Towards a Next-Generation Matrix Library for Java

Holger Arndt
Department of Computer Science
Technical University of Munich
85747 Garching, Germany
Email: arndt@ujmp.org

Markus Bundschus
Insitute for Computer Science
Ludwigs-Maximilians-Universität Muenchen
Oettingenstr. 67, 80538 Munich, Germany
Email: bundschu@ujmp.org

Andreas Naegele
Department of Computer Science
Technical University of Munich
85747 Garching, Germany
Email: naegele@ujmp.org

Abstract

Matrices are essential in many fields of computer science, especially when large amounts of data must be handled efficiently. Despite this demand for matrix software, we were unable to find a Java library which was flexible enough to match all our needs. In this paper, we present the *Universal Java Matrix Package (UJMP)*, an innovative software architecture in Java to store and process matrices with interfaces to external data sources such as Excel files or SQL-databases, allowing to handle data which would not fit into main memory. In contrast to all other approaches which we are aware of, our package is the only one to support very large matrices with up to 2^{63} rows or columns. Th use of variable argument lists provides a convenient way for accessing multi-dimensional data without the need to specify dimensionality at compile time. Arbitrary data types be handled through the use of Java Generics. Another key feature is the strict separation of interfaces and classes, making data storage implementation and math engine exchangeable, at runtime. This flexible architecture allows the user to decide whether operations should be optimized for speed or memory usage and makes the system easily extendable through existing libraries, incorporating their individual strengths.

1 Introduction

Matrix computations are essential in various fields of computer science, especially for data intensive tasks, occurring e.g. in machine learning, artificial intelligence, or data mining. Another example is the analysis of graphs in collaborative networks which require large sparse adjacency matrices to store the network structure. A remarkable amount of literature exists on the implementation of fast algorithms for matrix algebra, e.g. [1] or [2] and many of these algorithms are available as libraries for the programming language of your choice, e.g. the Fortran implementations LINPACK or LAPACK [3]. They use Basic Linear Algebra Subprograms (BLAS), which are divided into operations for vectors, matrix–vector and matrix–matrix. An implementation for C is also available and, additionally, the Fortran code has been translated to Java using f2j forming the JLAPACK package [4].

The Java Development Kit (JDK) itself does not provide routines for matrix algebra, however, a matrix package has been proposed by IBM in 1999 (see [5, 6]) and has been submitted to the Java Community Process Program [7]. Unfortunately, the draft has been withdrawn due to slow development progress. Therefore, a number of other libraries have been developed, with JAMA [8] and Colt [9] being the most common. JAMA provides a standard implementation for dense two-dimensional double matrices and is used inside many other packages to calculate the inverse of a matrix or perform a decomposition. Colt also supports sparse matrices as well as three-dimensional matrices with double values or Java objects. Other packages are commons-math by the Apache Software Foundation [10], Java Vecmath from the Java3D library [11] or Matrix Toolkits for Java (MTJ) [12].

However, some of these libraries have not received updates within the last years, and most of them have been developed for Java 1.4. Thus, they do not support the new language features of Java 5, such as Generics, providing type safety at compile time [13]. They also lack some basic functionality, such as import and export filters or additional functions such as sin, cos, mean or variance. Many of the libraries do not even support sparse or object matrices, or three-dimensional matrices.

As a matter of fact, the biggest drawback is the difficulty to combine an existing matrix library with another one or with your own implementation. This is desirable, as each of the packages has its own advantages or weaknesses, which becomes clear in table 1 showing a comparison between our software and the most common matrix libraries for Java.

Only Colt and MTJ are easily extendable as they provide abstract classes which can be extended, while an interface with more than 20 methods has to be implemented to extend commons-math.

In the following sections, we will present the *Universal Java Matrix Package (UJMP)*, a novel matrix library which has several major advantages over existing software packages: (1) exchangeability of matrix implementations due to a strict separation of programming interface and its implementation, (2) extensibility due to a coherent class hierarchy, and integration of existing matrix packages

Table 1: Comparison of our Universal Java Matrix Package (UJMP) with other common matrix libraries for Java.

