Those results were consistent for all the platforms we tested. On input ***AB*** there were a clear difference between JAMA's routine and the pure row-oriented routine for the test platform. JAMA's routine was less efficient then the pure row-oriented routine. For the platforms in this section we have the opposite result. This is a very interesting result since we have the same routines for all platforms. We cannot replace JAMA's routine with the pure row-oriented routine and claim that the latter one has the best overall performance on all platforms. We can probably claim this for the test platform, but not for all platforms.

Next we will indicate the cost of executing the innermost loop-body on *Solaris Ultrasparc* running JDK 1.2, is different then the same platform running JDK 1.3. On JDK 1.3 the innermost loop-body in JAMA's `A.times(B)` routine is less costly then the innermost loop-body of the pure row-oriented `A.times(B)` on JDK 1.3. The opposite is true for JDK 1.2. We will not analyse the same on other platforms, but just state that construction differences are significant from one platform to another. Optimising for one platform may give different results on other platforms**.** The difference between JDK 1.2 and JDK 1.3 for executing the innermost loop-body for the `A.times(B)` routines is shown in Table 4-1.

The innermost construction of JAMA's A.times(B) routine:

- `s+=Arowi[k]*Bcolj[k];`

The innermost construction of the pure row-oriented A.times(B) routine:

- `Crowi[k] += a*Browi[k];`

Innermost construction cost				
Matrix Dimension	Row JDK 1.2	Row JDK 1.3	JAMA JDK 1.2	JAMA JDK 1.3
80	20	51	28	46
115	59	131	79	100
138	103	210	137	163
240	525	1026	713	791
468	3942	10422	5297	7076
663	11099	21110	15063	19712
765	17133	32597	23231	24779
817	20778	39580	27414	36634
900	27722	53134	37719	40370
1000	38034	72648	51904	55891

Table 4-1
The time differences between JDK 1.2 and JDK 1.3 for the time of executing the innermost loop-body for the `A.times(B)` routines, time is in milliseconds (mS).

These results should give us the time of executing the innermost loop-body for both routines on *Solaris Ultrasparc* for both Sun JDK 1.2 and Sun JDK 1.3. For information on the test platform see Chapter 3, section 3.5. Remark, Table 4-1 does not show the relative cost of the constructions, the innermost construction cost we present here we found by subtracting the loop-overhead (empty for-loops) for Benchmark Z from the benchmark timings for Benchmark Z.

As we can see clearly from these timings, the cost of executing the innermost loop-body of JAMA's `A.times(B)` is less costly then the pure row-oriented `A.times(B)` on JDK 1.3. We have the reversed result on JDK 1.2. This difference is so significant that it explains the inconsistency of the whole routines going from JDK 1.2 to JDK 1.3 on the same platform. We will not explain why the pure row-oriented routine on input ***AB*** is less efficient then JAMA's routine on input ***AB*** for the two other platforms. The result we showed above may give us an indication that the same could be the case on these platforms, but we do not claim that it is so. There is an inconsistency in execution time between the innermost loop-body on different JDK's on the same platform and the obvious consequence for it when designing high performance routines is that the performance of a routine is platform dependent. Designing the "best" `A.times(B)` routine using Java arrays, may not be achievable because the "best" implies it must perform better then differently implemented `A.times(B)` routines on all platforms.

4.5 Summary, Discussions and Conclusions

Since this chapter is built on the results of Chapter 3, we also refer to those results in this section.

When implementing a matrix multiplication in any language its performance is very dependent on the underlying architecture of that language. In Java it is likely that we will implement a matrix multiplication using Java's native arrays. That is because they are more efficient when it comes to updating and accessing, then other structures (`Vector`, `ArrayList` and `LinkedList`), see Chapter 5.

The problem with Java arrays, if we look at it from a high performance viewpoint, is its row-wise structure. Declaring and initialising a two-dimensional array we get an array of references to row arrays, where the row arrays are independent objects. When we traverse the two-dimensional array row-wise it is more efficient then traversing it column-wise, see Chapter 3 Figure 3-0. This result did also appear when we implemented the matrix multiplication routine using Java arrays. We must utilise the construction of Java arrays for designing high performance routines**.** The idea was that if we implemented a routine pure row-wise we would achieve a routine with the overall best performance.

We implemented a straightforward matrix multiplication and by interchanging the three for-loops we get six distinct ways of doing a matrix multiplication. We tested all of these six ways on input $\boldsymbol{AB}$. Based on those results we eliminated the pure column-wise implementations. The pure row-oriented routines were most efficient on those routines followed by the partial row-wise and partial column-wise implementations. This gave us a strong indication that implementing a pure row-wise matrix multiplication would give us the most efficient matrix multiplication.

When we analysed JAMA's `A.times(B)` routine we noticed that it had a rather complex implementation compared to the straightforward implementation. We assumed it was because of performance. Breaking encapsulation, eliminating method calls, removing unnecessary declarations, initialisations and traversing `double[]` instead of `double[][]` is usual techniques for optimising code. All of these things were done in JAMA's routine. We further compared JAMA's routine against the straightforward matrix multiplication with the loop-order (i,k,j). The result of those timings was that JAMA's routine was in fact more efficient on the operations $\boldsymbol{Ab}$ and $\boldsymbol{AB}$. The straightforward implementation was more efficient on $\boldsymbol{b^TA}$.

Based on the optimisation techniques used in JAMA's `A.times(B)` and some additional optimising techniques we decided to do the same for a pure row-oriented `A.times(B)` routine. Since we have two ways of doing a pure row-wise matrix multiplication, we implemented both. The (i,k,j) routine was more efficient then the (k,i,j) routine on the inputs $\boldsymbol{Ab}$, $\boldsymbol{b^TA}$ and $\boldsymbol{AB}$. The next comparisons was JAMA's routine against the pure row-oriented routine. We tested them on the $\boldsymbol{Ab}$, $\boldsymbol{b^TA}$ and $\boldsymbol{AB}$ operations.

Since JAMA's `A.times(B)` routine traverses the columns of the ***B*** and ***C*** matrix and the rows of the matrix ***A,*** we consider it to be partial row-wise and partial column-wise. Since we had a pure row-wise `A.times(B)` routine and used many of the optimising techniques and some additional techniques, we assumed that we would have a more efficient routine then JAMA's routine. When we tested this assumption on ***Ab***, $\boldsymbol{b^TA}$ and ***AB*** on the test platform, we saw that the pure row-oriented routine was more efficient on $\boldsymbol{b^TA}$ and ***AB***, but JAMA's routine was still more efficient on ***Ab***.

We could not conclude that the pure row-oriented `A.times(B)` routine had an overall better performance then JAMA's `A.times(B)`even if we had an early break even result between JAMA's routine then the pure row-oriented `A.times(B)`, this because JAMA's routine had a slightly better performance on the ***AB*** operation on several other platforms then the test platform and it also had a more efficient ***Ab*** routine.

One of the reasons for choosing to analyse the JAMA package, was because it is intended from the designers view to be *the* `Matrix` class in Java for numerical linear algebra routines. Looking at some of the methods implemented in that class we can see that it is not highly optimised as the `times()` method. The designers of JAMA have stated that they have made a compromise between the design and performance, so that may be the reason for not optimising all the routines. Optimising leads to more complex, harder to understand and harder to maintain code that might be a reason for not optimising all of the routines and it is not certain we get the boost in performance for justifying it.

The model tried to find those bottlenecks that made a difference between supposed equally efficient routines. The model showed those time components that made the significant difference between routines that performs mathematical operation that is mathematical 'neutral'. For example performing an ***Ab*** operation involves the same amount of operations as a $\boldsymbol{b^TA}$ operation if we only consider the arithmetic operations. The arithmetic operations for these operations are executed the same amount of time in the routine, but other constructions like initialisation, increments and comparisons are executed different amount of time on input ***Ab*** and $\boldsymbol{b^TA}$. We have in the model tried to find the impact of these constructions, which ones made the significant difference. We have stressed throughout Chapter 3 and 4 that the row-wise layout has a significant impact on the performance for a matrix multiplication routine. This is showed in the model, especially comparing the JAMA's `A.times(B)` routine on $\boldsymbol{b^TA}$ with JAMA's routine on input ***Ab*** and the pure row-oriented on input $\boldsymbol{b^TA}$. But when we compared the pure row-oriented `A.times(B)` routine on input ***Ab*** and $\boldsymbol{b^TA}$, both considered to be pure row-wise, that is they do not traverse any columns in a consecutive order, we saw that there was a difference between them. The difference was not so significant as JAMA's routine on ***Ab*** and $\boldsymbol{b^TA}$, but it is clear that even if we eliminate all column traversing in Java arrays we still can get an inconsistency performance between routines that have the same number of arithmetic operations and the same complexity. So if we eliminate column traversing and that introduces more time expensive constructions, see section 4.4.1, and more iteration, see section 4.3.5, we might not achieve the performance we expect.

The weakness with this model was that it only gave the qualitative answers and not the quantitative answers. The reason for this we vaguely blamed on optimising compilers. Since this thesis was not really about going under the hood of Java's Virtual Machine or knowing what an optimising compiler can do, but rather look at it from a programmers viewpoint. We would see how much we could do with utilising the structure of Java arrays and some basic

optimisation techniques. But the issue concerning compilers and JVMs are very interesting for designing high performance routines.

Another important part of the implemented model is the use of for-loops as part of the constructions we choose to be the benchmark. For the first and second model we tried to measure the constructions independently of the running indexes. That means we would not have had any possibility to unveil why there was a difference in traversing a Java array row-wise, compared to column-wise, as with the Frobenius norm example.

We tested most of the routines on the inputs $\boldsymbol{Ab}$, $\boldsymbol{b^TA}$ and $\boldsymbol{AB}$. Most of the papers we read while working with this thesis usually tested the performance of a matrix where both matrices where square and had the same dimensions. Following that procedure, one might say that operation might be enough to give a general conclusion on the performance. When performing operations like $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$**,** is that it is a kind of performance test. They are supposed to be equal in performance, if they are not and if the difference is significant like we had for JAMA's routine we can conclude that it is a badly implemented routine. If testing the matrix multiplication routine only on $\boldsymbol{Ab}$, $\boldsymbol{b^TA}$ and $\boldsymbol{AB}$**,** it may still leave us some unanswered question, they will still give enough information to compare different implementations.

The clearest conclusion, is that implementing a high performance numerical package using Java arrays as two-dimensional arrays without considering its row-wise layout, would be impossible**.** We have showed through examples of the Frobenius norm and several matrix multiplication routines, that implementing routines that traverses the rows of Java arrays is to an extent more efficient then traversing the columns. Even if this was not true in some cases, we can conclude that it is preferable to consider a row-wise implementation if we want the routine to achieve the optimal result.

If implementing a routine pure row-wise introduces expensive constructions it may not be the optimal solution. This was the case when we compared JAMA's `A.times(B)` with the pure row-oriented `A.times(B)` routine on $\boldsymbol{AB}$ for JDK 1.3.1 on Solaris UltraSparc. The cost of the innermost for-loop was larger in the pure row-oriented routine compared to JAMA's routine. This difference was not significant but still there was a difference that caused JAMA's routine to be more efficient then the pure row-oriented routine routine on $\boldsymbol{AB}$.

We also showed some optimising techniques that enhanced the performance of the Java routine in general (optimising for-loops and eliminating unnecessarily declarations, see Chapter 3 section 3.7 and 3.8). These techniques did not have a significant impact when we compared the routines compared to eliminating column traversing, this was clearly showed when we used the model for explaining the differences.

When we compared the routines on other platforms (hardware and software), we achieved consistent results for JAMA's `A.times(B)` on $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$, but for some platforms the difference was much smaller then for the test platform. We had inconsistent results concerning the $\boldsymbol{AB}$ operation as already mentioned. Therefore we can conclude that the performance of a routine is platform dependent. Trying to create the "best" routine when it comes to performance, might not be of use, but rather try finding the best routine for the platform(s) it is likely to be used on.

If the difference between traversing the rows compared to the columns were not so significant, this chapter would have been unnecessarily. One proposal how we could make the difference

less significant, is to cluster the row objects together in memory [56]. If this method works satisfactorily we do not need to take the row-wise layout into consideration when implementing high performance numerical routines. We could also use the elegant notion of `double[][]` when traversing a routine, but that can only be achieved if the cost of accessing a `double[][]` is the same as accessing `double[]` no matter if we traverse it row-wise or column-wise. Another proposal that together with clustering array objects, is to eliminate array bounds check [80].

If Java arrays does not grand the implementers of high performance numerical routines, or any other high performance routines, not enough performance. Other constructions must be considered. One suggestion is to make a multidimensional Java array class [62].
After having tested the routines on several platforms we can conclude that Java is not performance independent. For Java to become a realistic choice it must also have performance portability.

Since there are so many drawbacks using Java as a numerical programming language, one might wonder why there is any interest in it and why do so many publish papers on the subject. The reason might be that people *will* use Java for numerical computations, so instead of encouraging people to use FORTRAN, C and C++ it may be more fruitful to invest time and resources finding how to use Java for numerical computation and how we can improve Java in the future.

We have in Chapter 3 and 4 analysed the use of Java arrays, not only on how to implement an efficient matrix multiplication routine, but also how we could utilise Java arrays and the Java language in general to increase the performance in numerical routines. A similar approach is very important for most of the constructions in the Java language for designing high performance routines and designing numerical libraries in Java, to unveil disadvantages, as the column-wise traversing of a two-dimensional array and advantages as we will analyse in Chapter 5.

Chapter 5 Storage Schemes in Java for Large Sparse Matrices

5.0 Introduction

We have several different storage schemes for large sparse matrices that are being used in languages like FORTRAN, C and C++. The storage schemes considered are compressed row storage (CRS), compressed column storage (CCS) and coordinate storage (COO). These are for general sparse matrices for numerical operations. For non-general matrices we have specialized storage scheme's based on CRS, CCS and COO. An example of this is symmetric matrices we only need to store the lower or upper matrix. Matrices with special structures like symmetry are not considered.

In this chapter we will use Java's native arrays to implement CRS, CCS and COO. We use Java arrays instead of Java's utility classes because of performance, see section 5.7. We will implement some fundamental linear algebra routines on these storage schemes for storing sparse matrices and they will be compared to each other on the basis of performance (speed) and memory use.

In Chapter 3 and 4 we focused on the row-wise layout of two-dimensional Java arrays and what impact it had on implementing algorithms. This is not an issue in this chapter, because we do not declare two-dimensional arrays for implementing CRS, CCS, COO. For these storage schemes we only declare and initialize one-dimensional Java arrays, see section 5.3.

In this chapter we will introduce the Sparse Arrays format using Java's native arrays. This is a new that format has more dynamic features compared to the traditionally storage schemes like CRS, CCS and COO. We will utilize these dynamic features for several matrix operations and compare the Sparse Arrays format to CRS on matrix multiplication on the basis of performance (speed) and memory. We do declare and initialize two-dimensional arrays implementing Sparse Arrays but we do not however traverse any columns for any routine implemented on Sparse Arrays.

Sparse Arrays is a unique construction that cannot be constructed in other languages. We shall see that Sparse Arrays can compete with the more traditionally storage schemes like CRS on numerical operations. Sparse Arrays is also competitive to the more traditionally object-oriented graph implementation in Java. Another feature of Sparse Arrays is that we can do both symbolic operations and numerical operations, no structure transformation is needed. In this chapter we will also consider Sparse Arrays for symbolic operation (Cholesky Factorisation).

A sparse matrix is usually defined as a matrix where ”many” of its elements are equal to zero or a matrix is sparse when we can benefit both in time and space storing, by only working on the nonzero structure [70].

Ideally, we would like to store and operate only on the nonzero entries of the matrix, but such a policy is not necessarily a clear win in either storage or work. The difficulty is that sparse data structures include more overhead (to store indices as well as numerical values of nonzero matrix entries) than the simple arrays used for dense matrices. Arithmetic operations on the stored data, cannot be performed as rapidly (due to indirect addressing of operands). Therefore there is a

trade off in memory requirements between sparse and dense representations and a trade off in performance between the algorithms that uses them. For this reason, a practical requirement for a family of matrices to be `usefully' sparse, is that they have only O(*n*) nonzero entries. Example a (small) constant number of nonzeros per row or column, independent of the matrix dimensions. In addition to the number of nonzeros, their particular locations or pattern in the matrix also has a major effect on how well sparsity can be exploited. Sparsity arising from physical problems usually exhibits some systematic pattern that can be exploited effectively, whereas the same number of nonzeros located randomly might offer relatively little advantage.

All the sparse matrices we use as test matrices in this chapter were taken from Matrix Market [81]. As in Chapter 3 and 4 all the matrices are square, but all of the matrices are classified as general, that is with no properties or structure that we can use special storage schemes to store.

5.1 Preliminaries

In this section we present some fundamental operations and terminology that will be used to perform both a matrix multiplication and a matrix addition where we have the matrices on Sparse Arrays format. Merging a sparse list of integers, see section 5.1.2, and addition of sparse vectors, see section 5.1.3 will also be referred to as the Pissanetsky approach.

5.1.1 Ordered and Unordered Representation

An ordered representation means that the elements of each row and column are sorted by their indexes in an increasing order, while an unordered representation means that the elements of each row and column are not necessarily ordered by their indexes.

None of the sparse algorithms we present here demands that the matrix representation is ordered. It is very costly to keep an ordered representation of a matrix, only a few sparse algorithms require a matrix representation to be ordered [4].

5.1.2 Merging of Sparse List of Integers

Merging a sparse list of integers is fundamental in many sparse algorithms [5] matrix multiplication and matrix addition for Sparse Arrays uses this operation, see section 5.5.3 and 5.5.4 respectively. It is therefore essential for the reader to understand this operation to further understand the implementations of these routines.

We do a merging of a sparse list of integers, between 0 and N, like this:

List A: 1,5,3,7
List B: 3,5,8,9
List C: 7,9,2,8

The result of merging these three lists could be:

The merged List D: 1,5,3,7,8,9,2

After merging List A and List B we only add the element 2 from List C to the merged List D, this because the rest of the elements of List C are already in the result list. Now we have taken a union of the three lists.

Next we take a look on how we do a merging of sparse list of integers. Each new element that we want to merge, must be tested to see if it is already in the result list. This we do by a switch array of the type `boolean`. The switch array is an expanded row, which has the length of a full row. To see if an element already is in the list, we index the switch array with the index of that element and see what the switch array is set to. If the switch array is set to `true` the element is already in the result list, if it is set to `false` we can insert the element into the result list and update a counter. This process we have to do in two stages, implementing in Java with Java arrays. First we have to set all the values in the switch array either to `true` or `false`. A `boolean` array is initialised to `false`. Now we know the size of the compressed resulting sparse vector, so by checking the counter, we can declare that vector.

The next stage is to accumulate the elements from the list to the compressed resulting sparse vector via the switch array. We now have to loop through all the elements in each array over again, but only adding those elements that are initialised to `true` in the switch array, this we only do once. When an element is added to the compressed resulting sparse vector we initialised that element in the switch array to `false`. We now have the resulting index vector.

5.1.3 Addition of Sparse Vectors

The definition of sparse vectors is that we only store the nonzero values and the indexes of the values. We use a `double[]` to store the values and we use `int[]` to store the indexes. What we did in section 5.1.2 was to merge the indexes of sparse vectors, this merging tells us where the indexes of the resulting values of an addition of sparse vectors would be.