	UJMP	Jama	Colt	MTJ	commons-math	vecmath
extendable	+	−	+	+	⊙	−
dense matrices	+	+	+	+	+	+
sparse matrices	+	−	+	+	−	−
2D matrices	+	+	+	+	+	+
3D matrices	+	−	+	−	−	+
4D matrices	+	−	−	−	−	+
matrices >4D	+	−	−	−	−	−
$> 2^{31}$ rows/columns	+	−	−	−	−	−
object entries	+	−	+	−	−	−
generic cell entries	+	−	−	−	−	−
matrices > RAM	+	−	−	−	−	−
entry iterators	+	−	+	+	−	−
advanced operators	+	−	+	−	−	−
import/export filters	+	−	−	+	−	−
visualization methods	+	−	−	−	−	−

and their algorithms, (3) direct access to various storage implementations (e.g. files or databases), (4) support for very large matrices and an innovative way of accessing data in multiple dimensions, (5) the ability to handle arbitrary data types equipped with a convenient conversion mechanism, and (6) visualization methods with support for very large matrices.

2 Concepts for a Novel Matrix Package

During our daily work it has become obvious, that none of the existing Java matrix libraries completely fulfilled our demands and so we started to collect requirements for a novel and universal approach.

Most importantly, a matrix library must be able to store large amounts of data in big matrices, with support for more than two or three dimensions. As described above in table 1, none of the existing libraries can handle multi-dimensional matrices or matrices with more than 2^{31} rows or columns. However, multi-dimensional matrices are neccessary e.g. in physics or when three-dimensional data with a time component is processed (e.g. in functional mag-

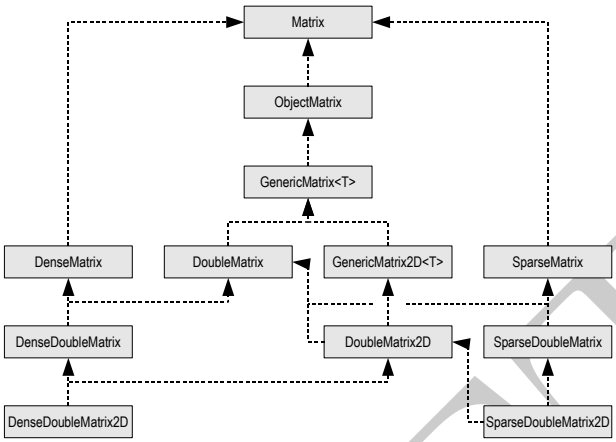

Figure 1: Interface hierarchy of UJMP. For simplicity, only interfaces relevant for double matrices are shown.

netic resonance imaging). Additionally, the library should be easily extendable through import and export filters and support various data sources, including main memory, hard disk, or databases. It should provide the user with processing methods for numerical data, text or other objects, and include a graphical interface for data visualization, avoiding the need to export data to other programs for this purpose. In the following sections, we will present a novel architecture which fulfills the required demands.

2.1 Interface and Class Model

One of the most important aspects of our Universal Java Matrix Package is the architecture of the class hierarchy and its interfaces. This design primarily supports extensibility and exchangeability of the matrix implementation, whereas this goal has been neglected by other libraries. As mentioned above, Colt and MTJ can be extended easily, but only within the limitations of their class definition. For instance, there is no way for Colt to support more than three dimensions, and MTJ cannot be modified to allow for other data types than `double`. Combining two libraries is thus extremely difficult, as the interfaces are not compatible to one another.

In order to overcome this limitation, our library makes extensive use of interfaces encapsulating the desired functionality, which renders the architecture independent from the actual implementation. Fig. 1 shows the interface hierarchy for `double` matrices; the structure for other types such as `float` or `boolean` is similar, but is omitted for simplicity. For convenience, we support the following object types and Java primitives as matrix cell entries: `boolean`, `byte`, `char`, `Date`, `double`, `float`, `int`, `short`, `long`, `String`, `Object` and generic objects. Support for `BigInteger` and `BigDecimal` is currently under development. All matrix

interfaces inherit from `Matrix`, offering the functionality shared by all matrix implementations independent of their data type and dimensionality, e.g. methods to query size or content type, import and export functionality and data processing methods. `ObjectMatrix` and `GenericMatrix` define the methods needed for handling objects, i.e. the getters and setters `getObject()` and `setObject()` for their entries. For a generic matrix, the object type of the cells can be specified, e.g. `GenericMatrix<MyObject>`.

`DoubleMatrix` is an interface which inherits from `GenericMatrix<Double>` which makes `getObject()` return a `Double` object. In addition, it declares `getDouble()` and `setDouble()`, allowing a matrix also to return `double` primitives instead of `Double` objects, as conversion between these types is unnecessary and time-consuming. While the interfaces defining dense and sparse matrices remain empty, `DoubleMatrix2D` adds a more efficient method to access cells in a two-dimensional matrix, as will be discussed in section 2.5. Three-dimensional matrices do not have their own interface definition, as they will be handled as ordinary multi-dimensional matrices.

For most interfaces, a template implementation [14] is provided (e.g. `AbstractDenseDoubleMatrix2D`), delivering the default functionality for this matrix type, i.e. operations for addition, subtraction, etc. Some methods in this template class are declared as final, since they may not be overriden. `isSparse()`, for instance, must always return `true` in a sparse matrix implementation. Other methods are not final providing the default implementations of algorithms, e.g. the computation of plus or minus. These methods may be overridden in derived classes, when custom implementations are desired. For example, our wrapper for the Colt library will use Colt's linear algebra functions and fall back to our implementation for operations not supported by Colt (e.g. calculation of the average value or visualization).

Besides interfaces and abstract classes, our library offers a number of default matrix implementations for the most common use cases. Two-dimensional implementations are available for dense matrices, with data stored in a two-dimensional array in memory, while another implementation uses a binary file on disk, thus enabling the software to handle matrices that do not fit into main memory.

For sparse matrices, we offer a reference implementation using a `HashMap` which may be supplemented in the future with other storage formats, such as compressed row storage (CRS). In addition, we have incorporated matrices linking to images, Excel files, or tables in an SQL database. As mentioned above, the intrinsic implementation becomes secondary, provided it obeys our interface definition.

2.2 Adaptors to Other Matrix Libraries

As mentioned in section 1, well-tested matrix libraries are available for Java, most of them being stable for years. They provide implementations specialized for fast matrix computations with advanced decomposition methods, and it is a challenging task to offer a new implementation with the same performance

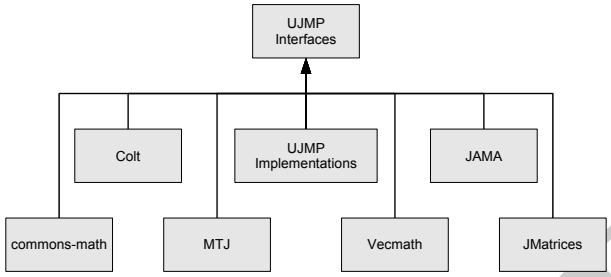

Figure 2: Overview of matrix implementations supported by our software.

and reliability. To benefit from those packages, we have created adaptors [14] making their implementation available to our matrix library, which is facilitated through the open architecture described in section 2.1.