After we have created the resulting index vector (the compressed list from section 5.1.2) we perform the addition of sparse vectors. We do this with the help of an expanded accumulator with the size of a full row. The expanded accumulator is an array of type `double[]`, we use this array for temporary storage of values. Let us call the expanded accumulator for X for storing values and the expanded accumulator for IX for index merging.

What we want to achieve is:

$$c = a + b \qquad (5\text{-}1)$$

Instead of accumulating the answer of $a + b$ directly to the c-vector we use the expanded accumulator. First we accumulate the elements of a into X via the indexes of a, then we must accumulate the elements of b into X via the indexes of b. When we do the second accumulation we must do the addition. If the indexes of a and b are similar we do an addition, if they are not similar we insert the element of b directly into X. If we have a third vector we want to add, we perform the same procedure as we did with vector b.

If merging the sparse vectors (indexes are performed as in section 5.1.2) we can use the compressed resulting index vector to get the elements of the expanded accumulator X. We do this by indexing X with IX. We have of course already created the compressed resulting value vector that shall contain the added values.

If the expanded row has so many elements that we do not consider it sparse, see section 5.0, we will get a better performance just by copy from X and IX directly by looping through them. In the routines presented here, we have not taken this scenario into consideration, so if this should happen frequently the routines must be considered inefficient.

5.2 Traditional General Storage Schemes for Large Sparse Matrices

In this section we will discuss the implementation of the traditional compressed row, compressed column and coordinate storage scheme. These storage schemes have enjoyed several decades of research and are the most commonly used storage schemes [76]. The compressed row and column have minimal storage requirements and at the same time it is proved to be very convenient for several important operations such as addition, multiplication, permutation and transpose of matrices. Next we will give the formal definition of these storage schemes.

The compressed row storage (CRS) format puts the subsequent nonzeros of the matrix rows in contiguous memory locations, as shown in Figure 5-0. Assuming we have a nonsymmetric sparse matrix ***A***, we create three vectors: one for the `double` type (`value`) and the other two for integers (`columnindex`, `rowpointer`). The `value` vector stores the values of the nonzero elements of the matrix ***A***, as they are traversed in a row-wise fashion. The `columnindex` vector stores the column indexes of the elements in the `value` vector. That is, if *value(k)* = $a_{i,j}$ then columnindex(*k*) = *j*. The `rowpointer` vector stores the locations in the `value` vector that start a row, if *value(k)* = $a_{i,j}$ then *rowpointer(i)* <= *k* < *rowpointer(i+1)*. By convention, we define *rowpointer(n+1) = nnz +1*, where *nnz* is the number of nonzeros in the matrix ***A***, the storage savings for this approach is significant. Instead of storing n^2 elements, we only need *2nnz + n +1* storage locations.

As an example, consider the non-symmetric *6x6* ***A*** matrix:

$$A = \begin{Bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{Bmatrix}$$

This matrix is stored on CRS, CCS and COO, as shown in Figure 5-0, 5-1 and 5-2 respectively.

Value	**10**	**-2**	**3**	**9**	**3**	**7**	**8**	**7**	**3**	**8**	**7**	**5**	**8**	**9**	**9**	**13**	**4**	**2**	**-1**
Columnindex	**1**	**5**	**1**	**2**	**6**	**2**	**3**	**4**	**1**	**3**	**4**	**5**	**2**	**4**	**5**	**6**	**2**	**5**	**6**

Rowpointer	**1**	**3**	**6**	**9**	**13**	**17**	**20**

Figure 5-0 The compressed row storage scheme of matrix *A*.

Value	10	3	3	9	7	8	4	8	8	7	7	9	-2	5	9	2	3	13	-1
Rowindex	1	2	4	2	3	5	6	3	4	3	4	5	1	4	5	6	2	5	6

Columnpointer	1	4	8	10	13	17	20

Figure 5-1 The compressed column storage scheme of matrix *A*.

Value	10	-2	3	9	3	7	8	7	3	8	7	5	8	9	9	13	4	2	-1
Columnindex	1	5	1	2	6	2	3	4	1	3	4	5	2	4	5	6	2	5	6
Rowindex	1	1	2	2	2	3	3	3	4	4	4	4	5	5	5	5	6	6	6

Figure 5-2 The coordinate storage scheme of matrix *A*.

The CCS format, as shown in Figure 5-1, is identical to the CRS format except that the columns, instead of the rows of $\boldsymbol{A}$ are stored (traversed). In other words, the CCS format is the CRS format for $\boldsymbol{A}^T$. The CCS format is specified by the 3 arrays {`value`, `rowindex`, `columnpointer`}, where `rowindex` stores the row indices of each nonzero and `columnpointer` stores the index of the elements in `value` which start a column of $\boldsymbol{A}$.

The coordinate storage format is also specified by three arrays {`value, rowindex, columnindex`} where `rowindex` stores the row indices of each nonzero and `columnindex` stores the columns indices for each nonzero, as shown in Figure 5-2. The COO format has no pointer array. The coordinate storage can be sorted after rows (COOR) or column indexes (COOC). We need *3nnz* storage locations for the COOC and COOR structures.

First we will see if there is any difference in performance between CRS, CCS , COOR and COOC when implemented in Java and on the following operations: $\boldsymbol{Ab}$, $\boldsymbol{b}^T\boldsymbol{A}$ and $\boldsymbol{AB}$.

5.3 Implementation of some Basic Linear Algebra routines

We have implemented $\boldsymbol{Ab}$, $\boldsymbol{b}^T\boldsymbol{A}$ and $\boldsymbol{AB}$ routines on CRS, CCS, COOC and COOR, by using Java arrays for storing the nonzero structure.

Next we show how we store the non-symmetric *6x6 A* matrix presented in the section before with Java arrays on the CRS, CCS, COOR and COOC format.

The CRS storage format:

```
double[] value =    {10,-2,3,9,3,7,8,7,3,8,7,5,8,9,9,13,4,2,-1};
int[] columnindex = {1,5,1,2,6,2,3,4,1,3,4,5,2,4,5,6,2,5,6};
int[] rowpointer =  {1,3,6,9,13,17,20};
```

The CCS storage format:

```
double[] value =    {10,3,3,9,7,8,4,8,8,7,7,9,-2,5,9,2,3,13,-1};
int[] rowindex =    {1,2,4,2,3,5,6,3,4,3,4,5,1,4,5,6,2,5,6};
int[] columpointer ={1,4,8,10,13,17,20};
```

The COOR storage format:

```
double[] value =    {10,-2,3,9,3,7,8,7,3,8,7,5,8,9,9,13,4,2,-1};
int[] columnindex = {1,5,1,2,6,2,3,4,1,3,4,5,2,4,5,6,2,5,6};
int[] rowindex =    {1,1,2,2,2,3,3,3,4,4,4,4,5,5,5,5,6,6,6};
```

The COOC storage format:

```
double[] value =   {10,3,3,9,7,8,4,8,8,7,7,9,-2,5,9,2,3,13,-1};
int[] rowindex =   {1,2,4,2,3,5,6,3,4,3,4,5,1,4,5,6,2,5,6};
int[] columnindex ={1,1,1,2,2,2,2,3,3,4,4,4,5,5,5,5,6,6,6};
```

When implementing the $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$ routines on CRS and CCS we get a rather similar implementation as of FORTRAN, C and C++ [13, 14]. It is not surprising since Java arrays can be used in similar ways as arrays in FORTRAN, C and C++. The fixed size of Java arrays causes no problems implementing these routines because we do not use Java arrays in a dynamic way (for example insertion or deletion of nonzero elements).

We consider matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ to have nonzero element in their rows and that the $\boldsymbol{b}$-vector is a full vector. This shall result in a full $\boldsymbol{c}$-vector in doing $\boldsymbol{c}=\boldsymbol{Ab}$ and $\boldsymbol{c}=\boldsymbol{b^TA}$. Since the $\boldsymbol{b}$-vector is full and the matrix $\boldsymbol{A}$ has nonzero element in their rows, we can create the resulting $\boldsymbol{c}$-vector before the actual numerical computation. Therefore implementing $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$ on CRS, CCS, COOR and COOC are straightforward. The problem arises when doing the matrix multiplication operation on CRS, CCS, COOR and COOC. We do not know the structure of the nonzero resulting matrix before we do the actual multiplication, this will be further analyzed in section 5.3.4.

In many of the routines presented in this chapter we, use additional accumulators, as explained in 5.1, to store temporary results for the many routines. They are either of type `int[]`, `double[]` and/or `boolean[]`. To get a clear idea of how much memory we use when declaring and initialsing those accumulators we must separate between the `int[]` and `boolean[]` which uses 32 bits for storing each element in its arrays and `double[]` which uses 64 bits for storing each element in its array. So when we compare routines on the basis of memory we must not only compare them on the numbers of accumulators they use, but also which types of accumulators they use.

5.3.1 CCS, CRS, COOR and COOC on *Ab*

Figure 5-3 shows the time testing of the $\boldsymbol{Ab}$ routine on CRS, CCS, COOR and COOC. We can see there is a clear and increasing difference between the CRS and CCS and between COOR and COOC, for large matrices, approximately $nnz>100000$. There is also a slightly difference between CCS and CRS, CRS is slightly more efficient than CCS. The same difference is there for COOR and COOC, COOR is slightly more efficient than COOC.

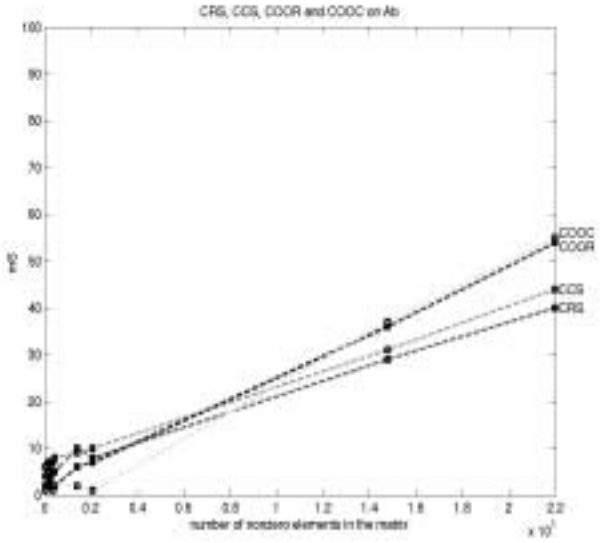

Figure 5-3
CCS, CRS, COOR and COOC on *Ab*.

The difference between the row and column storage scheme will be reversed when we consider $\boldsymbol{b^T A}$, as we will show in the next section.

5.3.2 CCS, CRS, COOR and COOC on $b^T A$

Figure 5-4 shows the time testing of the $\boldsymbol{b^T A}$ routine on CRS, CCS, COOR and COOC. As we can see from Figure 5-4 there is clear and increasing difference between the CRS/CCS structures and COOR/COOC for large matrices, approximately *nnz*>100000. There is also a slight difference between CCS and CRS, were CCS is slightly more efficient than CRS. The same difference is there for COOR and COOC, COOC is slightly more efficient than COOR.

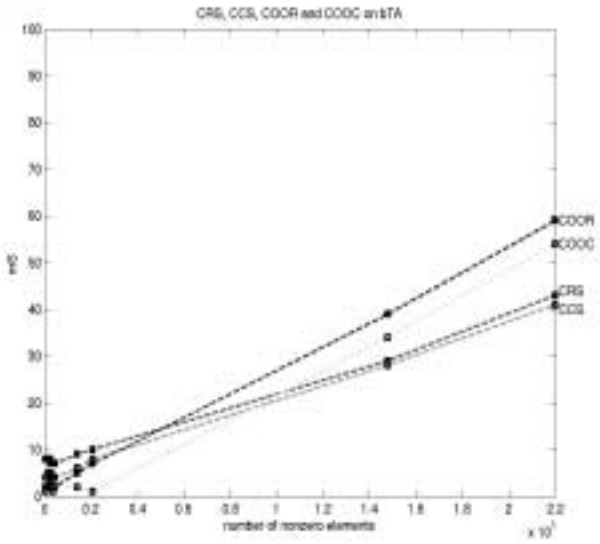

Figure 5-4
CCS, CRS and COOR and COOC on $b^T A$

There is not a significant difference between CCS and CRS on the basis of these two routines. If we look at the implementation of the two routines they are rather similar,
see Appendix B [The classes MatrixCCS and MatrixCRS]. The most obvious difference is that they have different constructions in the innermost for-loop. For the ***Ab*** routine, CCS has a different construction in its innermost for-loop then the ***Ab*** routine on CRS.

For ***Ab***:

```
CCS:  Y[rowindex[startrow]] += valuea[j]*value[startrow];
CRS:  z+=valuea[columnindex[startrowa]]*value[startrowa];
```

For $\boldsymbol{b^T A}$:

```
CRS: Y[i]+=value[startrowa]*value[columnindex[startrowa ]];
CCS:  z+=valuea[rowindex[startrow]]*value[startrow];
```

The construction:

```
z+=valuea[columnindex[startrowa]]*value[startrowa];
```

may be less efficient to execute then the next construction, because array access is 2 or 3 times as expensive as accessing non-array elements, see Chapter 2, section 2.8, and the next construction have more array access then the construction above:

```
Y[rowindex[startrow]] += valuea[j]*value[startrow];
```

The CCS routine cannot access the rows of matrix it stores in O(*n*) time, instead it must do the ***Ab*** operation by a partial storage. This leads to a seemingly more complex construction then for that of CRS. This gives us a clear indication that the difference between CCS and CRS on ***Ab*** is the result of the difference in the innermost construction.

There are not any fundamental differences between performing a matrix-vector or vector-matrix products in Java then in FOTRAN, C and C++, which makes it execute more efficiently. It is rather straightforward implementations and we can conclude the same as we do for the same operations implemented in FORTRAN and C. Using Java arrays does not allow us to implement matrix-vector or vector-matrix products that is more efficient then FORTRAN and C implementations.

5.3.3 CCS, CRS, COOR and COOC on *AB*

Different data structures will favour different loop-orders performing matrix multiplication operations. This can be summarised as follows:

The implementation of the ***AB*** operation on:

- CRS has the loop-order (i,k,j). We build the result matrix row-by-row.
- CCS has the loop-order (j,i,k). We build the result matrix column-by-column.
- COOR has the loop-order (i,k,j). We build the result matrix row-by-row.
- COOC has the loop-order (j,i,k). We build the result matrix column-by-column.

See Chapter 3, section 3.6, for better definitions of loop-orders.

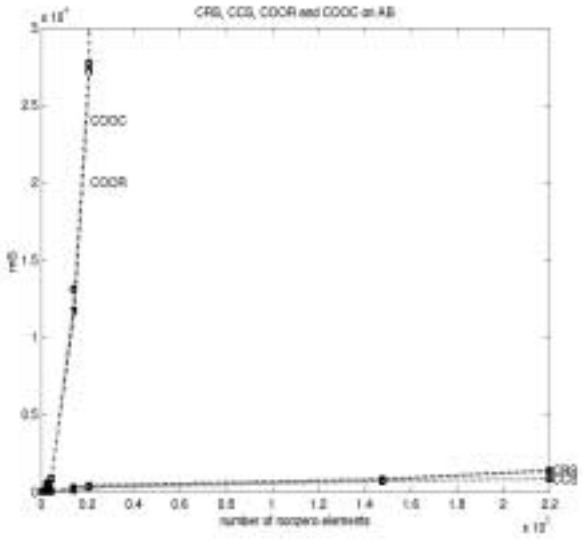

Figure 5-5
CCS, CRS, COOC and COOR on *AB*.

Figure 5-5 shows the time testing of the ***AB*** routine on CRS, CCS, COOR and COOC. We can see from Figure 5-5 that there is a significantly difference between CRS/CCS and COOR/COOC. There is also a slightly difference between CCS and CRS, were CCS is slightly more efficient than CRS. The main difference between CRS/CCS and COOR/COOC is that we have extensive if-testing for each nonzero element in the ***AB*** routine for COOC and COOR.

The idea behind the matrix multiplication routine for CRS and CCS are the same, and the idea for matrix multiplication routine for COOC and COOR are the same. When implementing the routines on COOR and COOC we have a significantly difference. Looping through rows in CRS we have pointers that indicates where the row starts and where it ends in the value array and index array. We do not have any pointers in the COOR and COOC structures. The consequence of this is that we cannot loop through a row or column without testing if we are actually traversing the row or the column or if we have began traversing a new row or column. We must test the row index or the column index, by testing with an if-statement. That is exactly what we have done when implementing the routines on COOR and COOC. There is a cost of executing this if-statement. We have seen that there is a clear difference between CRS/CCS and COOR/COOC on $\boldsymbol{Ab}$, $\boldsymbol{b^TA}$ and $\boldsymbol{AB}$, this if-statement testing is the main reason for that difference.

The COOR and COOC format cannot compete with CRS or CCS on any of the routines in performance and they also use additional memory to store its nonzero values. This storage scheme is suitable for storing matrices that we read from a file before we eventually translate it into another storage scheme that is more suitable considering its nonzero structure or pattern.

The implementation of the ***AB*** routine on CRS, CCS, COOR or COOC was not straightforward. We will in the next sections implement two different ***AB*** routines and discussing the advantages and disadvantages of these implementations. We are only considering matrix multiplication routines on the CRS format, considering that there are symmetry between CRS and CCS that both works for the rows of CRS as for the columns of CCS.

5.3.4 Matrix Multiplication on Compressed Row Storage

In this section we are interested in finding an efficient algorithm for matrix multiplication on CRS using Java's native arrays.

Implementing routines for $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$ are rather straightforward. Pseudocodes for these routines given in FORTRAN syntax [13, 14] can easily be adapted to Java. When it comes to routines for the matrix multiplication operation it is not so straightforward. The reason for this is that we do not know the pattern or the number of nonzero elements of the resulting matrix without doing a symbolic multiplication [15, 17, 18]. That means we cannot create the exact nonzero structure of the resulting matrix and perform the numerical multiplication without doing the symbolic multiplication first.

This section deals with CRS, the same ideas presented here also apply for CCS but with a column-oriented structure.

Next in this section we present two algorithms for performing a matrix multiplication on CRS, where both the input matrices are on CRS and the resulting matrix is on CRS using Java's native arrays. The first one is based on [77] implementation of matrix multiplication in FORTRAN, the second is based on [15,17,18] implementation of matrix multiplication in FORTRAN.

1. The Guesstimate Approach (Creating an arbitrary sized structure)

   In this approach we create the structure, the value array and index array, we perform the numerical multiplication on at arbitrary size as explained above. There is no need to create the pointer array at arbitrary size since it is the size of the row dimension of $\boldsymbol{A}$ $\boldsymbol{(C=AB)}$, plus one extra element place. The 'guesses' of the size can be done in various ways, we have however given a variable as an input to the routine. This variable is multiplied with the value that gives the nonzero value (of the $\boldsymbol{A}$ matrix). The result of this multiplication gives the size of the arbitrary nonzero structure.

   The problem when creating an arbitrary sized structure is, if it is smaller then the actual nonzero structure. If this happens we must resize the arbitrary structure and begin all over again or from where we stopped in the routine

   After we have done the numerical multiplication, the arbitrary sized structure may contain zero elements. If we want to eliminate those zero elements, we have to create the actual nonzero structure and copy the elements from the arbitrary structure to the actual nonzero structure.

   The problem by creating an arbitrary sized structure is that we can fail multiple times, by creating a size that are equal to or larger then the size of the resulting matrix. This can cause a very unpredictable routine with bad efficiency and if we create an arbitrary sized structure that is much larger then the actual matrix we may have a memory problem. This unpredictability may be a reason for not choosing this routine for many performance critical applications.