The adaptor to a two-dimensional dense double matrix from the Colt library, for instance, inherits from `AbstractDenseDoubleMatrix2D` which provides a large number of processing functions for this matrix type (plus, minus, average, variance, etc.). Only the getters and setters for the cell entries remain to be implemented, and are mapped to a Colt matrix stored within our adaptor class. Computations supported by Colt are overridden in the adaptor, allowing Colt to calculate plus, minus and multiply, while our library is used for other functions (average, variance, import, export, visualization, etc.). On the other hand, our software relies on other libraries for some functions. In particular, we do not support decomposition methods (e.g. eigen value or singular value decomposition) at the moment. Fig. 2 shows the matrix libraries which can be used together with our software. To offer a central point for specifying the matrix implementation to use, we employ the factory pattern [14]. `MatrixFactory.sparse(x,y)`, for instance, can be configured to create a matrix from the Colt library. Switching the underlying matrix implementation at runtime is also possible as well as intermixing different implementations, e.g. using Colt for sparse and JAMA for dense matrices.

2.3 Adaptors to Other Data Sources

Our software is not limited to integrating other matrix implementations, as it has been designed to handle any type of data independent of the underlying storage implementation, provided that it can be transformed into matrix format. This evidently applies to data aligned in two-dimensional grids or 3D cubes, but also to one-dimensional lists or high-dimensional data. Through the use of adjacency or incidence matrices, graphs with arbitrary entities as nodes can be tranformed into a matrix representation and thus can be processed with our software.

In fact, there are only a few prerequisites to render arbitrary data repre-

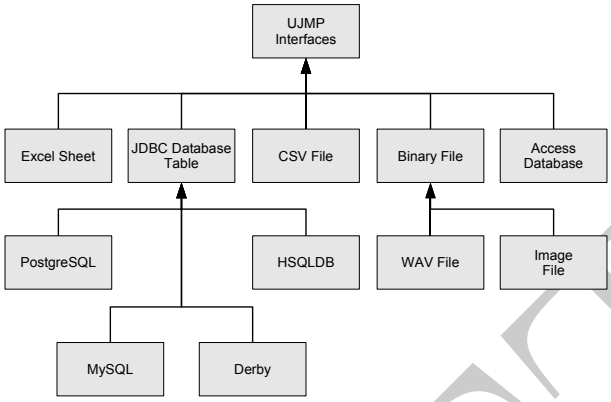

Figure 3: Overview of the data sources which can be connected to our software.

sentable in matrix format. In particular, data must be (1) aligned orthogonally and not e.g. hexagonally, (2) ordered in the sense that, for each pair of cells, a relation exist with one cell being e.g. "lower" or "more left" than the other, (3) non-ambigously addressable, i.e. every item has a position with no two cells sharing the same coordinates, and (4) defined in size and dimensionality.

Data complying with these conditions can easily be mapped to a matrix in our framework with only getter and setter methods and `getSize()` remaining to be implemented. Fig. 3 shows an overview of data sources which are supported in the current version of our library. As can be seen, we provide interfaces to a number of databases e.g. MySQL, PostgreSQL, HSQLDB, or Derby. Other databases should work as well, as long as a suitable JDBC driver exists. The user can submit connection details and a select statement to these adaptors and the query result will be wrapped inside a matrix with as many rows and columns as the underlying SQL `ResultSet`. Accessing single cells in this matrix moves the cursor within this `ResultSet` accordingly and the specified entry is retrieved. When the JDBC driver of the database can handle query results which do not fit into main memory by partitioning the `ResultSet`, this also holds for the adaptor matrix, giving the user direct access to large data sources. The same feature is available in adaptors to binary files on disk, including WAV audio or bitmap files. In addition, we have implemented a read-only adaptor for CSV files which allows row-wise access to large amounts of data not fitting into main memory. We are not aware of any other software that can handle these files in a similar way. Without our library, the only work-around is to write a script file for preprocessing, which becomes impractical when more advanced calculations are required.

Import and export from Microsoft Access databases or Excel sheets is also supported, however, the libraries used for reading the data (Jackcess and JExcelApi) require that the content fits into main memory.