The guesstimate approach performs well if it is completed the first time. It has, not surprisingly, the best performance all of the matrix multiplication routines we present in this chapter, see Table 5-0 and Table 5-3.

The implementation of this routine in Java is in Appendix B [The class MatrixCRS]

2. The Symbolic Multiplication Approach

The usual approach for performing a sparse matrix multiplication, is to do a symbolic multiplication for finding the nonzero structure, then performing the numerical multiplication on that structure [15, 17, 18].

The idea behind this approach is to find the nonzero structure of each row (the index array and the pointer array for CRS), then store it in temporary arrays and expand this structure for each row created. Since we know the dimension of the resulting matrix we can create the row pointer array. We must have an index array that holds the indexes for a new row and to add it to the index array that holds all the rows created so far. Then we must create a new index array with a size, so that we can include the original rows and the new row. We must copy both the original index rows and the newly created row to the new temporary array. This operation must be done for each new row created.

After creating the index array and initialized the pointer array, we perform the numerical multiplication. This can be done since we know the size of the value array to be the same as the already created index array.

The most time consuming part of this routine is the extensive updating of the index array performing the symbolic multiplication. We can do this updating in Java either by a loop or by using the `arraycopy()` method form the `System` class, see section 5.7.

These next examples illustrates both of these techniques:

```
int[] indextemp = new int[n];
int[] indexk = new int[k];
int indexnew = new int[n+k];
```

These three arrays illustrate the arrays involved in updating the temporary structure. The `indextemp` array illustrates the array that holds the indexes of the rows created so far. The `indexk` array holds the index values of the row newly created and finally the `indexnew` array holds both the values of `indextemp` and `indexk.`

First we show how we can do this by for-loops:

```
for(int i = 0;i<n;i++){
     indexnew[i] = indextemp[i];
}
for(int i = indextemp.length, ii = 0;i<k;i++,ii++){
     indexnew[ii] = indexk[i];
}
```

Second we show how we can do this by using `System.arraycopy()`:

```
System.arraycopy(indextemp,0,indenew,0,n);
System.arraycopy(indexk,0,indexnew,n,k);
```

The larger the array the more useful it is to use `System.arraycopy()`, the reason for this is that the method is compiled to native code [65].

The implementation of this routine in Java is in Appendix B [The class `MatrixCRS`].

The problem with the symbolic multiplication approach is that we have to store a temporary structure consisting of the rows and pointers that has been created so far. When a new row is created, we must update the temporary structure. If we use Java arrays to store the temporary structure, we must update it for each new resulting row. This means that we have to resize the temporary structure so that we can copy the new resulting row. This can be so time consuming that the guesstimate approach may be tempting. This approach though, will be completed, but when, we do not know.

The ***AB*** routine			
Matrix Dimension	Nonzeros	CRS on ***AB*** (guesstimate approach)	CRS on ***AB*** (symbolic approach)
115	421	0	7
468	2820	5	154
2205	14133	26	2574
4884	147631	733	68056
10974	219512	765	195711
17282	553956	2859	NA

Table 5-0
The CRS, guesstimate and symbolic approach, on input *AB*.

Table 5-0 shows clearly that the guesstimate approach, when completed at the first run, is much more efficient then the symbolic approach. For Nonzeros = 219512 the guesstimate approach is approximately 256 times more efficient then the symbolic approach. This makes the symbolic approach a rather unacceptable bad routine. The reason is also very clear, first we perform the routine 'two'-times, one for the symbolic multiplication and one for the numerical multiplication, second we do extensive updating for each row that is created in the symbolic routine. The only thing that is positive with the symbolic approach is that it will be completed at the first run.

We now have two rather 'easily' implemented matrix multiplication routines on CRS. But what we are looking for, is an algorithm that is more reliable then the guesstimate approach and more efficient then the symbolic approach. One way achieve this maybe to concatenate the rows of the resulting matrix ***C*** into its structure without needing to update or traverse all of the rows already inserted in the structure, like we do in the symbolic multiplication on CRS.

The way we use Java arrays in the symbolic multiplication on CRS is not satisfactory. In the next section we introduce the Sparse Arrays format that addresses the problem of concatenating rows without needing to update the rest of the nonzero structure.

Since this thesis do not involve extensive development (coding) of routines in FORTRAN, C and C++, it is therefore difficult to give a complete discussion on the transition from implementing numerical routines in FORTRAN, C and C++ to implement the same routines in Java. We only deal with limited number of routines in this thesis and most of them were taken from [64 ,11-18**,** 25] and all of their routines was implemented using FORTRAN. This means we only considered methods implemented in FORTRAN. The transition from FORTRAN to Java was for elementary operations, like matrix multiplication, matrix-vector and vector-matrix rather straightforward. We can access a FORTRAN array in a similar manner as we can with a Java array and these arrays are fundamental storage schemes for these operations. The problem first arises when we were to implement routines that worked on graph structures. Since FORTRAN is not an object-oriented language we do not create these graph structures as object-oriented data structures. Therefore it is not straightforward taking the transition from FORTRAN to Java with routines that operates on graphs. For an example of routines in FORTRAN that works on graph structure see [25]. We tried to do such a transition for a Minimum Degree and a Symbolic Factorisation (Cholesky method). The routines for FORTRAN we considered were from [25], but we got a very complex Java routine. One reason for that was that we had to translate *a lot of* goto statements from FORTRAN to `do-while` loops in Java (`break` and `return` statements could also worked, but these statements are considered to be bad implementations in Java). So the conclusions is that translating simple routines that works on data structures and to a degree has the same syntax in FORTRAN as in Java is straightforward. When we are dealing with data structures that in Java is strictly object-oriented, while in FORTRAN it is not, the transition is not so simple. These assumptions may seem weak and they are because we have not done enough transition from FORTRAN to Java, to be able to discuss all eventualities, but even so they may give some indications on the problems of such transitions.

5.4 Sparse Arrays

In this section we introduce a new concept for storing sparse matrices. This concept is special for Java, because here we have arrays of arrays where each array is an object.

This concept is different from a CRS concept in the following way:

1. We can manipulate the rows of Sparse Arrays independently without the need for updating the rest of the structure.
2. Each row has its own unique reference(s), each row consists of one value array and one index array.
3. We use *2nnz + 2n* storage locations for Sparse Arrays compared to *2nnz + n + 1* for CRS.

If we declare a `double[][]` and an `int[][]`, we have the possibility to store the references of `double[]`'s and `int[]`'s. The lengths of the `double[]`'s and `int[]`'s does not need the be of the same length. This feature is what we use to create the Sparse Arrays structure. It is also illustrated in Chapter 1, Figure 1-3.

Consider the non-symmetric *6x6* ***A*** matrix:

$$\boldsymbol{A} = \begin{Bmatrix} 10 & 0 & 0 & 0 & -2 & 0 \\ 3 & 9 & 0 & 0 & 0 & 3 \\ 0 & 7 & 8 & 7 & 0 & 0 \\ 3 & 0 & 8 & 7 & 5 & 0 \\ 0 & 8 & 0 & 9 & 9 & 13 \\ 0 & 4 & 0 & 0 & 2 & -1 \end{Bmatrix}$$

We can store the nonzero structure of this matrix as follows:

The index array:

```
int[][] index = {{0,2},
                 {0,1,6},
                 {1,2,3},
                 {0,2,3,4},
                 {1,3,4,5},
                 {0,1,3,5}
               };
```

The value array:

```
double[][] value = {{10, -2},
                    {3,9,3},
                    {7,8,7},
                    {3,8,7,5},
                    {8,9,9,13},
                    {4,2, -1}
                    };
```

In `value[i]` and `index[i]` , `i=0,…,value.length-1` we can store the references to `double[]` arrays. Those arrays we store in `value` can be replaced, accessed and manipulated without traversing and updating the rest of the arrays.

There is a trade-off using Sparse Arrays. The trade-off is that we must use more memory to store a nonzero element than we did with CRS and CCS. We have two arrays, one for storing the references to the value arrays and one for storing the references to the index arrays. This means we have one more array with the dimension size, then CRS. The total storage locations for Sparse Arrays are *2nnz + 2n* compared to *2nnz + n + 1* for CRS as we saw in section 5.2

Further we shall see that by using Sparse Arrays, there is a more practical solution to perform a matrix multiplication. This because we can directly insert both index array and value array into the structure, without needing to update or traverse the rest of the structure.

If we have the dimensions to both ***A*** and ***B***, we know the dimensions of the resulting matrix ***C***. If we were to multiply ***A*** and ***B***, where ***A*** is *mxn* and ***B*** is *nxp* that means ***C*** is *mxp*. We can therefore construct `double[m][1]` and `int[m][1]` to contain the rows of the resulted matrix ***C,*** before we actually creates the actual rows of the resulting matrix ***C***.

What we are looking for in a data structure that stores large sparse matrices in Java, is a structure that is to a degree object-oriented. A structure where we can implement simpler algorithms compared to other alternative structures and where we can compete with other structures when it comes to performance and memory use, Sparse Arrays may give these possibilities.

A SparseArrays 'skeleton' class will look like this:

```
public class SparseArrays{
     private double[][] Avalue;
     private int[][] Aindex;
     private int nonzero;
     private int dimension;
     public SparseArrays(double[][] A, int[][] B, int nonzero){
          this.A = A;
          this.B = B;
          this.nonzero = nonzero;
          this.dimension = A.length;
     }
     public Vector matrixvector(Vector v){…}
     public Vector vectormatrix(Vector v){…}
     public SparseArrays times(SparseArrays B){…}
     public SparseArrays timesA(SparseArrays B){…}
     public SparseArrays timesB(SparseArrays B){…}
     public SparseArrays add(SparseArrays B){…}
     public SparseArrays addition(SparseArrays){…}
     public void symbolicFactorisation(){…}
}
```

From this 'skeleton' class we can see that the we have `double[][]` and `int[][]`, arrays for storing value rows and index rows respectively, as instance variables to the `SparseArrays` class. That means we access these instance variables in the same manner as we do for the `Matrix` class (from the JAMA [2] package) we introduced in Chapter 3, section 3.3.

5.4.1 The Sparse Matrix Concept

The Sparse Matrix Concept is a general object-oriented structure that do not take advantages of the features of the Java language, to show this we will next illustrate two approaches to perform a matrix-vector product with the Sparse Matrix Concept. This concept we introduce for storing sparse matrices may seen similar to Sparse Arrays, but it does not take advantage of the features that we have with Sparse Arrays.

The Sparse Matrix Concept can be implemented like this [63]:

```
public class SparseMatrix{
     private Rows[] rows;
     public SparseMatrix(Rows[] rows){
          this.rows=rows;
     }
}
public class Rows{
     private double[] values;
     private int[] indexes;
     public Rows(double[] values, int[] indexes){
          this.values = values;
          this.indexes = indexes;
     }
}
```

This data structure has similarities with Sparse Arrays. The concept is the same, using `double[]` and `int[]` for storing values and indexes of the nonzero elements of the matrix, but we do not use the extra object layer for each row in the Sparse Array. That extra object layer may cause difference in time performance when comparing them for various operations. The Sparse Matrix Concept is also more memory demanding because of this extra object layer.

We have two ways of representing the Sparse Matrix Concept:

- The first approach is to make it as similar as Sparse Arrays as possible, by only storing the row arrays (values and indexes) in the `Rows` object, and not place any methods that work explicitly on that row. The problem here is the `Rows` object, since it only adds overhead accessing these arrays. The code for matrix-vector product for this approach is given in Appendix C [The classes `SparseMatrix` and `Rows`].

- The second approach is that we have an object-oriented implementation of the Sparse Matrix concept, where we place the methods that work explicitly on the arrays (value and index) in the `Rows` objects. It is natural to add features that work on the instances of the `Rows` object in that particular class. The code for matrix-vector product for this approach is given in Appendix C [The classes `SparseMatrix` and `Rows`].

The first implementation of the Sparse Matrix concept is similar to Sparse Arrays. But since Java can store arrays of arrays it is preferable, considering both performance (speed) and memory, to use Sparse Arrays instead. If we really were to implement the Sparse Matrix concept the second implementation would be natural considering an object-oriented approach, but as we can see from the timings of the ***Ab*** routine on this last approach it is less efficient and it demands more memory (the extra `Rows` object). So again we prefer the Sparse Arrays approach for implementing this routine, see Table 5-1.

The ***Ab*** routine				
Matrix Dimension	Nonzeros	Sparse Arrays on ***Ab***	The Sparse Matrix Concept on ***Ab*** (first approach)	The Sparse Matrix Concept on ***Ab*** (second approach)
115	421	0	0	3
468	2820	0	0	3
2205	14133	3	3	5
4884	147631	20	21	28
10974	219512	31	32	43
17282	553956	72	77	99

Table 5-1
Sparse Arrays and the Sparse Matrix concept on *Ab*. Time is in milliseconds (mS).

5.5 Implementation of some basic Linear Algebra routines on Sparse Arrays

When we introduce a new data structure like Sparse Arrays it must have some features that make it justifiable to use. Therefore we implement the ***Ab*** and the $\boldsymbol{b^TA}$ routine. We choose these two routines to illustrate that we have a performance consistency between CRS and Sparse Arrays on these routines. This is important, because it gives us an indication that we can replace CRS with Sparse Arrays even for routines that do not take advantage of the possibility of manipulating the rows of Sparse Arrays independently, as the case is for *Ab* and b^TA routines. Next we will take a closer look at some routines that takes advantage of the possibility to manipulate the rows of Sparse Arrays, like matrix multiplication, matrix addition, insertion and deletion of elements and permutation of rows. When we say take advantage of, we mean compared to a similar routine implemented on CRS, that has to update the whole nonzero structure. All these routines will be compared against the implementation of CRS.

5.5.1 Sparse Arrays on *Ab*

The first basic linear algebra routine we implemented on Sparse Arrays is ***Ab:***

```
public Vector matrixvector(Vector V){
          double[] b = V.getValuearray();
          double x = 0.00;
          double[] value, c;
          int[] index;
          int k = 0,
          int vlength = 0;
          int alength = Avalue.length;
          c = new double[b.length];
          for(int i = 0;i<alength;i++){
               value = Avalue[i];
               index = Aindex[i];
               vlength = value.length;
               for(int j = 0;j<vlength;j++){
                    x += value[j]*b[index[j]];
               }
               c[i]=x;
               x = 0;
          }
          return new Vector(c, c.length);
}
```

Here we do an inner product between the ***b***-vector and all of the rows of matrix ***A,*** each result of an inner product is an element in the resulting ***c***-vector. The outermost for-loop access all the rows of matrix ***A***, while the innermost for-loop actually performs the inner product. We index the ***b***-vector with the index array of a row in ***A***. Here we create one array object which is the resulting ***c***-vector. This implementation is very similar to the ***Ab*** implementation on CRS. But we do not benefit from the fact that we can manipulate the rows of matrix ***A*** on Sparse Arrays independently from the rest of the nonzero structure.

Remark! Before we do the actual inner product in the innermost for loop we get the row of ***A*** like this:

```
value = Avalue[i];
index = Aindex[i];
```

This is important because we getter better performance traversing `double[]` instead of `double[][]`, as shown in Chapter 3, Figure 3-7. This would also apply for $\boldsymbol{b^TA}$, see the next section.

Table 5-2, in section 5.5.2, shows that we can replace the CRS structure for Sparse Arrays and still have a consistent performance on the ***Ab*** routine.

5.5.2 Sparse Arrays on b^TA

The second basic linear algebra routine we implement on Sparse Arrays is b^TA:

```
public Vector vectormatrix(Vector v){
     double[] value = null;
     double[] bvalue = v.getValueArray();
     int[] indexrow = null;
     int alength = Avalue.length;
     double[] cvalue = new double[bvalue.length];
     double value = 0;
     for(int i = 0;i<alength;i++){
          valuerow = Avalue[i];
          indexrow = Aindex[i];
          val = bvalue[i];
          int vlength = valuerow.length;
          for(int j = 0;j<vlength;j++){
               cvalue[indexrow[j]] += value*valuerow[j];
          }
     }
     return new Vector(cvalue, cvalue.length);
}
```

Since we cannot access the columns of the Sparse Arrays, we perform a partial storage. For each element of the ***b***-vector, we traverse the rows of the matrix ***A.*** and multiplies the ***b*** element with all of the elements of the compressed row. The result is stored in the place given by the index vector of the matrix ***A***. The only object we create in this routine is the resulting ***c***-vector.

Table 5-2, shows that we can replace the CRS structure for Sparse Arrays and still have a consistent performance on the b^TA routine.

The Ab and b^TA routines					
Matrix Dimension	Nonzeros	Sparse Arrays		CRS	
		Ab	b^TA	Ab	b^TA
115	421	0	1	0	0
468	2820	0	0	1	0
2205	14133	3	2	2	2
4884	147631	20	18	21	24
10974	219512	30	29	34	34
17282	553956	72	68	83	87

Table 5-2
CRS and Sparse Arrays on the routines *Ab* and b^TA. Time is in milliseconds (mS)

The results in Table 521 are very interesting because they give us an indication that we can replace CRS totally. But of course as shown in Chapter 4, performance is platform dependent. The results tells us that we cannot expect that a Ab or a b^TA operations is significantly more efficient on CRS then on Sparse Arrays for the same operations on any platform.

To be able to use Sparse Arrays for *any* operations, not only a limited number of operations, makes it a more powerful data structure.

These implementations (***Ab*** and $\boldsymbol{b^TA}$) are very similar to a ***Ab*** and $\boldsymbol{b^TA}$ implementation on CRS, see Appendix B [The class `MatrixCRS`]. But we do not benefit from the fact that we can manipulate the rows of Sparse Arrays independently from the rest of the structure. What we are trying to show here is how to implement a $\boldsymbol{b^TA}$ operation on Sparse Arrays compared to $\boldsymbol{b^TA}$ on CRS and as a result see that the implementation is not so different. This means that going from CRS to Sparse Arrays may not be such a huge transformation.

5.5.3 Sparse Arrays on *AB*

In this section we will show how matrix products can be implemented in a very elegant and efficient way using Sparse Arrays:

- The SPARSEKIT approach [77]. This approach means that we store each row element contiguously in an array (values and indexes).
- The Pissanetsky approach. This approach means that we perform the 'Merging sparse list of integers and addition of sparse vectors' [5, 6], as explained in section 5.1.

When it comes to performance of these two implementation Table 5-3 shows that the SPARSEKIT approach is the most efficient routine for the test matrices.
Considering the memory use, the first approach uses three additional accumulators (two `int[]` and one `double[]`) and creates two new array objects for each new row. The second approach uses two additional accumulators (one `double[]` and one `boolean[]`) and creates two new array objects for each new row, see section 5.1 for definition of accumulators. The two array objects we create for each new row is one `double[]` and one `int[]`, storing only the nonzero elements for each row.