We would like to mention that our software also provides adaptors for Java

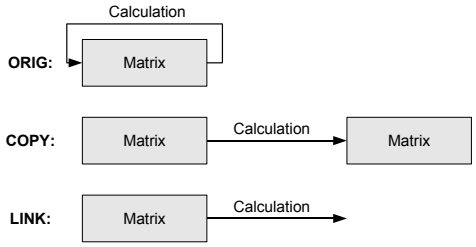

Figure 4: Illustration of the different calculation modes.

collection classes, making a `List`, a `Set` or a `Map` adressable as an ordinary matrix. This way it is possible e.g. to add a number of double values to a `List` and use our software for visualization or to calculate the average value of the entries.

2.4 Calculations

For most computations executed on a matrix of data values, three different processing modes can be desired.

(1) An intuitive option is to perform the operation on the original data and return a new matrix with the calculated values. In this case, the resulting matrix and also the original data are still available and can be used for further processing.

(2) In other cases, creating a new matrix object is not intended, especially when the source matrix is large, e.g. a table in a SQL database. For example, a user may want to round all double values in a table to the closest integer number. The desired behavior is to adjust all values directly in the database without the need to copy the data to a new matrix and write back the result afterwards. The drawback in this scenario is that the original data is destroyed after processing.

(3) A third way of processing is to return a different view on the data. Using the same example as above, a "calculation matrix" is returned which computes the rounded value from the original data each time a cell is requested by the user, thus leaving the original data unchanged.

The calculation modes are illustrated in Fig. 4. However, not all of these modes may be available depending on the calculation. Entry-wise operations such as abs, sin, cos, tan, round, etc. support all of them, while operations changing the size of a matrix, cannot modify the original.

Calculation methods are encapsulated in separate Java classes with a similar structure of interfaces and abstract classes as for matrices. As a result, not only the data storage implementation of our library can be exchanged, but also the underlying calculation engine. E.g. for calculating the inverse of a matrix, our library can either use the implementation of Colt, JAMA, commons-math or MTJ.

We have done performance tests, comparing calculations with our matrix library to others, which can be found on our web page [15]. We found that the Universal Java Matrix Package is among the fastest libraries for matrix multiplication and calculating the transpose. Interestingly, it is not possible to name a single best library, as the results vary a lot depending on the operating system and computer architecture. In this light, it makes sense that our software can integrate other libraries, so that the user can chose the fastest implementation for his environment.

It should also be mentioned, that we provide interfaces to Matlab, Octave, R, and GnuPlot, allowing the user to use their computation methods or visualization capabilities if desired.

2.5 Getter and Setter Methods

Accessing data in a multi-dimensional matrix is not as straightforward as it at first seems. To specify the position of a cell in each dimension, the same number of arguments in a method is required. For two- or three-dimensional matrices, the methods `getDouble(x,y)` or `getDouble(x,y,z)` can be declared easily, however, as the number of dimensions increases, it becomes impractical to enumerate all parameters separately. In this case, transmitting an array of coordinates is preferred, the size of which can be adjusted as required.

In this scenario, however, there would be different methods depending on the dimensionality, which makes it tedious to create matrix functions. E.g. a subtraction function would have to be written several times, for matrices with two parameters, three parameters and for the array type. Leaving out the methods for lower dimensions is not an option, as an array would have to be created each time a cell entry is set or retrieved, what decreases performance and impairs readability of the program code.

With the new language features of Java 5, a solution for this dilemma has arrived. Variable arguments lists (Varargs) can automatically convert a number of parameters into an array of objects [13], thus mapping `getDouble(x,y)` and `getDouble(x,y,z)` to the same method `getDouble(long... pos)`. `pos` is an array filled with as many arguments as the user has provided. This makes it possible to have a single method for matrices with an arbitrary number of dimensions, with only the length of the array varying. The requirement that the variable argument list must be the last parameter in a function remains the only restriction. Accordingly, the new value for the cell entry must be the first parameter within the setter function: `setDouble(double v, long... pos)`.

However, for every call of a getter or setter method, an array object is created implicitly, which is more time consuming than the passing of the single coordinates, as free memory first has to be allocated. In addition, used arrays must be disposed of by the garbage collector when they are not needed anymore, which also has a negative impact on performance.

As a reference value, it takes on average 0.8 seconds to set all entries in a

$10,000$x$10,000$ double matrix on current hardware[1], when the arguments are passed directly, and 3.2 seconds using Varargs.

A possible work-around may be to have the user create a number of position arrays and reuse them as often as possible, changing their values when required. However, this method annihilates the intuitive way of entering coordinates in a variable argument list and there is no advantage in comparison to passing the array itself.

It is preferable to equip a two-dimensional matrix with a `getDouble(long x,long y)` method in addition to the Varargs method. This method is declared in the interface `DoubleMatrix2D`, whereas `DoubleMatrix` declares the Varargs method. Depending on the interface used to access the implementation, a call to `getDouble(x,y)` will be mapped to one of the two implementations, allowing to decide between easy access in multiple dimensions and high performance for the two-dimensional case.

As you have noticed above, we use 64-bit `long` values to specify the location of a cell. Although it is not possible in Java to have arrays with more than 2^{31} entries (as `int` values are signed only 31 bit remain), there may be examples, in which access to data in larger matrices is required, e.g. when the matrix is sparse. However, also dense matrices can hit the upper size limit, e.g. it is impossible to address single bytes in a file larger than 2GB when `int` coordinates are used.

The time to convert a `long` value into an `int` is not negligible, although it is shorter compared to the creation of arrays for the Varargs coordinates. Using the same example setup as described above, setting matrix entries via two `int` coordinates takes on average 0.5 seconds and is faster than type-casting the `long` values.

Therefore, two-dimensional matrices are equipped with an additional method `getDouble(int x,int y)`, which avoids the type cast for matrix content addressed with less than 2^{31} bit. In other cases, e.g. when the entries are stored in a file on disk, `long` values are used and the type cast is required for `int` arguments.

2.6 Automatic Entry Type Conversion

In most cases, data stored in a matrix is strictly specified, e.g. working on a mathematical task requires matrices with `double` values. In other cases, `float` or `int` values are preferred, while `String` values are used for text processing tasks. As described above, our library allows for specification of the data type and provides appropriate interfaces. In particular, `DoubleMatrix` is used for double precision numbers, `StringMatrix` for text and `ObjectMatrix` for Java objects. A `GenericMatrix` can be configured with a custom object type, making Java enforce the correct format for cells at compile time, generating an error if the user tries to assign inappropriate data.

[1]Intel Core 2 Duo T8300 2.4 GHz, 800 MHz FSB, 3 MB Cache, Java 1.6 Update 11, Windows Vista 64, 4GB 2xPC2-5300 DDR2 SODIMM (667 MHz)

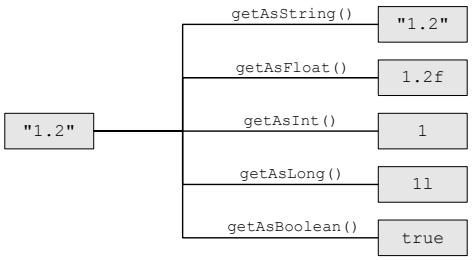

Figure 5: Automatic type conversion of a String into other formats.

However, it may also be desirable to have greater flexibility regarding the data stored within a matrix. For example, when a CSV file is imported from disk, all entries are `String` values at first and additional knowledge is required to decide whether a value should be treated as a `String`, `double` or `int`, etc. Converting the entire matrix into the desired format may be an adequate preprocessing step which, however, is only appropriate when all entries share the same target type. In mixed data formats the first column may contain a label while the remaining columns represent numeric measurements. Since transforming all data into `double` values would destroy the label information, the only choice is to convert each entry into the desired format and use an `ObjectMatrix` for storage. As all cells are objects in the transformed matrix, reading data requires type casts and exception handling in case that the actual differs from the expected object type.