The implementation of matrix multiplication routine on Sparse Arrays (SPARSEKIT approach):

```
public SparseArrays timesA(SparseArrays B){
     double[][] Cvalue = new double[dimension][1];
     int[][] Cindex = new int[dimension][1];
     int[] temp = new int[dimension];
     double[] tempValue = new double[dimension];
     int[] tempIndex = new int[dimension];
     double[][] Bvalue = B.getValueArray();
     int[][] Bindex = B.getIndexArray();

     double scalar = 0;
     int len = -1;
     int index = 0;
     int jcol = 0;
     int jpos = 0;
     int nonzero = 0;

     for(int i = 0;i<temp.length;i++){temp[i]=-1;}
     for(int i = 0;i<Avalue.length;i++){
          double[] avalue = Avalue[i];
```

```
            int[] aindex = Aindex[i];
            for(int j = 0;j<avalue.length;j++){
                 scalar = avalue[j];
                 index = aindex[j];
                 double[] bvalue = Bvalue[index];
                 int[] bindex = Bindex[index];
                 for(int k = 0;k<bvalue.length;k++){
                      jcol = bindex[k];
                      jpos = temp[jcol];
                      if(jpos == -1){
                           len++;
                           nonzero++;
                           tempIndex[len] = jcol;
                           temp[jcol] = len;
                           tempValue[len] = scalar*bvalue[k];
                      }else{
                           tempValue[jpos]+=scalar*bvalue[k];
                      }
                 }
            }
            double[] cvalue = new double[len+1];
            int[] cindex = new int[len+1];
            System.arraycopy(tempValue, 0, cvalue, 0,len+1);
            System.arraycopy(tempIndex, 0, cindex, 0,len+1);
            Cvalue[i] = cvalue;
            Cindex[i] = cindex;
            for(int ii = 0;ii<len+1;ii++){temp[tempIndex[ii]]=-1;}
            len = -1;
       }
       return new SparseArrays(Cvalue, Cindex, nonzero);
}
```

Since we have a row-oriented Sparse Arrays, we perform a matrix multiplication with the loop-order (i,k,j). This means we traverse the matrices involved row-wise and we create the resulting matrix row-by-row. Remember that here we do a sparse matrix multiplication. For each row of matrix ***A*** we traverse those rows of ***B***, which is indexed by the index array of the sparse row of ***A***. Each element in the value array of ***A***, is multiplied with the value array of ***B***, which is indexed by the index array of that row. The result is stored in the result row of ***C,*** which is indexed by the index value of the element in the ***B*** row.

Next we will give a more detailed explanation on how we use the 'Contiguously storing of nonzero values' approach. The explanation takes into consideration that the reader is familiar with the general sparse matrix multiplication on CRS, with loop-order (i,k,j).
The key to this approach is the accumulator, called `temp` in the routine above, which is initialized to –1 (all integers less then 0 would do). Each time we encounter an element we want to insert into the new resulting row, we test with the accumulator if its index is set to –1 or a value greater then –1 in the `temp` array. If it is –1 we can directly insert the new resulting array in the next free position, in the `tempValue` and `tempIndex` array and we update the `temp` array with the index value where that element is in the new resulting array. If it is different from –1 we already have an element in that position in the new resulting array and we must therefore

perform an addition with the element already inserted in the `tempValue` array. The value from the temp array (which is different from -1), gives us the index value that indicates where in the `tempValue` we are storing the result of the above addition. The main idea here is that we now have an unsorted (unordered) representation of one row in the resulting matrix. The elements are stored contiguously in the `tempValue` and `tempIndex` array not in their real index positions, the benefit from this is that we can traverse those arrays with a loop or `System.arraycopy()`, see section 5.8, to copy the elements from the expanded accumulators to the index array and the value array. The reason why this may be a more efficient method then traversing the index arrays of ***B***, is that we do not traverse elements already dealt with (merged elements) like we do in the Pissanetsky approach . See section 5.1 about accumulating elements from the expanded accumulators stored in their real position to compressed arrays which only stores the nonzero values. This may also give an explanation on why there is a difference between the efficiency of these two routines, as shown in Table 5-3. The SPARSEKIT approach is approximately 7 times more efficient then the Pissanetsky approach, for Nonzero = 553956.

The ***AB*** routine			
Matrix Dimension	Nonzeros	Sparse Arrays on ***AB*** (SPARSEKIT) The values in parenthesis are the times using `for`-loops instead of `System.arraycopy()`	Sparse Arrays on ***AB*** (Merging/Addition)
115	421	1 (1)	4
468	2820	7 (6)	62
2205	14133	27 (31)	364
4884	147631	992 (1025)	2794
10974	219512	1267 (1262)	9155
17282	553956	4339 (4504)	30770

Table 5-3
Sparse Arrays (the Pissanetsky and SPARSEKIT approach) on the *AB* routine. Time is in milliseconds.

When comparing CRS on ***AB*** *(*the guesstimate approach*)* to Sparse Arrays on ***AB*** (the SPARSEKIT approach) we have that the guesstimate approach is still more efficient, with a factor of approximately 1.5 for Nonzero = 553956. This tells us that we have a matrix multiplication that is completed at the first run and is 'only' an average of 1.4 times slower then CRS on ***AB.*** It is important to state that we cannot draw too general conclusions on the performance of these two routines on the basis of the test matrices we used in this chapter. These results give a strong indication on their overall performance and the matrix multiplication on Sparse Arrays (the SPARSEKIT approach) is both fast and reliable. It uses more memory (number of extra accumulators) then all the matrix multiplication routines presented in this chapter, excluding ***AB*** on the COOR and COOC formats. It may be a serious disadvantage using this routine if memory is an important issue in the application where it would be used. The values in parenthesis for Sparse Arrays on ***AB*** (SPARSEKIT approach) are using for-loops instead of `System.arraycopy()` when we copy the compressed elements from the accumulators (values and indexes) to the arrays we have in the Sparse Arrays structure, see Appendix C [The class `SparseArrays`] for the code.

5.5.4 Matrix Addition on Sparse Arrays

In this section we introduce a routine for matrix addition using Sparse Arrays. This routine uses the SPARSEKIT approach, see section 5.5.3 and [77].

The implementation of matrix addition routine on Sparse Arrays:

```
public SparseArrays addition(SparseArrays B){
     double[][] Cvalue = new double[dimension][1];
     int[][] Cindex = new int[dimension][1];
     int[] temp = new int[dimension];
     double[] tempValue = new double[dimension];
     int[] tempIndex = new int[dimension];
     double[][] Bvalue = B.getValueArray();
     int[][] Bindex = B.getIndexArray();

     double scalar = 0;
     int len = -1;
     int index = 0;
     int jcol = 0;
     int jpos = 0;

     for(int i = 0;i<temp.length;i++){temp[i]=-1;}

     for(int i = 0;i<Avalue.length;i++){
          double[] avalue = Avalue[i];
          int[] aindex = Aindex[i];
          for(int j = 0;j<avalue.length;j++){
                    jcol = aindex[j];
                    len++;
                    tempIndex[len] = jcol;
                    temp[jcol] = len;
                    tempValue[len] = avalue[j];
          }
          double[] bvalue = Bvalue[i];
          int[] bindex = Bindex[i];
          for(int j = 0;j<bvalue.length;j++){
                    jcol = bindex[j];
                    jpos = temp[jcol];
                    if(jpos == -1){
                         len++;
                         tempIndex[len] = jcol;
                         temp[jcol] = len;
                    tempValue[len] = bvalue[ii];
                    }else{
                    tempValue[jpos] += bvalue[ii];
                    }
               }
          }
          double[] cvalue = new double[len+1];
          int[] cindex = new int[len+1];
```

```
     System.arraycopy(tempValue, 0, cvalue, 0,len+1);
     System.arraycopy(tempIndex, 0, cindex, 0,len+1);
          Cvalue[i] = cvalue;
          Cindex[i] = cindex;
for(int ii = 0;ii<len+1;ii++){temp[tempIndex[ii]]=-1;}
          len = -1;
     }
     return new SparseArrays(Cvalue, Cindex, nonzero);
}
```

We use the same approach as we did for matrix multiplication, see section 5.5.3, but now for each resulting row we must perform an addition between row *i* of matrix ***A*** with row *i* of matrix ***B*** (***C***=***A***+***B***). We contiguously stores the result of this addition in the `tempValue` (values) and `tempIndex` (indexes) accumulators. Then we copy the elements from `tempValue` and `tempIndex` to `cvalue` and `cindex`, which is only stores the nonzero elements. These last arrays are then included in the resulting Sparse Arrays structure. Matrix Addition is also implemented with the Pissanetsky approach in Appendix C [The `SparseArrays` class].

5.5.5 Permutation on Sparse Arrays

Permutation of a matrix is an important operation in linear algebra, therefore it is interesting to see how we can benefit from the dynamic features of Sparse Arrays to perform permutation on matrices stored on Sparse Arrays.

First we will show a method of performing a row permutation on Sparse Arrays.

```
double[][] Avalue = {{1.0,2.0,3.0},
                     {2.0,5.0},
                     {1.0}
                };
int[][] Aindex = {{0,1,2},
                  {0,1},
                  {0}
                };
```

The permutation vector:

```
int[] perm = {2,0,1};
```

The routine:

```
int[][] temprowindex = new int[dimension][1];
double[][] temprowvalue = new double[dimension][1];

for(int i = 0;i<dimension;i++){
     temprowindex[i] = Aindex[perm[i]];
     temprowvalue[i] = Avalue[perm[i]];
}

Avalue = temprowvalue;
Aindex = temprowindex;
```

The most important thing to notice in the routine above is that we only replace the references for the rows into `temprowindex` and `temprowvalue,` we do not however create new array objects and copy the rows from one array object to another. After we have performed the permutation of the rows we initialise the `Aindex` and `Avalue` with the references of `temprowindex` and `temprowvalue` respectively.

Comparing this to a similar operation on CRS, we would create a whole new nonzero structure and copy the rows from the original structure to the new structure according to its permutation vector. This approach, which we do not claim to be the optimum, would be more costly both in time and space then the Sparse Arrays approach illustrated above.

Next we illustrate how to do a permutation of columns on Sparse Arrays:
Input : Matrix $\boldsymbol{A}$ and the permutation vector `perm`

1. $\boldsymbol{B} = \boldsymbol{A}^{\boldsymbol{T}}$
2. Row permutation of $\boldsymbol{B}$ according to its permutation vector `perm`, like explained above.
3. $\boldsymbol{A} = \boldsymbol{B}^{\boldsymbol{T}}$

The cost of performing a transpose on CRS is high both in time and space. This is clearly illustrated in [9, 10]. We have not implemented the transpose routine on Sparse Arrays.

5.6 Insertion and Deletion of Elements on Sparse Arrays

Insertion and deletion of elements of a matrix on Sparse Arrays are clearly more efficient then doing the same operations on CRS. The reason for this is that when we are inserting an element in CRS we have to resize both the index and value array and update the pointer array. This can only be achieved by creating two new arrays with one extra element position then the original arrays. Then we have to copy the original array elements to the new arrays and place the new element in its rightful place. Remember, that we must traverse the whole nonzero structure, and also update the pointer array. When removing an element in CRS, we must also create two new arrays but with a size one less then the original arrays. Again we must copy all the elements from the original arrays to the new arrays, excluding the element we want to remove. We must also update the pointer array.

These two routines show how we can perform these operations on Sparse Arrays.
Contrary to the CRS insertion we now update only the row, were we inserted the value.

The implementation of removing an element from Sparse Arrays:

```
public boolean remove(int i, int j){
     double[] avalue = Avalue[i];
     int[] aindex = Aindex[i];
     boolean isInRow = false;

     for(int ii = 0;ii<aindex.lenght;ii++){
          if(j==aindex[ii])isInRow = true;
     }
     if(isInRow){
          double[] value = new double[avalue.length-1];
          int[] index = new int[avalue.length-1];

          for(int ii = 0;jj=0;ii<aindex.length;ii++){
               if(j!=aindex[ii]){
                    value[jj] = avalue[ii];
                    index[jj] = aindex[ii];
                    jj++;
               }
          }
          Avalue[i] = value;
          Aindex[i] = index;
     }
     return isInRow;
}
```

First we have to test if there is an element in the position given by the parameters `i` and `j`, if there is not, we return `false` from this method.

If there is an element in this position in Sparse Arrays we must create two new arrays (one `int[]` and one `double[]`). The length of these two new arrays is one less then the original arrays.

Next we copy the elements from the original array to the new array excluding the element in the given position. To find that element we must do an if-statement for each element. Then we replace the original arrays with the new arrays in the Sparse Arrays. The original arrays are no longer referenced by the program and are apt to be garbage collected.

If the representation of the arrays was structured before the element was removed, it still would be structured when removing an element with the `remove()` method.

The implementation of an element insertion in Sparse Arrays:

```
public boolean setValue(int i, int j, double value){
     double[][] avalue = Avalue[i];
     int[][] aindex = Aindex[i];
     boolean isInRow = true;
     for(int ii = 0;ii<aindex.length;ii++){
          if(aindex[ii]==j){
               avalue[ii] = value;
               isInRow = false;
          }
     }
     if(isInRow){
          double[] valuenew = new double[aindex.length+1];
          int[] indexnew = new int[aindex.length+1];
          for(int ii = 0;ii<aindex.length;ii++){
               valuenew[ii] = avalue[ii];
               indexnew[ii] = aindex[ii];

          }

          valuenew[valuenew.length-1] = value;
          indexnew[indenew.length-1] = j;

          Avalue[i] = valuenew;

          Aindex[i] = indexnew;
     }
     return test;
}
```

First we must test if an element is already in that position in Sparse Arrays. If it is we only have to overwrite that element with the new element, if not, we must update the row by creating a new position in the structure.

Since we do not intend to uphold any ordered representation of the rows, we can create that position anywhere in the row. In this routine we will for simplicity add the row element at the end of the index and value array. We have to create two new arrays, one `double[]` for storing the values and one `int[]` for storing the indexes. Both these arrays must have a length that is one more then the original array. Then we copy the elements from the original arrays to the new arrays and finally inserting the element. Concerning the memory use in this method, we only create two new array objects and two array objects is apt to be garbage collected. If we had an ordered representation of the elements and we wanted to uphold that representation, we would have to place the new element in its right position. To achieve that, we had to test each index and place the new element in its right position index-wise.

In section 5.9 we will discuss if Sparse Arrays is suitable not only for numerical operations but also for symbolic operations (that means we only work on the nonzero structure of the matrix, not considering the numerical values). Since insertion and deletions are fundamental for symbolic operations we have with the routines above that Sparse Arrays may be more suitable then CRS, CCS or COO for symbolic operations but if Sparse Arrays is the ‘best’ structure for symbolic operations in Java is more uncertain, which needs more investigation.

5.7 Java's Utility Classes

In this section we will take a brief look at some of Java's utility classes and explain why the are not well suited for storing large sparse matrices for numerical operations.

The following classes are part of Java's API package `java.util` and can also be considered to store sparse matrices:

- `java.util.Vector`
- `java.util.ArrayList`
- `java.util.LinkedList`
- `java.util.Stack`

All of these classes have tempting interfaces that can be used for basic linear algebra routines. The reason why they are not suitable is overhead, which are caused because none of them can store primitive types directly. Instead they must be in a wrapper class like this:

```
Vector vector = new Vector(10);
int number = 10;
Integer integer = new Integer(number);
Vector.add(integer);
```

We cannot add the `int` variable directly into the `Vector` object. So when we want to access the elements from the `Vector` object, we must cast it since the `Vector` class only returns `Object`, like this:

```
Integer number = (Integer)vector.get(0);
```

The example for `Vector` would also apply for the three other classes (`Stack`, `ArrayList` and `LinkedList`). If a large ordered collection has elements frequently deleted or inserted throughout its lifetime, a linked list may provide the best performance rather then an array [39]. On the other hand a static (unchanging) collection that needs to be accessed by index, performs better with an underlying implementation which are arrays [39].

`Vector` and `ArrayList` uses Java arrays as their underlying structure while `LinkedList` has underlying implementations based on doubly linked list. `LinkedList` has the worst performance of the four classes. The `Stack` class does not have an interface, which is general enough to be used. `LinkedList` provides better performance when adding and deleting elements towards the middle of the list, if the array-copying overhead of other `List`s is higher than linear access time of the `LinkedList` [78]. `ArrayList` seems to perform better then `Vector` but it is not significant so therefore it can not be compared to the performance of the Java array [39, 54].

Java utility classes is not suitable high performance numerical routines, this is unfortunately because they are part of the Java API, which makes them accessible for all Java applications.

5.8 The use of `System.arraycopy()`

A scenario that appears in `many` routines presented in this chapter is that we have to copy elements from one array to another. This array copy operation can either be achieved by using loops or the `arraycopy()` method of the `System` class [78]. With `arraycopy()` we can copy elements from one array to another. When making this copy, the destination array can be larger, if the destination array is smaller, an `ArrayIndexOutOfBoundsException` will be thrown at run-time. `arraycopy()` is likely to be faster then a for-loop because it is implemented in native code [65]. If we only want to copy the nonzero elements (of numerical type `double[]`, `int[]`, `float[]` etc) from one array to another and the elements that are nonzero are not stored in contiguous order, it is not possible using `arraycopy()` in one method call. Using `arraycopy()` copying a column of a two-dimensional array declaration, like `double[][]` is not possible. First we must take a closer look at how `arraycopy()` works.

The signature of the method is like this:

```
public void arraycopy(Object[] source, int start, Object[]
destination,int start, int length)
```

- The elements that get copied is in an increasing order in the source array.
- The `start` variable index the start position of where we start in the source array.
- The destination array must have the same length as the variable [`length`].
- The two arrays must be of the same type.

This means that we can only copy a piece of one array to another array in an increasing order, and it has to be done in one operation.

For the method `arraycopy()` to be really useful it must be considerable faster then any loop alternatives, see section 5.5.3 and Table 5-3 for the impact of `arraycopy()` versus loops.

Another construction besides `arraycopy()`, which could be very useful is a construction that compress and expands Java arrays, given certain criteria's.

For example:

```
double[] array  = {1.0,0.0,1.0,0.0,1.0,0.0,1.0,0.0};
```

As we can see from this declaration and initialisation it contains a lot of zero values. If we want to compress this array based on the criteria that we only store nonzero values in the result and compressed array, we would obtain an array that only contained the nonzero values. Such a construction is not part of the Java language. For matrix computation such a construction could be very useful, especially going from dense to sparse matrices and vica versa.

The `arraycopy()` method is useful for copying contiguously stored elements from one array to another like we did for the Sparse Arrays on ***AB*** (the SPARSEKIT approach). An important observation for the matrix multiplication routine is that we do not store the elements sorted in each row. None of the routines so far in this chapter we have not demanded that the elements in a row to be sorted by increasing index size so this approach would do fine. But if an operation

(Gaussian Elimination and Cholesky Factorization) demands a sorted structure it would be preferable to do a (for example) matrix multiplication and all the structures involved has and would maintain their sorted order. So if we have stored elements in an accumulator in their real index positions like we do in the Pissanetsky approach (when we merge sparse list of integers and add sparse vectors). What we have is a sparse vector that we shall copy over to a compressed array storing only the nonzero values. If we have had a construction that compressed arrays on certain criteria's we could compress that sparse array and those nonzero elements would be in their sorted order.

5.9 A Look Ahead with Sparse Arrays: Cholesky Factorization

In this section we will take a look ahead with Sparse Arrays. In particular if and how we can perform symbolic operations on the Sparse Arrays structure. That means we model graphs with Sparse Arrays. It is preferable that the reader is familiar with the Cholesky method before reading this section.

Graphs can be used to model symmetric matrices, factorisations and algorithms on non-symmetric matrices, as shown in Figure 5-6. Such as fill paths in Gaussian elimination, strongly connected components in matrix irreducibility, bipartite matrices. Not only do graphs make it easier to understand sparse matrix algorithms but they broaden the area of manipulating sparse matrices using existing graph algorithms and techniques. The nonzero pattern of a sparse matrix also describes a binary relation between unknows. Therefore the nonzero pattern of a sparse matrix of a linear system can be modelled with a graph *G(V,E),* where *n* vertices in *V* represent the *n* unknows and where there is an edge from vertex *i* to vertex *j* when ***A****ij* is nonzero. Thus, when a matrix has a symmetric nonzero pattern, the corresponding graph is undirected.

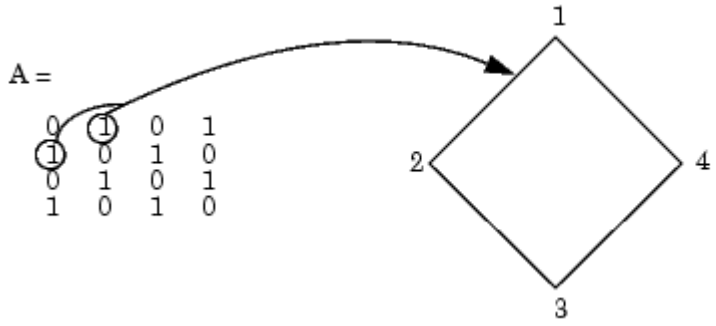

Figure 5-6
Matrix *A* represented as a graph.

The optimal use of Sparse Arrays would be to perform both symbolic and numerical operations on it. The benefit will be that we do not need to transform a matrix into another structure after performing a symbolic operation on it. If this should be preferable, the routines on Sparse Arrays must at least be as efficient as another structure (object-oriented) for symbolic operations. We will in this section show how we can implement a symbolic operation on Sparse Arrays. So far in this chapter we have only implemented numerical operations on Sparse Arrays, if we exclude the insertion and deletion routines. The symbolic operation we implement, is a symbolic factorisation of the Cholesky method routine. We will not give all the details of this implementation.

First we give a formal definition of the Cholesky method:

- Every *n*-by-*n* symmetric positive definitive (SPD) matrix $\boldsymbol{A}$ can be factored as
$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T$$
where $\boldsymbol{L}$ is a lower triangular matrix with positive diagonal elements.

- $\boldsymbol{L}$ is called the Cholesky factor.
- Solution to the system $\boldsymbol{Ax} = \boldsymbol{b}$ is obtained by solving
$$\boldsymbol{Ly} = \boldsymbol{b} \quad \text{(forward substitution)}$$
$$\boldsymbol{L}^T\boldsymbol{x} = \boldsymbol{b} \quad \text{(back substitution)}$$

- Cholesky factorisation algorithm:
  - Variant of Gaussian elimination.
  - Takes advantage of symmetry to reduce work and storage about half.
  - Consists of a triple nested loop, so there are 3! ways of arranging it.

We implemented such a solution for solving $\boldsymbol{Ax} = \boldsymbol{b}$ in Java. We used self defined classes and Java's API classes to create a object oriented graph structure. We used `java.util.LinkedList` to model an adjacency list structure for each vertex.

We performed the following routines:

1. Minimum Degree Ordering, performed on matrix $\boldsymbol{A}$ to reduce fill.
2. Symbolic Factorization, finds the nonzero pattern of $\boldsymbol{L}$.
3. Numerical Factorization, performs the numerical computations of the elements in $\boldsymbol{L}$.
4. Symmetric Solver, solves the system $\boldsymbol{L}\boldsymbol{L}^T\boldsymbol{x} = \boldsymbol{b}$.

We will not comment on how we performed the Minimum Degree Ordering, Numerical Factorisation or Symmetric Solver routine. Instead we focus on the Symbolic Factorization routine.

We choose to focus on a Row-Cholesky since Sparse Arrays is a row-oriented data structure:
- The row counter is in the outer loop. The inner loops solve a triangular system for each new row in terms of the previously computed rows.

The object-oriented graph structure we designed does not claim to be the optimal of performing the symbolic factorization. Instead it is meant as a starting point for creating an object-oriented design to perform high performance symbolic operations. We have not illustrated the implementation of this structure. We measured the routines against each other. The symbolic factorization for Sparse Arrays was clearly more efficient then the symbolic factorization routine implemented using an object-oriented design. This comparison did not tell us much, because we had not optimized the symbolic factorization routine using the object-oriented design.

The implementation of the symbolic factorization routine using Sparse Arrays:

The main idea behind this routine is to find those elements that are not present in each row that will contain numerical results when we perform the numerical factorization. We must therefore locate those elements for each row and expand the array objects for each new element.

We construct the ***L*** matrix, see 5-2, by finding the elements of ***L*** by the following

$$Lij = \left[Aij - \sum_{k=1}^{j-1} LikLjk \right] / Lij$$

$$j = 1,\ldots,i\text{-}1 \qquad (5\text{-}2)$$

and

$$Lii = \left[Aii - \sum_{k=1}^{i-1} Lik \right]^{1/2} \qquad (5\text{-}3)$$

The algorithm for *(5-2)* and *(5-3)* is like the following:

```
do i = 1,N
        do j = 1,i
                do k = 1,j-1
                     A(i,j) = A(i,j) – A(i,k)*A(j,k)
                if(j<i) A(i,j) = A(i,j)/A(j,j)
                else    A(i,i) = sqrt(A(i,i))
```

The algorithm above and *(5-2)* and *(5-3)* is for a numerical factorisation on a dense matrix, but we use this procedure to find those elements that is used in a sparse numerical factorisation for storing numerical results.

A short description of that procedure is:

- For each row and for each element in that row, if that element is present we do not need to find out if that element is used to store an numerical result,when performing the numerical factorisation, but if it is not present we must find out if that element is used. If we are going to find out if element ***L****ij* is going to be used, we find the row *j* and for each element in that row we check if it has the same index, the index size must be less then *j*, as an element in row *i*. If that is the case we must insert an element in position {*i,j}* in the sparse matrix. It is important to notice that we only need to find out if we have one element in row *i* and *j* with the same column index, this to minimize subscript searching that means to avoid unnecessarily searching for elements in the matrix.

For the implementation of a symbolic operation on Sparse Arrays see Appendix C [The classes `SparseArrays` and the methods `symbolicFactorisation()` and `jIsInbi()`]

The reason for discussing a non-optimal symbolic operation and a non-optimal graph implementation is to sketch a beginning for designing object-oriented graphs in Java for matrix computation.

The advantages of Sparse Arrays to implement graphs in Java may be:

- Small overhead accessing numerical elements
- Unnecessary to transform the structure from a symbolic structure to a numerical structure.

The problem with Sparse Arrays is:

- That it favours row-oriented routines.
- That it is overhead updating the arrays.
- That we cannot give the array objects other features, methods and instance variables, as we can with self defined classes or API classes.

We could use the Sparse Matrix concept if adding features to the rows were essential for designing a graph implementation in Java.

We will certainly not recommend the use of Sparse Arrays as a structure for symbolic operations. Since symbolic operations usually means frequently deleting and insertion of elements and that means frequently updating of a Java array as we have shown in this chapter demands extensive copying of elements from one array to another, which is very time consuming. But we have shown how we can do it for one symbolic operation and the routine is very elegant if not (if it where eventually compared to other implementations) very efficient.

5.10 Summary, Discussions and Conclusions

Currently there is no packages (in the scale of JAMA and Jampack) implemented in Java for numerical linear algebra on sparse matrices. This may change in the near future, therefore it is interesting to implement several different storage schemes in Java for sparse matrices, for the possibility to unveil their usability when it comes to performance and memory. There are numerous storage schemes for sparse matrices. Since we only deals with general matrices we will only deal with general storage schemes like CRS, CCS and COO.

Implementing a matrix-vector product and vector-matrix product on CRS, CCS and COO was rather straightforward. The exception was the COO's format lack of a pointer array caused unwanted overhead, by explicitly testing each element for what row or column it was in. This made us discard COO as a high performance storage scheme. The difference between CRS and CCS was insignificant, so there was no point choosing between them on the basis of performance.

The problem implementing linear algebra routines on CRS, CCS and COO was when we did a matrix multiplication. Since we do not know the nonzero structure of the resulting matrix we must dynamically build the resulting matrix. Since Java arrays have no 'vector' capabilities, such operations leads to unwanted overhead.

We introduced Sparse Arrays as a consequence of doing a matrix multiplication, where the two input matrices where on CRS format and the output on CRS format. We implemented two approaches for matrix multiplication on CRS, the guesstimate approach was more efficient then the symbolic approach, if the guesstimate approach was completed at first try. We further concluded that even if the guesstimate is a very efficient routine it is also unpredictable routine that may fail multiple times. The symbolic approach was unacceptable slow compared to the guesstimate approach. We then implemented two matrix multiplication routines with Sparse Arrays. Here we used the SPARSEKIT approach and the Pissanetsky approach. It was very clear from the time testing of these routines that the SPARSEKIT approach was more efficient then Pissanetsky approach. The SPARSEKIT approach was for the test matrices just an average of 1.4 times slower then the guesstimate approach on CRS (completed at the first try). The drawback for the SPARSEKIT approach is the memory use and that it creates two new arrays objects for resulting row it creates, this can become a performance issue if the garbage collector kicks in, see Chapter 2, section 2.3. So even if it seems that we have a routine that is completed at first run, is very fast it and is considered reliable (compared to the guesstimate approach) may not give us the expected performance for all cases.

It may seem unnecessarily to present the Pissanetsky approach at all, since the SPARSEKIT approach in all the routines we have presented has a superior performance. But the Pissanetsky approach does something that can be very important for some routines like the Cholesky method and Gaussian Elimination and that is to keep an ordered representation, that means the result of the merging and addition is stored in accumulators in their real index positions. So the only thing that is missing as an efficient method for retrieving those elements without traversing the whole accumulator into the compressed array in ordered representation.

With Sparse Arrays we could manipulate the rows of the matrix without needing to update or traverse the rest of the structure, unlike the structures of CRS, CCS, COOR and COOC. Sparse Arrays use more memory for each nonzero element then CRS and CCS. We have one array object in Sparse Arrays then in CRS and CCS. The goal in presenting matrix multiplication**,** matrix addition**,** removing elements, inserting elements and row-permutation

was not to present the optimum algorithms for these routines on Sparse Arrays. We rather wanted to show the benefit of using Sparse Arrays instead of CRS and to illustrate the effect of having the possibility to manipulate only the rows of the structure without needing to update the rest of the structure. We also implemented a matrix-vector product and a vector-matrix product routines on Sparse Arrays. There where two reasons for doing this, to see if we got a similar implementation to that of CRS and to see if they had a similar performance to CRS. The implementation was very similar and Sparse Arrays was in fact slightly more efficient then CRS on the $\boldsymbol{Ab}$ and $\boldsymbol{b^TA}$ routines. This gave us an indication that Sparse Arrays could replace CRS for many routines without significant loss of performance. We still do not know if Sparse Arrays could replace CRS for *all* routines, this is an open question, which is worth further researching.

If we compare Sparse Arrays to the Sparse Matrix concept on the basis of performance, we can see from Table 5-1 that Sparse Arrays was slightly more efficient then the Sparse Matrix concept on the $\boldsymbol{Ab}$ routine. We focused on two ways of implementing routines using the Sparse Matrix concept. We could make it as similar as possible to Sparse Arrays, that is all numerical methods is placed in the `SparseMatrix` class. The problem with this approach is the extra object `Rows` which now only adds overhead and memory. Next we use the `Rows` class as it should be used consider an object-oriented approach, that is all methods that works explicitly on the instances of the `Rows` object are placed in the `Rows` class. The problem with this is that it is even slower. If we compare them on memory, we have one extra object layer for each row (index and value row) for the Sparse Matrix concept while for Sparse Arrays we have one extra array object for storing the references to the index or value array. If we have a matrix with the dimension *mxn,* we have *m* objects for the Sparse Matrix concept compared to *one* array object for Sparse Arrays. The memory demand may be greater for the Sparse Matrix concept compared to Sparse Arrays. Even if the performance between Sparse Arrays and the Sparse Matrix concept was insignificant for the $\boldsymbol{Ab}$ routine, the difference between Sparse Arrays and the Sparse Matrix concept could be more significant comparing them on more dynamic routines as a matrix multiplication routine. For the Sparse Matrix concept we must create one extra object for each row we create in the resulting matrix, this object creation could have a significant impact on performance. We can have more flexibility with a row object like the `Rows` object, then we have with Sparse Arrays. This would be a reason for choosing the Sparse Matrix concept. Sparse Arrays would be the first choice if we only look at performance and the use of memory.

Last in this chapter we discussed how we could do a symbolic factorization on Sparse Arrays and showed an implementation of this, see Appendix C. We used a row-oriented symbolic factorization approach, since Sparse Arrays as we have declared in this chapter is row-oriented. Since Sparse Arrays is first introduced in this thesis we cannot state that the implementation is the optimal (considering performance or memory). The main goal implementing a symbolic factorization on Sparse Arrays was to show an elegant implementation and give the reader the possibility to develop an even “better” implementation. After the symbolic factorization we could do the numerical factorization directly with Sparse Arrays. Another subject, which is important, is the transformation between structures that we perform symbolic operations on and structures that we perform numerical operations on. The efficiency must be upheld using structures for both operations. The Sparse Arrays structure may be seen as a hybrid structure since we can use for both operations. Sparse Arrays for symbolic operations is still an open question, but based on the result in this chapter it is worth further research.
We also compared the symbolic factorization on Sparse Arrays with the symbolic factorization on the object-oriented graph representation. In the examples and for the test matrices, Sparse

Arrays was more efficient. This did not however tell us anything definitive, because we had not optimized the object-oriented graph representation or the symbolic routine implemented on that structure. We did not optimize this structure or in another way try to develop it any further. The main purpose of such a development would be to describe how to develop object-oriented numerical classes for high performance. We did not start on that project, because it seemed difficult with too many pitfalls. With the introduction of JITs and HotSpots, we might be able to create an object-oriented data structures for the symbolic factorisation algorithm and other symbolic operations, which will have a satisfying performance. This is a hypothesis that we did not test in this thesis.

Since there are no packages implemented in Java for sparse matrix computations, we cannot compare Sparse Arrays to other structures that store sparse matrices using Java arrays or other constructions. In the introduction we stated that the numerical community was searching for alternative constructions for storing matrices then Java's native arrays. We have in this chapter showed that using the dynamic features of Java arrays we could design more efficient routines on Sparse Arrays then on CRS. Therefore considering constructions (data structures) like Sparse Arrays or the Sparse Matrix Concept that uses Java's native arrays for implementing a sparse matrix, package would be natural, and it also show the relevance of Java's native arrays as an important data structure.

Concluding Remarks

In Chapter 3, 4 and 5 we have tried to unveil the features of Java arrays which makes it suitable for matrix computations. We have also considered those features that make Java arrays not suitable for matrix computations. Speed in all the routines we have presented have been the main factor for classifying them as 'good'. We want of course that every numerical routines to be as fast as possible for every input, but the memory use is also important both the static size of the structures and the extra use of accumulators we use for some of our routines. If the trade-off between time and space becomes to large, that is we have to use significant amount of memory to execute a routine, no matter how fast it is, the routines will not be suitable for all tasks because it may be runned under circumstances where we have limited of free memory. It is important to keep this in mind when evaluating the routines we have presented, especially in Chapter 5.

In Chapter 3 and 4 we evaluated Java arrays for storing dense matrices. It is clear that the difference in the time for row traversing versus column traversing is unacceptable. The main reason for that, is if it should be used in a highly optimised numerical package, the implementers must take the row-oriented structure of a two-dimensional Java array for each routine to achieve the optimal performance. To abandon the `Object[i][j]` notation for arrays to a more object-oriented approach like `Arrays.getValue(i,j)` might be an opportunity, and the disadvantages we have with Java arrays must be eliminated.

When we inspected the other routines in JAMA and Jampack they took the row-wise layout into consideration, but only the `times()` routine, in JAMA's package, accessed `double[]` instead of `double[][]` in its innermost for-loop, how much gain the routines would achieve by just traversing `double[]`, have not be measured. In Chapter 3 we achieved approximately an average of 30% better performance by traversing `double[]` instead of `double[][],` see section 3.7, so taking that into consideration the JAMA package (or the `Matrix` class) have the potential to achieve a better performance. Also the Jampack routines took the row-wise layout into consideration and it traverses `double[][]` instead of `double[]` and we conclude that Jampack has a potensial for a better performance. Another observation of these packages is that those routines we considered in the Jampack package access its arrays through a reference, as discussed in Chapter 2 this could decrease performance compared to breaking encapsulation. But that also depends on which compiler we use (a JIT may eliminate this issue), this would also apply to those routines in the JAMA package that access its elements from a reference.

Both the JAMA and Jampack packages are excellent implemented, has a very high degree of usabillity and they are both higly recommendable for matrix computations.

There is no package in Java for sparse matrix computation, we have therefore discussed several data structures for storing sparse matrices in Java. It is straightforward to implement and use traditionally storage schemes like CRS, CCS and COO in Java, it does not differ from the implementation of these storage scheme in FORTRAN [77] or C++ [82]. The ineffciency and unpredictability implementing matrix multiplications on these structures demanded a more suitable storage scheme for that operation in Java, and the solution is Sparse Arrays. What we have done in Chapter 5 could be seen as a pre-analyses of a sparse package implemented in Java. There is of course other issues concerning sparse packages in Java that also will relate to the storage schemes other then the issues we have mentioned, but those that we have considered

will also be considered in a ’real’ package. So in short we have outlined a propolsal to use Sparse Arrays as data structure for storing sparse matrices in Java for numerical operations.

We have not discussed if Java is suited for numerical computing, instead we have focused on how we could utilise the langauge to implement effcient routines. This work have given ideas of how some constructions in Java limite ’natural’ development and ideas of their improvements or replacements.

APPENDIX A The Benchmarks Timings

1.0 Introduction

In this appendix we have gathered all the benchmarks result of Chapter 3. We reference these tables from Chapter 3 and 4. All the benchmark times are in milliseconds.

1.1 JAMA's `A.times(B)` on Ab versus b^TA

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	Ab	b^TA	Ab	b^TA	**Ab**	b^TA
80	0	0	0	0	0	0
115	0	0	0	0	0	0
138	0	0	0	0	0	0
240	0	0	0	0	0	1
468	0	0	0	0	2	2
663	0	0	0	0	5	4
765	0	0	0	0	6	6
817	0	0	0	0	7	7
900	0	0	0	0	9	9
1000	0	0	0	0	10	11
1374	0	0	0	0	19	19
1500	0	0	0	0	23	24
2000	0	0	0	0	48	210

Table 1-0
The loop-overhead for the benchmarks, that is the benchmarks without the loop-body.

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	Ab	b^TA	Ab	b^TA	Ab	b^TA
80	0	1	0	0	0	0
115	0	1	0	0	1	0
138	0	12	0	0	1	1
240	0	5	0	0	3	4
468	0	51	0	1	14	14
663	0	124	0	0	33	27
765	0	194	0	0	37	37
817	0	223	0	2	42	41
900	0	279	0	0	53	51
1000	0	382	0	1	62	62
1374	0	1055	0	2	140	142
1500	0	1144	0	3	167	167
2000	0	11982	0	4	327	543

Table 1-1
The benchmark timings.

1.2 The pure row-oriented `A.times(B)` on Ab versus b^TA

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	Ab	b^TA	Ab	b^TA	Ab	b^TA
80	0	0	0	0	0	0
115	0	0	0	0	1	0
138	0	0	1	0	1	0
240	0	0	1	0	2	0
468	0	0	3	0	11	0
663	0	0	5	0	24	1
765	0	0	6	0	28	2
817	0	0	7	0	33	2
900	0	0	8	0	52	3
1000	0	0	14	0	62	4
1374	0	0	22	0	105	6
1500	0	0	27	0	120	8
2000	0	0	40	0	194	14

Table 1-2
The loop-overhead for the benchmarks, that is the benchmarks without the loop-body.

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	Ab	b^TA	Ab	b^TA	Ab	b^TA
80	0	0	0	0	0	0
115	0	0	0	0	1	1
138	0	0	1	0	2	1
240	0	0	2	0	7	2
468	0	0	8	0	24	10
663	0	0	16	0	48	19
765	0	0	26	0	64	25
817	0	0	26	0	74	28
900	0	0	27	0	86	33
1000	0	0	45	0	107	52
1374	0	0	67	0	196	117
1500	0	0	97	0	253	140
2000	0	0	202	0	417	236

Table 1-3
The benchmark timings.

1.3 JAMA's `A.times(B)` versus the pure row-oriented on *Ab*

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	JAMA	ROW	JAMA	ROW	JAMA	ROW
80	0	0	0	0	0	0
115	0	0	0	0	0	1
138	0	0	0	1	0	1
240	0	0	0	1	0	2
468	0	0	0	3	2	11
663	0	0	0	5	5	24
765	0	0	0	6	6	28
817	0	0	0	7	7	33
900	0	0	0	8	9	52
1000	0	0	0	14	10	62
1374	0	0	0	22	19	105
1500	0	0	0	27	23	120
2000	0	0	0	40	48	194

Table 1-4
The loop-overhead for the benchmarks, that is the benchmarks without the loop-body.

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	JAMA	ROW	JAMA	ROW	JAMA	ROW
80	0	0	0	0	0	0
115	0	0	0	0	1	1
138	0	0	0	1	1	2
240	0	0	0	2	3	7
468	0	0	0	8	14	24
663	0	0	0	16	33	48
765	0	0	0	26	37	64
817	0	0	0	26	42	74
900	0	0	0	27	53	86
1000	0	0	0	45	62	107
1374	0	0	0	67	140	196
1500	0	0	0	97	167	253
2000	0	0	0	202	327	417

Table 1-5
The benchmark timings.

1.4 JAMA's `A.times(B)` b^TA on versus the pure row-oriented on Ab

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	JAMA	ROW	JAMA	ROW	JAMA	ROW
80	0	0	0	0	0	0
115	0	0	0	0	0	1
138	0	0	0	0	0	1
240	0	0	0	0	1	2
468	0	0	0	0	2	11
663	0	0	0	0	4	24
765	0	0	0	0	6	28
817	0	0	0	0	7	33
900	0	0	0	0	9	52
1000	0	0	0	0	11	62
1374	0	0	0	0	19	105
1500	0	0	0	0	24	120
2000	0	0	0	0	210	194

Table 1-6
The loop-overhead for the benchmarks, that is the benchmarks without the loop-body.

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	JAMA	ROW	JAMA	ROW	JAMA	ROW
80	1	0	0	0	0	0
115	1	0	0	0	0	1
138	12	0	0	1	1	2
240	5	0	0	2	4	7
468	51	0	1	8	14	24
663	124	0	0	16	27	48
765	194	0	0	26	37	64
817	223	0	2	26	41	74
900	279	0	0	27	51	86
1000	382	0	1	45	62	107
1374	1055	0	2	67	142	196
1500	1144	0	3	97	167	253
2000	11982	0	4	202	543	417

Table 1-7
The benchmark timings.

1.5 JAMA's `A.times(B)` on Ab versus the pure row-oriented on b^TA

	Benchmark X		Benchmark Y		Benchmark Z	
Matrix Dimension	JAMA	ROW	JAMA	ROW	JAMA	ROW
80	0	0	0	0	0	0
115	0	0	0	0	0	0
138	0	0	0	0	0	0
240	0	0	0	0	0	0
468	0	0	0	0	2	0
663	0	0	0	0	5	1
765	0	0	0	0	6	2
817	0	0	0	0	7	2
900	0	0	0	0	9	3
1000	0	0	0	0	10	4
1374	0	0	0	0	19	6
1500	0	0	0	0	23	8
2000	0	0	0	0	48	14

Table 1-8
The loop-overhead for the benchmarks, that is the benchmarks without the loop-body.

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	JAMA	ROW	JAMA	ROW	JAMA	ROW
80	0	0	0	0	0	0
115	0	0	0	0	1	1
138	0	0	0	0	1	1
240	0	0	0	0	3	2
468	0	0	0	0	14	10
663	0	0	0	0	33	19
765	0	0	0	0	37	25
817	0	0	0	0	42	28
900	0	0	0	0	53	33
1000	0	0	0	0	62	52
1374	0	0	0	0	140	117
1500	0	0	0	0	167	140
2000	0	0	0	0	327	236

Table 1-9
The benchmark timings.

1.6 JAMA's `A.times(B)` on $b^T A$ versus the pure row-oriented on $b^T A$

	Benchmark X		Benchmark Y		Benchmark Z	
Matrix Dimension	JAMA	ROW	JAMA	ROW	JAMA	ROW
80	0	0	0	0	0	0
115	0	0	0	0	0	0
138	0	0	0	0	0	0
240	0	0	0	0	1	0
468	0	0	0	0	2	0
663	0	0	0	0	4	1
765	0	0	0	0	6	2
817	0	0	0	0	7	2
900	0	0	0	0	9	3
1000	0	0	0	0	11	4
1374	0	0	0	0	19	6
1500	0	0	0	0	24	8
2000	0	0	0	0	210	14

Table 1-10
The loop-overhead for the benchmarks, that is the benchmarks without the loop-body.

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	JAMA	ROW	JAMA	ROW	JAMA	ROW
80	1	0	0	0	0	0
115	1	0	0	0	0	1
138	12	0	0	0	1	1
240	5	0	0	0	4	2
468	51	0	1	0	14	10
663	124	0	0	0	27	19
765	194	0	0	0	37	25
817	223	0	2	0	41	28
900	279	0	0	0	51	33
1000	382	0	1	0	62	52
1374	1055	0	2	0	142	117
1500	1144	0	3	0	167	140
2000	11982	0	4	0	543	236

Table 1-11
The benchmark timing.

1.6 JAMA's `A.times(B)` on *AB* versus the pure row-oriented on *AB* running JDK 1.2 (testplatform)

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	JAMA	ROW	JAMA	ROW	JAMA	ROW
80	0	0	0	0	6	2
115	0	0	1	0	17	5
138	0	0	1	0	30	10
240	0	0	0	0	148	54
468	0	0	3	2	1060	365
663	0	0	5	5	3006	1018
765	0	0	7	6	4606	1560
817	0	0	6	6	6420	1894
900	0	0	9	9	7621	2554
1000	0	0	10	10	10286	3464

Table 1-12
The loop-overhead for the benchmarks, that is the benchmarks without the loop-body.

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	JAMA	ROW	JAMA	ROW	JAMA	ROW
80	0	0	0	1	34	22
115	1	0	1	1	96	64
138	2	0	1	1	167	113
240	9	0	8	2	861	579
468	51	0	17	7	6357	4307
663	127	0	33	16	18069	12117
765	181	0	49	22	27837	18693
817	225	0	52	34	33834	22692
900	277	0	62	32	45340	30276
1000	400	0	86	35	62190	41498

Table 1-13
The benchmark timings.

1.8 JAMA's `A.times(B)` on *AB* versus the pure row-oriented on *AB* running JDK 1.3 (testplatform)

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	JAMA	ROW	JAMA	ROW	JAMA	ROW
80	0	0	2	2	18	18
115	0	0	4	15	31	24
138	0	0	12	12	43	30
240	0	0	13	12	161	114
468	0	0	20	18	1082	734
663	0	0	31	23	3033	2049
765	0	0	36	27	4707	3123
817	0	0	41	29	5651	3817
900	0	0	48	34	7583	5069
1000	0	0	55	41	10423	6962

Table 1-14
The loop-overhead for the benchmarks, that is the benchmarks without the loop-body.

Matrix Dimension	Benchmark X		Benchmark Y		Benchmark Z	
	JAMA	ROW	JAMA	ROW	JAMA	ROW
80	6	0	8	7	64	69
115	15	0	16	13	131	155
138	23	0	28	20	206	240
240	34	0	34	25	952	1140
468	84	0	52	34	8158	11156
663	179	0	86	52	22745	23159
765	236	0	103	61	29486	35720
817	270	1	118	75	36285	43397
900	350	1	136	85	47959	58208
1000	461	0	177	85	66314	79610

Table 1-15
The benchmark timings.

Appendix B CRS, CCS, COOC and COOR in Java

In this Appendix we show the classes for implementing the CRS (`MatrixCRS`), CCS (`MatrixCCS`), COOR (`MatrixCOOR`) and COOC (`MatrixCOOC`) structures that we present in Chapter 5.

Remarks. These classes stores only square matrices ($m=n$). The routines were written early in the work on this thesis and may therefore suffer from that on the basis of consistent notation, that means comments and the code itself. But the overall meaning of the classes and the implemented routines should be rather clear for the reader that has some experience with Java code and these operations.

We have chosen not to comment the code itself, because each routine where the implementation is essential is explained in the text (in the chapter where we represent the routine). Another reason for not comment the code itself is because the routines are 'small' and rather self-explanatory.

1. The class `MatrixCRS`

```
/** A class for a sparse matrix on compressed row format
  */
public class MatrixCRS{
      private double[] value;
      private int columnindex[];
      private int[] rowpointer;
      private int nonzeros;
      private int dimension;
      public MatrixCRS(double[] value,int[] columnindex,int[] rowpointer){
            this.value = value;
            this.columnindex = columnindex;
            this.rowpointer = rowpointer;
            nonzeros = value.length;
            dimension = rowpointer.length-1;
      }


      /**Method for returning the array of values
      */
      public double[] getValueArray(){return value;}

      /**Method for returning the column index array
      */
      public int[] getColumnArray(){return columnindex;}


      /**Method for returning the array of pointers
      */
      public int[] getPointerArray(){return rowpointer;}

      /**Method for returning a column index value
      */
      public int getColumnindex(int i){return columnindex[i];}

      /**Method for returning a pointer value
      */
```

```
public int getRowpointer(int i){return rowpointer[i];}

/**Method for returning the dimension of the matrix
*/
public int getDimension(){return dimension;}

/**Method for returning the number of nonzero values of the *matrix
*/
public int getNonzeroSize(){return size;}

/**Method for returning a value
*/
public double getValue(int i){return value[i];}

/**Method for multiplying two matrices on CRS format.
* This is the guesstimate approach
*/
public MatrixCRS times(MatrixCRS B,int arbitrary,int dim1){

      int arbitrarysize = size*arbitrary;
      double[] C = new double[arbitrarysize];
      int[] Cindex = new int[arbitrarysize];
      int Cpointer[] = new int[dim1+1];
      Cpointer[0]=0;
      double scalar = 0.0;
      int job = 1; int startrowa=0; int startrowb=0;
      int len = -1; int ierr = 0;
      int jj=0; int jcol=0; int jposition=0;
      int stoprowb=0;int stoprowa = 0;
      int temp[] = new int[dim1];
      boolean values = true;
      double[] valuesB = B.getValueArray();
      int[] Bcolindex = B.getColumnArray();
      int[] Bpointer = B.getPointerArray();
      for(int i=0;i<dimension;i++){temp[i]=-1;}
      for(int i = 0;i<dimension;i++){
            startrowa = rowpointer[i];
            stoprowa = rowpointer[i+1]-1;
            for(;startrowa<=stoprowa;startrowa++){
            scalar = value[startrowa];
            jj = columnindex[startrowa];
            startrowb = Bpointer[jj];
            stoprowb = Bpointer[jj+1]-1;
            for(;startrowb<=stoprowb;startrowb++){
                  jcol=Bcolindex[startrowb];
                  jposition = temp[jcol];
                  if(jposition == -1){
                        len++;
                        Cindex[len]=jcol;
                        temp[jcol]=len;
                        C[len]=scalar*valuesB[startrowb];
                  }
                  else{
                  C[jposition]+=scalar*valuesB[startrowb];
                  }
            }

          }
          for(int k=Cpointer[i];k<=len;k++){
            temp[Cindex[k]]=-1;
          }
```

```
            Cpointer[i+1]=len+1;
      }

      return arbitraryToCompressed(C,Cindex,Cpointer,len);
}


/** Method for copying this arbitrary sized structure over to a
*compressed nonzero structure
*/
public MatrixCRS arbitraryToCompressed(double[] C,int[] Cindex,int[]
Cpointer, int len){
      int[] CindexA = new int[len];
      double[] CA = new double[len];
      System.arraycopy(C, 0, CA, 0, len);
      System.arraycopy(Cindex, 0, CindexA, 0, len);
      return new MatrixCRS(CA, CindexA, Cpointer);
}


/**Method for multiplying two matrices on CRS format.
* This is the symbolic approach
*/
public MatrixCRS timessymbolic(MatrixCRS B,int dimension){
      int[] Cindex = new int[dimension];
      int Cpointer[] = new int[dimension+1];
      Cpointer[0]=0;
      int startrowa=0; int startrowb=0;
      int stoprowb=0;int stoprowa = 0;
      int len = -1;
      int lentemp = -1;
      int jj=0; int jcol=0; int jposition=0;
      boolean test = true;
      int temp[] = new int[dimension];
      int[] indexTemp = null;
      int[] Bindex = B.getColumnArray();
      int[] Bpointer = B.getPointerArray();
      for(int i=0;i<dimension;i++){temp[i]=-1;}
      for(int i = 0;i<dimension;i++){
            startrowa = rowpointer[i];
            stoprowa = rowpointer[i+1]-1;
            for(;startrowa<=stoprowa;startrowa++){
                        jj = columnindex[startrowa];
                        startrowb = Bpointer[jj];
                        stoprowb = Bpointer[jj+1]-1;
                        for(;startrowb<=stoprowb;startrowb++){
                              jcol=Bindex[startrowb];
                              jposition = temp[jcol];
                              if(jposition == -1){
                                    len++;
                                    lentemp++;
                                    Cindex[lentemp]=jcol;
                                    temp[jcol]=len;
                              }
                        }
            }
            for(int k = 0;k<=lentemp;k++){temp[Cindex[k]]=-1;}
            Cpointer[i+1]=len+1;

            if(test){
                  indexTemp = new int[len+1];
```

```
                  System.arraycopy(Cindex, 0, indexTemp, 0, len+1);
                        test = false;
                  }
                  else{
                        if(lentemp>-1){
                        int[] a = new int[len+1];
      System.arraycopy(indexTemp, 0, a, 0,indexTemp.length);
      System.arraycopy(Cindex, 0, a, indexTemp.length, lentemp+1);
                        indexTemp = a;
                        }
                        }
                  }
                  lentemp = -1;
                  Cindex = new int[dimension];
            }
            return numMatrixCRS(indexTemp, Cpointer, B);
      }


      /**Method for multiplying two matrices on CRS format.
      * This is the numerical approach
      */
      public MatrixCRS numMatrixCRS(int[] Cindex, int[] Cpointer, MatrixCRS
B){
                  double[] C = new double[Cindex.length];
                  double scalar = 0.0;
                  int startrowa=0; int startrowb=0;
                  int len = -1;
                  int jj=0; int jcol=0; int jposition=0;
                  int stoprowb=0;int stoprowa = 0;
                  int temp[] = new int[dimension];
                  double[] valuesB = B.getValueArray();
                  int[] Bindex = B.getColumnArray();
                  int[] Bpointer = B.getPointerArray();
                  for(int i=0;i<dimension;i++){temp[i]=-1;}
                  for(int i = 0;i<dimension;i++){
                        startrowa = rowpointer[i];
                        stoprowa = rowpointer[i+1]-1;
                        for(;startrowa<=stoprowa;startrowa++){
                              scalar = value[startrowa];
                              jj = columnindex[startrowa];
                              startrowb = Bpointer[jj];
                              stoprowb = Bpointer[jj+1]-1;
                              for(;startrowb<=stoprowb;startrowb++){
                                    jcol=Bindex[startrowb];
                                    jposition = temp[jcol];
                                    if(jposition == -1){
                                          len++;
                                          temp[jcol]=len;
                                    C[len]=scalar*valuesB[startrowb];
                                    }
                                    else{
            C[jposition]+=scalar*valuesB[startrowb];
                                    }
                              }
                        }
                     for(int k=Cpointer[i];k<=len;k++){temp[Cindex[k]]=-1;}

            }
            return new MatrixCRS(C, Cindex, Cpointer);
}
```

```
        /**Method for multiplying a vector with a matrix
        */
        public Vector vectormatrix(Vector v){
            int startrowa = 0,stoprowa = 0;
            double[] value = v.getValueArray();
            Y = new double[value.length];
            for(int i = 0;i<rowpointer.length-1;i++){
                  startrowa = rowpointer[i];
                  stoprowa = rowpointer[i+1]-1;
                  for(;startrowa<=stoprowa;startrowa++){
            Y[i]+=value[startrowa]*value[columnindex[startrowa]];
                  }
            }
            return new Vector(Y,Y.length);
      }

      /**Method for multiplying a matrix with a vector
        */
      public Vector matrixvector(Vector v){
            int startrowa = 0,stoprowa = 0,i = 0;
            double z = 0.0;
            double[] valuea = v.getValueArray();
            Y = new double[value.length];
            for(int j = 0;j<rowpointer.length-1;j++){
                  startrowa = rowpointer[j];
                  stoprowa = rowpointer[j+1]-1;
                  for(;startrowa<=stoprowa;stoprowa++){
            z += valuea[columnindex[startrowa]]*value[startrowa];
                  }
                  Y[j] = z;
            }
            return new Vector(Y,Y.length);
      }

}
```

2. The class `MatrixCCS`

```
/** A class for a sparse matrix on compressed coloumn format
  */
public class MatrixCCS{

      private double[] value;
      private int[] rowindex;
      private int[] columnpointer;
      private int nonzeros;
      private int dimension;

      public MatrixCCS(double[] value,int[] rowindex,int[] columnpointer){
            this.value = value;
            this.rowindex = rowindex;
            this.columnpointer = columnpointer;
            nonzeros = value.length;
            dimension = columnpointer.length-1;
      }
```

```
/**Method for returning the dimension of the matrix
*/
public int getDimension(){return dimension;}

/**Method for returning the number of nonzero values of
* the matrix
*/
public int getNonzeroSize(){return nonzeros;};

/**Method for returning the array of values
*/
public double[] getValueArray(){return value;}

/**Method for returning the row index array
*/
public int[] getRowArray(){return rowindex;}

/**Method for returning the array of pointers
*/
public int[] getPointerArray(){return columnpointer;}

/**Method for returning a value
*/
public double getValue(int i){return value[i];}

/**Method for returning a index
*/
public int getRowindex(int i){return rowindex[i];}

/**Method for returning a pointer
*/
public int getColumnpointer(int i){return columnpointer[i];}



/**Method for a matrix-vector product
*/
public Vector matrixvector(Vector v){
      int k1,k2,i;
      double[] valuea = v.getValueArray();
      Y = new double[valuea.length];
      for(int j = 0; j<columnpointer.length-1;j++){
            startrow = columnpointer[j];
            stoprow = columnpointer[j+1]-1;
            for(;startrow<=stoprow;startrow++){
            Y[rowindex[startrow]] += valuea[j]*value[startrow];
            }
      }
      return new Vector(Y,Y.length);
}

  /**A method for vector-matrix product
    */
    public Vector vectormatrix(Vector v){
      int startrow = 0;
      int stoprow = 0;
      double z = 0;
      double[] valuea = v.getValueArray();
      double[] c = new double[valuea.length];
      for(int i = 0;i<columnpointer.length-1;i++){
            startrow=columnpointer[i];
```

```
                stoprow =columnpointer[i+1]-1;
                for(;startrow<=stoprow;startrow++){
                z += valuea[rowindex[startrow]]*value[startrow];
                }
                c[i]=z;
                z=0;
          }
          return new Vector(c,c.length);
        }

        /**Method for multiplyiong two matrices on CCS format.
        */
        public MatrixCCS times(MatrixCSC B,int arbitrary){
          double[] C = new double[nonzeros*arbitrary];
          int[] Cind = new int[nonzeros*arbitrary];
          int Cpointer[] = new int[dimension+1];
          Cpointer[0]=0;
          double scal = 0.0;
          int job = 1;int l1=0;int l2=0;
          int len=-1; int jj=0; int jcol=0;
          int iw[] = new int[dim1];
          int jpos=0; int kb=0; int zb=0;
          int[] BcolPointer = B.getPointerArray();
          int[] BrowInd = B.getRowArray();
          double[] Bvalue = B.getValueArray();
          for(int ii=0;ii<dim1;ii++){iw[ii]=-1;}
          for(int i = 0;i<dim1;i++){
                l1=BcolPointer[i];
                l2=BcolPointer[i+1]-1;
                for(;l1<=l2;l1++){
                scal=Bvalue[l1];
                jj=BrowInd[l1];
                kb=columnpointer[jj];
                zb=columnpointer[jj+1]-1;
                for(;kb<=zb;kb++){
                      jcol=rowindex[kb];
                      jpos=iw[jcol];
                      if(jpos==-1){
                            len++;
                            Cind[len]=jcol;
                            iw[jcol]=len;
                            C[len]=scal*value[kb];
                      }
                      else{
                             C[jpos]+=scal*value[kb];
                       }
                }

              }
             for(int kk=Cpointer[i];kk<=len;kk++){iw[Cind[kk]]=-1;}
             Cpointer[i+1]=len+1;
          }
      return new MatrixCCS(C,Cind,Cpointer);
      }
}
```

3. The class `MatrixCOOR`

```
/**A  class for a sparse matrix on coordinat storage format sorted
  *after rows
  */
  public class MatrixCOOR{
      private double[] value;
      private int[] columnindex;
      private int[] rowindex;
      private int nonzeros;
      private int dimension;
      public MatrixCoord(double[] value,int[] coloumnindex,int[] rowindex,int
dimension){
            this.value = value;
            this.columnindex = columnindex;
            this.rowind = rowindex;
            this.dimension = dimension;
            nonzeros = value.length;
      }

      /**Method for returning the dimension of the matrix
      */
      public int getDimension(){return dimension;}

     /**Method for returning the number of nonzero values of
      * the matrix
      */
      public int getNonzeroSize(){return nonzeros;};

      /**Method for returning the array of values
      */
      public double[] getValueArray(){return value;}

      /**Method for returning the column index array
      */
      public int[] getColumnArray(){return columnindex;}

      /**Method for returning the row index array
      */
      public int[] getRowArray(){return rowindex;}

      /**Method for returning a column index value
      */
      public int getColumnindex(int i){return columnindex[i];}

      /**Method for returning a row index value
      */
      public int getRowindex(int i){return rowindex[i];}

      /**Method for returning a value
      */
      public double getValue(int i){return value[i];}
```

```
/**Method for returning a value
*/
public double getValue(int i,int j){
      double x = 0;
      for(int k=0;k<rowindex.length;k++){
            if(rowindex[k]>i)break;
                  if(rowindex[k]==i)
                        if(columnindex[k]==j)x=value[k];
      }
      return x;
}

/**Method for multiplying two matrices on coordinate storage
  *format sorted by rows
  */
  public MatrixCOOR times(MatrixCOOR B, int arbitrary){
      int k=0;int z=0;
      double valuea=0;
      int jcol=0;int jpos=0;int len=-1;int ja=0;
      double valuec[] = new double[arbitrary*nonzeros];
      int iC[] = new int[arbitrary*nonzeros];
      int jC[] = new int[arbitrary*nonzeros];
      int jw[] = new int[dimension];
      boolean test = true;
      boolean test1 = true;
      boolean test2 = true;
      for(int a=0;a<dimension;a++){jw[a]=-1;}
      for(int i=0;i<dimension;i++){
            while(test1&&k<value.length){
              if(rowindex[k]==i){
                  ja=columnindex[k];
                  valuea=value[k];
                  for(int j=0;j<B.getNonzeroSize()&&test;j++){
                        if(ja==B.getRowindex(j)){
                              z=j;
                              test=false;
                        }
                  }
                  test = true;
                  while(test2&&z<B.getNonzeroSize()){
                        if(ja== B.getRowindex (z)) {
                          jcol=B.getColumnindex(z);
                          jpos=jw[jcol];
                              if(jpos==-1){
                                    len++;
                                    iC[len]=i;
                                    jC[len]=jcol;
                                    jw[jcol]=len;
                              valuec[len]=valuea*B.getValue(z);
                              }
                              else{
            valuec[jpos]=valuec[jpos]+valuea*B.getValue(z);
                              }
                              z++;
                        }
                        else{test2=false;}
                        }
                        z=0;
                        test2=true;
                        k++;
             }
```

```
                        else{test1=false;}
                }
                test1=true;
                for(int b=0;b<dimension;b++)jw[b]=-1;
          }
          return new MatrixCOOR(valuec,jC,iC,B.getDimension());
    }



  /** Method for a matrix-vector product, matrix is
    * stored on COOR
    */
    public Vector matrixvector(Vector v){
          int k = 0;
          boolean test = true;
          double z = 0.0;
          double[] vec = v.getValueArray();
          double[] Y = new double[vec.length];
          for(int i = 0;i<vec.length;i++){
            while(test&&(k<rowindex.length)){
                if(rowindex[k]==i){
                      z += value[k]*vec[columnindex[k]];
                      k++;
                }
                else {test=false;}
             }
                Y[i] = z;
                test = true;
                z = 0;
          }
          return new Vector(Y,Y.length);
    }

    /**Method for a vector-matrix product, matrix stored on COOR
    */
    public Vector vectormatrix(Vector v){
          int j = 0;int k = 0;
          double[] vec = v.getValuArray();
          double[] Y = new double[vec.length];
          boolean test = true;
          for(int i = 0;i< vec.length;i++){
                while(test&&(k<rowindex.length)){
                 if(rowindex[k]==i){
                    j = columnindex[k];
                    Y[j]+=value[k]*vec[j];
                    k++;
                  }
                  else {test=false;}
                }
                test = true;
          }
          return new Vector(Y,Y.length);
    }

}
```

4. The class `MatrixCOOC`

```
/** A  class for a sparse matrix on coordinat storage format sorted       *
after columns
*/
  public class MatrixCOOC{
      private double[] value;
      private int[] columnindex;
      private int[] rowindex;
      private int nonzeros;
      private int dimension;

      public MatrixCOOC(double[] value,int[] coloumnindex,int[] rowindex,int
dimension){
            this.value = value;
            this.columnindex = columnindex;
            this.rowindex = rowindex;
            this.dimension = dimension;
            nonzeros = value.length;
      }

    /**Method for returning the dimension of the matrix
      */
      public int getDimension(){return dimension;}

    /** Method for returning the number of nonzero values of
      * the matrix
      */
      public int getNonzeroSize(){return nonzeros;};

      /**Method for returning the array of values
      */
      public double[] getValueArray(){return value;}

      /**Method for returning the column index array
      */
      public int[] getColumnArray(){return columnindex;}

      /**Method for returning the row index array
      */
      public int[] getRowArray(){return rowindex;}

      /**Method for returning a column index value
      */
      public int getColumnindex(int i){return columnindex[i];}

      /**Method for returning a row index value
      */
      public int getRowindex(int i){return rowindex[i];}

      /**Method for returning a value
      */
      public double getValue(int i){return value[i];}
```

```
/**Method for returning a value
*/
public double getValue(int i,int j){
      double x = 0;
      for(int k=0;k<rowindex.length;k++){
            if(rowindex[k]>i)break;
                  if(rowindex[k]==i)
                        if(columnindex[k]==j)x=value[k];
      }
      return x;
}

/** Method for multiplying two matrices on coordinate storage
  * format(COOC)
  */
  public MatrixCOOC times(MatrixCOOC B, int arbitrary){
      int k=0;int z=0;
      double valuea=0;
      int jcol=0; int jpos=0;int len=-1;int ja=0;
      double valuec[] = new double[arbitrary*nonzeros];
      int iC[] = new int[arbitrary*nonzeros];
      int jC[] = new int[arbitrary*nonzeros];
      int jw[] = new int[dimension];
      boolean test = true;boolean test1 = true;
      boolean test2 = true;
      for(int a=0;a<dimension;a++){jw[a]=-1;}
      for(int i=0;i<dimension;i++){
            while(test1&&k<value.length){
              if(B.getColumnindex(k)==i){
                  ja=B.getRowindex(k);
                  valuea=B.getValue(k);
                  for(int j=0;j<B.getNonzeroSize()&&test;j++){
                        if(ja==columnindex[j]){
                              z=j;
                              test=false;
                        }
                  }
                  test = true;
                  while(test2&&z<B.getNonzeroSize()){
                         if(ja==columnindex[z]){
                              jcol=rowindex[z];
                              jpos=jw[jcol];
                              if(jpos==-1){
                                    len=len+1;
                                    iC[len]=jcol;
                                    jC[len]=i;
                                    jw[jcol]=len;
                              valuec[len]=valuea*value[z];
                              }
                              else{
                              valuec[jpos]+=valuea*value[z];
                              }
                              z++;
                        }
                        else {test2=false;}
                        }
                        z=0;
                        test2=true;
                        k++;
                 }
                else{test1=false;}
```

```
                }
                test1=true;
                for(int b=0;b<dimension;b++)jw[b]=-1;
          }
          return new MatrixCOOC(valuec,jC,iC,B.getDimension());
        }


    /**Method for a matrix-vector product, matrix is
    * stored on COOC
    */
    public Vector matrixvector(Vector v){
          int j = 0;
          int k = 0;
          boolean test = true;
          double[] vec = v.getValueArray();
          double Y[] = new double[vec.length];
          for(int i = 0;i<vec.length;i++){
                while(test&&(k<rowindex.length)){
                      if(columnindex[k]==i){
                            j = rowindex[k];
                      Y[j] += value[k]*vec[columnindex[k]];
                            k++;
                      }
                      else {test=false;}
                }
                test=true;
          }
          return new Vector(Y,Y.length);
    }

    /**Method for a vector-matrix multiplication, matrix is stored
    *  on COOC
    */
    public Vector vectormatrix(Vector v){
          int k = 0;
          boolean test = true;
          double z = 0.0;
          double[] vec = v.getValueArray();
          double[] Y = new double[vec.length];
          for(int i = 0;i<v.getSize();i++){
                while(test&&(k<rowindex.length)){
                      if(columnindex[k]==i){
                            z += value[k]*vec[rowindex[k]];
                            k++;
                      }
                      else {test=false;}
                }
                Y[i] = z;
                test=true;
          }
          return new Vector(Y,Y.length);
    }

}
```

5. The class `Vector`

```
/**The class Vector, this class stores a full vector
  */
  public class Vector {
      double[] value;
      int size;

      public Vector(double[] value,int size){
            this.size=size;
            this.value=value;
      }

      /**Method for returning a value in the vector
        */
      public double getValue(int i){
            return value[i];
      }

      /**Method for returning the size of the vector
        */
      public int getSize(){
            return size;
      }

      /**Method for returning the value array of the vector
        */
      public double[] getValueArray(){
            return value;
      }
  }
```

Appendix C Sparse Arrays class and The Sparse Matrix Concept class

In this Appendix we show the classes for implementing Sparse Arrays and the Sparse Matrix concept that we present in Chapter 5.

Remarks. These classes stores and operates on square matrices ($m=n$). The routines were written early in the work on this thesis and may therefore suffer from that on the basis of consistent notation, that means comments and the code itself. But the overall meaning of the classes and the implemented routines should be rather clear for the reader that has some experience with Java code and these operations.

We have chosen not to comment the code itself, because each routine where the implementation is essential is explained in the text (in the chapter where we represent the routine). Another reason for not comment the code itself is because the routines are 'small' and rather self-explanatory.

1. The class `SparseArrays`

```
/**A class to represent sparse matrices in Java as Sparse Arrays.
  */
  public class SparseArrays{

      private double[][] Avalue;
      private int[][] Aindex;
      private int dimension;
      private int nonzero;
      private int stop;

      public SparseArrays(double[][] Avalue,int[][] Aindex, int nnz){
            this.Avalue = Avalue;
            this.Aindex = Aindex;
            dimension = Avalue.length;
            nonzero = nnz;
        }

        /**A method for setting the value array
        */
        public void setValueArray(double[][] value){Avalue = value;}

        /**A method for setting the value array
        */
        public void setIndexArray(int[][] index){Aindex = index;}


      /**A method for getting the value array
        */
        public double[][] getValueArray(){return Avalue;}

        /**A method for getting the value array
        */
        public int[][] getIndexArray(){return Aindex;}
```

```
      /** A method for multiplying a matrix with a vector
        */
        public Vector matrixvector(Vector V){
            double[] v = V.getValueArray();
            double x = 0.00;
            double[] value;
            int[] index;
            int k = 0;
            int vlength = 0;
            double[] c = new double[v.length];
            int alength = Avalue.length;
            for(int i=0;i<alength;i++){
                  value = Avalue[i];
                  index = Aindex[i];
                  vlength = value.length;
                  for(int j=0;j<vlength;j++){
                        x += value[j]*v[index[j]];
                  }
                  c[i]=x;
                  x = 0;
            }
            return new Vector(c, c.length);
         }

 /** A method for multiplying a vector with a matrix
  */
public Vector vectormatrix(Vector v){
      double[] valuerow = null;
      double[] bvalue = v.getValueArray();
      int[] indexrow = null;
      int alength = Avalue.length;
      double[] cvalue = new double[bvalue.length];
      double value = 0;
      for(int i = 0;i<alength;i++){
            valuerow = Avalue[i];
            indexrow = Aindex[i];
            double val = bvalue[i];
            int vlength = valuerow.length;
            for(int j = 0; j<vlength;j++){
                  cvalue[indexrow[j]] += value*valuerow[j];
            }
      }
      return new Vector(cvalue, cvalue.length);
}

/** A method for matrix multiplication, this routine uses the
* idea with merging sparse list of integers and addition of
* sparse vectors
*/
public SparseArrays times(SparseArrays B){
      double[][] Bvalue = B.getValueArray();
      int[][] Bindex = B.getIndexArray();
      double[][] Cvalue = new double[dimension][1];
      int[][] Cindex = new int[dimension][1];
      int nonzero = 0;
      for(int i = 0;i<dimension;i++){
            double[] avalue = Avalue[i];
            int[] aindex = Aindex[i];
            boolean[] switchArray = new boolean[dimension];
            double[] tempValue = new double[dimension];
```

```
            for(int ii = 0;ii<aindex.length;ii++){
                  int index = aindex[ii];
                  double value = avalue[ii];
                  double[] bvalue = Bvalue[index];
                  int[] bindex = Bindex[index];
                  for(int jj=0;jj<bindex.length;jj++){
                        tempValue[bindex[jj]] += value*bvalue[jj];
                        if(!switchArray[bindex[jj]]){
                              switchArray[bindex[jj]] = true;
                              nonzero++;
                        }
                  }
            }
            double[] cvalue = new double[num];
            int[] cindex = new int[num];
            int k = 0;
            for(int jj = 0;jj<aindex.length;jj++){
                  int index = aindex[jj];
                  int[] bindex = Bindex[index];
                  for(int zz = 0;zz<bindex.length;zz++){
                        if(switchArray[bindex[zz]]){
                        cindex[zz] = bindex[zz];
                        switchArray[bindex[zz]] = false;
                        k++;
                  }
            }
      }
      for(int jj=0;jj<cindex.length;jj++){
            cvalue[jj] = tempValue[cindex[jj]];
      }
      Cvalue[i] = cvalue;
      Cindex[i] = cindex;
      }
      return new SparseArrays(Cvalue, Cindex, nonzero);
}


/** A method for matrix multiplication, this routine uses the
  *idea of contiguously storing of values and indexes
  * and the use of System.arraycopy()
  */
public SparseArrays timesA(SparseArrays B){
      double[][] Cvalue = new double[dimension][1];
      int[][] Cindex = new int[dimension][1];
      int[] temp = new int[dimension];
      double[] tempValue = new double[dimension];
      int[] tempIndex = new int[dimension];
      double[][] Bvalue = B.getValueArray();
      int[][] Bindex = B.getIndexArray();
      int nonzero;
      double scalar = 0;
      int len = -1;
      int index = 0;
      int jcol = 0;
      int jpos = 0;

      for(int i = 0;i<temp.length;i++){temp[i]=-1;}
      long l1 = System.currentTimeMillis();
      for(int i = 0;i<Avalue.length;i++){
            double[] avalue = Avalue[i];
            int[] aindex = Aindex[i];
```

```
            for(int j = 0;j<avalue.length;j++){
                  scalar = avalue[j];
                  index = aindex[j];

                  double[] bvalue = Bvalue[index];
                  int[] bindex = Bindex[index];

                  for(int k = 0;k<bvalue.length;k++){
                        jcol = bindex[k];
                        jpos = temp[jcol];
                        if(jpos == -1){
                              len++;
                              nonzero++;
                              tempIndex[len] = jcol;
                              temp[jcol] = len;
                              tempValue[len] = scalar*bvalue[k];
                        }else{
                              tempValue[jpos]+=scalar*bvalue[k];
                        }
                  }
            }
            double[] cvalue = new double[len+1];
            int[] cindex = new int[len+1];

            System.arraycopy(tempValue, 0, cvalue, 0,len+1);
            System.arraycopy(tempIndex, 0, cindex, 0,len+1);
            Cvalue[i] = cvalue;
            Cindex[i] = cindex;
            for(int ii = 0;ii<len+1;ii++){temp[tempIndex[ii]]=-1;}
            len = -1;
      }
      return new SparseArrays(Cvalue, Cindex, nonzero);
}


/** A method for matrix multiplication, this routine uses the
  *idea of contiguously storing of values and indexes
  * and the use of for-loops
  */
public SparseArrays timesB(SparseArrays B){
      double[][] Cvalue = new double[dimension][1];
      int[][] Cindex = new int[dimension][1];
      int[] temp = new int[dimension];
      double[] tempValue = new double[dimension];
      int[] tempIndex = new int[dimension];
      double[][] Bvalue = B.getValueArray();
      int[][] Bindex = B.getIndexArray();
      int nonzero;
      double scalar = 0;
      int len = -1;
      int index = 0;
      int jcol = 0;
      int jpos = 0;

      for(int i = 0;i<temp.length;i++){temp[i]=-1;}
      long l1 = System.currentTimeMillis();
      for(int i = 0;i<Avalue.length;i++){
            double[] avalue = Avalue[i];
            int[] aindex = Aindex[i];
            for(int j = 0;j<avalue.length;j++){
                  scalar = avalue[j];
```

```
                  index = aindex[j];

                  double[] bvalue = Bvalue[index];
                  int[] bindex = Bindex[index];

                  for(int k = 0;k<bvalue.length;k++){
                        jcol = bindex[k];
                        jpos = temp[jcol];

                        if(jpos == -1){
                              len++;
                              nonzero++;
                              tempIndex[len] = jcol;
                              temp[jcol] = len;
                              tempValue[len] = scalar*bvalue[k];
                        }else{
                              tempValue[jpos]+=scalar*bvalue[k];
                        }
                  }
            }
            double[] cvalue = new double[len+1];
            int[] cindex = new int[len+1];

            for(int ii = 0;ii<len+1;ii++){cvalue[ii]=tempValue[ii];}
            for(int ii = 0;ii<len+1;ii++){cindex[ii]=tempIndex[ii];}
            Cvalue[i] = cvalue;
            Cindex[i] = cindex;
            for(int ii = 0;ii<len+1;ii++){temp[tempIndex[ii]]=-1;}
            len = -1;
      }
      return new SparseArrays(Cvalue, Cindex, nonzero);
}

/** A method for matrix addition, this routine uses the
* idea with merging sparse list of integers and addition of
* sparse vectors
*/
public SparseArrays add(SparseArrays B){
      int[][] Bindex = B.getIndexArray();
      double[][] Bvalue = B.getValueArray();
      int[][] Cindex = new int[dimension][1];
      double[][] Cvalue = new double[dimension][1];
      int nonzero = 0;
      for(int i = 0;i<dimension;i++){
            int[] bindex = Bindex[i];
            double[] bvalue = Bvalue[i];
            int[] aindex = Aindex[i];
            double[] avalue = Avalue[i];
            boolean[] switchArray = new boolean[dimension];
            double[] tempValue = new double[dimension];
            for(int ii = 0;ii<aindex.length;ii++){
                  switchArray[aindex[ii]] = true;
                  tempValue[aindex[ii]] = avalue[ii];
                  nonzero++
            }
            num = aindex.length;
            for(int ii = 0;ii<bindex.length;ii++){
                  if(!switchArray[bindex[ii]]){
                        switchArray[bindex[ii]] = true;
                        tempValue[bindex[ii]] = bvalue[ii];
                        nonzero++;
```

```
                }
                else{
                        tempValue[bindex[ii]] += bvalue[ii];
                }
          }
          int[] cindex = new int[num];
          double[] cvalue = new double[num];
          for(int ii = 0;ii<aindex.length;ii++){
                cindex[ii] = aindex[ii];
                switchArray[aindex[ii]] = false;
          }
      for(int ii = aindex.length,jj=0;ii<bindex.length;ii++,jj++){
                if(switchArray[bindex[jj]]){
                        cindex[ii] = bindex[jj];
                }
          }
          for(int ii = 0;ii<cvalue.length;ii++){
                cvalue[ii] = tempValue[cindex[ii]];
          }

          Cindex[i] = cindex;
          Cvalue[i] = cvalue;
          nonzero += num;
      }
      return new SparseArrays(Cvalue, Cindex, nonzero);
}


/** A method for matrix addition, this routine uses the
  *idea of contiguously storing of values and indexes
  * and the use of arraycopy()
  */
public SparseArrays addition(SparseArrays B){
      double[][] Cvalue = new double[dimension][1];
      int[][] Cindex = new int[dimension][1];
      int[] temp = new int[dimension];
      double[] tempValue = new double[dimension];
      int[] tempIndex = new int[dimension];
      double[][] Bvalue = B.getValueArray();
      int[][] Bindex = B.getIndexArray();
      int nonzero;
      double scalar = 0;
      int len = -1;
      int index = 0;
      int jcol = 0;
      int jpos = 0;

      for(int i = 0;i<temp.length;i++){temp[i]=-1;}

      for(int i = 0;i<Avalue.length;i++){
          double[] avalue = Avalue[i];
          int[] aindex = Aindex[i];
          for(int j = 0;j<avalue.length;j++){
                      jcol = aindex[j];
                      len++;
                      tempIndex[len] = jcol;
                      temp[jcol] = len;
                      tempValue[len] = avalue[j];
                      nonzero++;
          }
          double[] bvalue = Bvalue[i];
```

```
            int[] bindex = Bindex[i];
            for(int j = 0;j<bvalue.length;j++){
                        jcol = bindex[j];
                        jpos = temp[jcol];
                        if(jpos == -1){
                              len++;
                              nonzero++;
                              tempIndex[len] = jcol;
                              temp[jcol] = len;
                        tempValue[len] = bvalue[ii];
                        }else{
                        tempValue[jpos]+= bvalue[ii];
                        }
                  }
            }
            double[] cvalue = new double[len+1];
            int[] cindex = new int[len+1];
      System.arraycopy(tempValue, 0, cvalue, 0,len+1);
      System.arraycopy(tempIndex, 0, cindex, 0,len+1);
            Cvalue[i] = cvalue;
            Cindex[i] = cindex;
for(int ii = 0;ii<len+1;ii++){temp[tempIndex[ii]]=-1;}
            len = -1;
      }
      return new SparseArrays(Cvalue, Cindex, nonzero);
}


/** A method for removing an element in Sparse Arrays
*/
public boolean remove(int i, int j){
      double[] avalue = Avalue[i];
      int[] aindex = Aindex[i];
      boolean isInRow = false;
      for(int ii = 0;ii<aindex.length;ii++){
            if(j==aindex[ii])isInRow = true;
      }
      if(isInRow){
            double[] value = new double[avalue.length-1];
            int[] index = new int[avalue.length-1];
            for(int ii = 0,jj=0;ii<aindex.length;ii++){
                  if(j!=aindex[ii]){
                        value[jj] = avalue[ii];
                        index[jj] = aindex[ii];
                        jj++;
                  }
            }
            Avalue[i] = value;
            Aindex[i] = index;
      }
      return isInRow;
}

/** A method for inserting an element in Sparse Arrays
*/
public boolean setValue(int i, int j, double value){
      double[] avalue = Avalue[i];
      int[] aindex = Aindex[i];
      boolean test = true;
      for(int ii = 0;ii<aindex.length;ii++){
            if(aindex[ii]==j){
```

```
                  avalue[ii] = value;
                  test = false;
            }
      }
      if(test){
            double[] valuenew = new double[aindex.length+1];
            int[] indexnew = new int[aindex.length+1];
            for(int ii = 0;ii<aindex.length;ii++){
                  valuenew[ii] = avalue[ii];
                  indexnew[ii] = aindex[ii];
      }
      valuenew[valuenew.length-1] = value;
      indexnew[indexnew.length-1] = j;
      Avalue[i] = valuenew;
      Aindex[i] = indexnew;
      }
      return test;
 }


/**Method for performing a symbolic factorisation on Sparse Arrays
*/
public void symbolicFactorisation(){
      int[] b = null, bi = null, bj = null;
      double[]a = null, ai = null;
      int x = 0, j1 = 0;
      int xx = 0;
      int yy = 0;
      boolean test = false;
      for(int i = 2;i<Aindex.length;i++){
            bi = Aindex[i];
            ai = Avalue[i];
           for(int j = 1;j<i-1;j++){
            bj = Aindex[j];
            if(jIsInbi(j,bi)){
            for(int z = 0;z<bj.length&&(!test);z++){
                  x = bj[z];
            for(int y = 0;(y<stop)&&(y<bi.length)&&(!test);y++){
            if(x==bi[y]){
            test=true;
            xx = x;
            yy = y+1;
           }
           }
        }
        if(test){
            a = new double[ai.length+1];
            b = new int[bi.length+1];
           for(int ii = 0,jj=0;ii<b.length;ii++,jj++){
            if(ii==yy){
                  b[ii]=j;
                  a[ii]=0.0;
                  jj--;
           }
        else{
           b[ii]=bi[jj];
           a[ii]=ai[jj];
        }
      }
         Avalue[i] = b;
         Aindex[i] = a;
```

```
         test = false;
      }
    }
  }
 }
      stop = 0;
}

/**Method for checking if there is an element already present in *position j
in the row bi
*/
public boolean jIsInbi(int j, int[] bi){
      boolean test = true;
      for(int i = 0;i<bi.length;i++){
            if(bi[i]==j){
            test = false;
      }
}
            if(test==false){
            for(int ii = 0;ii<bi.length;ii++){
                  if(bi[ii]>j){
                        stop = ii;
                  }
            }
           }
      return test;
}
}
```

2. The class `SparseMatrix` and `Rows`

```
/**A class the stores a sparse matrix on the sparse matrix concept.
*/
public class SparseMatrix{
      private Rows[] rows;
      public SparseMatrix(Rows[] rows){this.rows=rows;}

      /**Method for performing a matrix-vector product
      *This is the approach similar to the Sparse Arrays implementation
      */
      public Vector matrixvectorA(Vector v){
            double[] vec = v.getValueArray();
            double[] c = new double[vec.length];
            double z = 0;
            double[] value = null;
            int[] index = null;
            Rows row = null;
            for(int i = 0;i<vec.length;i++){
                  row = rows[i];
                  value = row.getValueArray();
                  index = row.getIndexArray();
                  for(int j = 0;j<value.length;j++){
                        z+=value[j]*vec[index[j]];
                  }
                  c[i] = z;
                  z = 0;
            }
            return new Vector(c, c.length);
      }
```

```
	/**Method for performing a matrix-vector product
	* This is the object-oriented approach
	*/
	public Vector matrixvectorB(Vector v){
		double[] vec = v.getValueArray();
		double[] c = new double[vec.length];
		double z = 0;
		double[] value = null;
		int[] index = null;
		Rows row = null;
		for(int i = 0;i<vec.length;i++){
			c[i] = rows[i].vectorproducts(v);
		}
		return new Vector(c, c.length);
	}
}


/**A class to represent the rows of the Sparse Matrix class
  */
public class Rows{
	private double[] values;
	private int[] indexes;

	public Rows(double[] values, int[] indexes){
		this.values = values;
		this.indexes = indexes;
	}

	/** Returns the row values.
	*/
	public double[] getValueArray(){return values;}

	/** Returns the row indexes.
	*/
	public int[] getIndexArray(){return indexes;}

	/** Performs a dot prodcut.
	*/
	public double vectorproducts(Vector v){
			double[] vec = v.getValueArray();
			double z = 0;
			for(int j = 0;j<values.length;j++){
				z+=values[j]*vec[indexes[j]];
			}
			return z;
	}
}
```

3. The class `Vector`

```
/**The class Vector, this class stores a full vector
  */
  public class Vector {
      double[] value;
      int size;

      public Vector(double[] value,int size){
            this.size=size;
            this.value=value;
      }

      /**Method for returning a value in the vector
        */
      public double getValue(int i){
            return value[i];
      }

      /**Method for returning the size of the vector
        */
      public int getSize(){
            return size;
      }

      /**Method for returning the value array of the vector
        */
      public double[] getValueArray(){
            return value;
      }
  }
```

References

[1] Ronald F. Boisvert, Jack J. Dongarra, Roldan Pozo, Karin A. Remington, and G. W. Stewart(1998) Developing numerical libraries in Java

[2] JAMA: A Java Matrix Package
http://math.nist.gov/javanumerics/jama/

[3] JAMPACK: A JAVA PACKAGE FOR MATRIX COMPUTATIONS, G. W. Stewart
ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html

---

[4] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 1, Unordered and ordered representation, pages 22-23

[5] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 1, Merging a sparse list of integers, pages 30-31

[6] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 1, Addition of sparse vectors with the help of an expanded integer array of pointers, pages 35-36

[7] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 7, Transposition of a sparse matrix, pages 236-238

[8] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 7, Ordering a sparse representation, pages 239-240

[9] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 7, Permutation of rows or columns of a sparse matrix: First procedure, pages 240-241

[10] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 7, Permutation of rows or columns of a sparse matrix: Second procedure, pages 240-241

[11] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 7, Addition of sparse matrices, pages 242-243

[12] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 7, Example of addition of two sparse matrices, pages 243-244

[13] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 7, Product of a general sparse matrices by a full row vector, pages 248-249

[14] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 7, Product of a row vector by a general sparse matrices, pages 249-250

[15] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 7, Multiplication of sparse matrices, pages 253-254

[17] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 7, Algorithm for the symbolic multiplication of two sparse matrices given in row-wise format, pages 255-256

[18] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press,
chapter 7, Algorithm for the numerical multiplication of two sparse matrices given in row-wise format, pages 256-258

---

[19] Alan George and Joseph W. Liu (1983), Computer Solution of Large Sparse Positive Definite Systems, Prentice Hall, chapter 1, Cholesky's Method and the Ordering Problem, pages 2-8

[20] Alan George and Joseph W. Liu (1983), Computer Solution of Large Sparse Positive Definite Systems, Prentice Hall, chapter 1, Positive Definite Versus Indefinite Matrix Problems, pages 9-10

[21] Alan George and Joseph W. Liu (1983), Computer Solution of Large Sparse Positive Definite Systems, Prentice Hall, chapter 3, Basic Terminology and Some Definitions, pages 36-41

[22] Alan George and Joseph W. Liu (1983), Computer Solution of Large Sparse Positive Definite Systems, Prentice Hall, chapter 3, Computer representation of graphs, pages 41-43

[23] Alan George and Joseph W. Liu (1983), Computer Solution of Large Sparse Positive Definite Systems, Prentice Hall, chapter 5, Symmetric Factorization, pages 92-101

[24] Alan George and Joseph W. Liu (1983), Computer Solution of Large Sparse Positive Definite Systems, Prentice Hall, chapter 5, Computer representation of Elimination Graphs, pages 102-114

[25] Alan George and Joseph W. Liu (1983), Computer Solution of Large Sparse Positive Definite Systems, Prentice Hall, chapter 5, The Minimum Degree Ordering Algorithm, pages 115-138

[26] Alan George and Joseph W. Liu (1983), Computer Solution of Large Sparse Positive Definite Systems, Prentice Hall, chapter 5, Sparse Storage Schemes, pages 138-158

---

[27] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 1, Why Is It Slow, pages 1

[28] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 2, Garbage Collection, pages 23-26

[29] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 3, Garbage Collection, pages 64

[30] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 3, Faster VMs, pages 68-73

[31] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 3, Better Optimizing Compilers, pages 73-82

[32] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 3, Compile to Native Machine Code, pages 89-90

[33] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 3, Native Method Calls, pages 90-91

[34] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 4, Avoiding Garbage Collection, pages 112-115

[35] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 6, Variables, pages 169-172

[36] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 7, Loops and Switches, pages 174-178

[37] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 7, Appropriate Data Structures and Algorithms, pages 293-294

[38] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 7, Collections, pages 294-296

[39] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 7, Java 2 Collection, pages 296-298

---

[40] Ronald F.Boisvert, Jose Moriera, Michael Philipsen, Roldan Pozo, Java and Numerical Computing

[41] Java Grande Forum
http://www.javagrande.org/

[42] Daniel Steinberg, Java Grande Forum Pushes Java™ Technology Toward New Heights
http://developer.java.sun.com/developer/technicalArticles/Interviews/javagrande/

[43] Java Numerics
http://math.nist.gov/javanumerics/

[44] Eric Armstrong, HotSpot: A new breed of virtual machine
http://www.javaworld.com/jw-03-1998/jw-03-hotspot.html

[45] The Java HotSpot™ Virtual Machine: Technical White Paper

http://java.sun.com/products/hotspot/docs/whitepaper/Java_HotSpot_WP_Final_4_30_01.html#pgfId=1082013

[46] Doug Bell, Make Java Fast: Optimize!
http://www.javaworld.com/javaworld/jw-04-1997/jw-04-optimize.html/

[47] The Java Grande Forum Benchmark Suite
http://www.epcc.ed.ac.uk/javagrande/

[48] Reggie Hutcherson, The basics of Java platform performance
http://www.javaworld.com/javaworld/jw-03-2000/jw-03-javaperf_2.html/

[49] Bill Venners, The lean, mean, virtual machine
http://www.javaworld.com/javaworld/jw-06-1996/jw-06-vm.html/

[50] Tim Lindholm Frank Yellin, The Java™ Virtual Machine Specification Second Edition
http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html/

[51] James Gosling,Bill Joy,Guy Steele,Gilad Bracha, The Java Language Specification
http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html

[52] Ronald F Boisvert and Roldan Pozo, Java Numerical Performance
http://math.nist.gov/~RBoisvert/talks/java-wg25-2001/wg25-01_files/frame.htm/

[53] John Zukowski, The historical classes—Arrays
http://www-106.ibm.com/developerworks/java/library/j-arrays/?dwzone=java/

[54] Ulrich Stern, Java vs C++
http://verify.stanford.edu/uli/java_cpp.html/

[55] Bill Venners, Java's garbage-collected heap
http://www.javaworld.com/javaworld/jw-08-1996/jw-08-gc.html/

[56] James Gosling, The Evolution of Numerical Computing in Java
http://java.sun.com/people/jag/FP.html

[57] Mark Roulo, Accelerate your Java apps!
http://www.javaworld.com/javaworld/jw-09-1998/jw-09-speed.html

---

[58] Kees van Reeuwijk, Frits Kuijlman and Henrik J Sips, Spar: a Set of Extensions to Java for Scientific Computation*

[59] Mark Bull, Lorna Smith, Lindsey Pottage and Robin Freeman

[60] David Bacon, Kava : A Java Dialect with a Uniform Object Model for Lightwieght Classes*

[61] Mark Bull, Lorna Smith, Lindsey Pottage, Robin Freeman, Benchmarking Java against C and FORTRAN*

[62] Jose Moirera, Sam Midkiff, Manish Gupta A comparisons of Three Approaches to Language, Compiler, and Library Support for Multidimensional Arrays in Java*

[63] Arnold Nielsen, Thilo Kielman, Henri Bal and Jason Maaasen, Object Based Communication in Java*

*Proceedings of the ACM 2001 Java Grand/ISCOPE Conference

---

[64] Yousef Saad (1996).Iterative Methods for Sparse Linear Systems, WPS

[65] Ian F Darwin (2001) Java CookBook O'Reilly

[67] Java Documentation API version 1.3.1
http://java.sun.com/j2se/1.3/docs/api/index.html

[68] David A. Watt and Deryck F. Brown(2001), Java Collections, , Wiley, chapter 3, pages 34-49

[69] Extensible Markup Language (XML)
http://www.w3.org/XML/

[70] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press, Introduction pages 1-3

[71] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 1, What to Measure, pages 13-14

[72] Frequently asked questions about the Java HotSpot VM, Benchmarking he Java HotSpot VM, http://java.sun.com/docs/hotspot/PerformanceFAQ.html

[73] Loyd N. Trefethen, David Bau, III, Numerical Linear Algebra, Appendix, The Definition of Numerical Analysis, pp 321-327

[74] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 3, Optimization Performed When Using the –O Option, pages 85-88

[75] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 4, Object Creation Statistics, pages 96-97

[76] Sergio Pissanetsky, (1984), Sparse Matrix Technology,Academic Press, chapter 1, The sparse row-wise format pages 20-21

[77] Yosef Saad (1996), SPARSEKIT
http://www.cs.umn.edu/research/arpa/SPARSKIT/sparskit.html

[78] Jack Shirazi, (2000), Java™ Performance Tuning, O'REILLY, chapter 11, Java 2 Collections, pages 296-298

[79] Performance tests show Java as fast as C++, Carmine Mangione**,**
http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf.html

[80] Towards Array Bound Check Elimination in Java Virtual Machine Language/Hongwei Xi and Songtao Xia

[81] Matrix Market
http://math.nist.gov/MatrixMarket/

[82] Roldan Pozo and Karin A. Remington, (April 1996), SparseLib++

[83] Grunnkurs i matematiske beregninger
http://www.math.uio.no/math100/b/kurs/index.html