In order to simplify this usage scenario, we have implemented an automatic type conversion mechanism, transforming the data as desired. In addition to the method `getDouble()` for double matrices, a method `getAsDouble()` is provided for all matrices, which converts the data to a `double` value. Numbers such as `int` or `float` are cast to `double`, while `String` values are converted using `Double.parseDouble()`. Exceptions are automatically caught, and `Double.NaN` is returned when the conversion was unsuccessful. Conversion methods are provided for all types described in section 2.1, e.g. `getAsInt()`, `getAsFloat()`, `getAsString()`, etc.

Fig. 5 exemplarily shows the conversion of a `String` into different target types. This mechanism allows the user to choose the best data representation according to each use case, without having to implement his own conversion routine. Thus it becomes feasible to combine heterogeneous data in a single matrix and decide depending on the context how it should be processed. The same feature is known from Excel sheets, in which each cell can hold either numbers, text, or date and time values.

2.7 Coordinate Iterators

It often becomes necessary to loop over entries in a matrix, e.g. in order to apply a computation or to search for items with certain properties. For this scenario, the iterator pattern [14] provides a convenient solution. In our software, we have included the method `allCoordinates()`, which returns an iterator looping over all possible positions in a matrix. At each step an array of long values is returned defining the current position. For each iterator only one array is created with its entries being modified, as the iterator is moved to the next position. This procedure has the advantage over creating a new array for each step, that performance remains high and memory is not wasted for creating arrays.

However, when the matrix is sparse, it is not feasible to iterate over all entries comprised in it. Instead, a second iterator `availableCoordinates()` can be used, which enumerates present storage locations, as probably most cells in a sparse matrix remain empty. This iterator is of particular importance when the matrices are very large, as looping over all coordinates would be impractical in this case.

The Colt library, for example, provides a sparse matrix implementation and a method `getNonZeros()` to return its non-zero entries. However, this method checks for each possible entry whether it contains a zero value or not, which requires a long time for large matrices. On the other hand, applying a function to a sparse matrix using `forEachNonZero()` works properly and only present entries are considered.

Listing 1 shows an example for summing up entries in a 4-dimensional sparse matrix, which is also appropriate for higher dimensions.

Listing 1: Example for summing up entries in a 4-dimensional matrix.

```
// create a 4D sparse matrix 100x100x100x100
Matrix m = MatrixFactory.sparse(100,100,100,100);

// set entry at [5,5,5,5] to 1.0
// and entry at [10,10,10,10] to 2.0
m.setAsDouble(1.0, 5, 5, 5, 5);
m.setAsDouble(2.0, 10, 10, 10, 10);

// perform the calculation
double sum = 0.0;
for(long[] pos : m.availableCoordinates())
  sum += m.getAsDouble(pos);

System.out.println(sum); // prints 3.0
```

Execution of this particular example took less than 10ms on the same reference platform as above, whereas iterating over all possible positions requires more than 10s. This is example is not supposed to benchmark performance, but should rather point out the importance of these iterators also in terms of code design: Without iterators, the user would have to nest 4 separate for-loops in another, which becomes impractical as the number of dimensions increases. In addition, our library provides two iterators named `selectedCoordinates()`. One

of them takes a multi-dimensional array of long values as argument, representing the coordinates, over which the user intends to iterate. The second method of the same name uses a `String` to encode the desired selection in a style similar to Matlab (for an introduction to Matlab see e.g. [16]). The String `"1,3-5;*"`, for example, returns an iterator over all columns and rows 1, 3, 4 and 5.

2.8 Matrix Annotation

In many cases, data is only valuable when additional information about it is available. For income data it is crucial to know, whether the currency is US$, Yen or €. A time series of measurements has a different meaning when the samples are taken every day or every millisecond. In extreme cases, data may be rendered useless when this meta-information is lost.

We have therefore provided an option to attach annotation to each matrix. This meta-information can be used to assign a name to a matrix or to label a row, column, or the entire axis. The annotation object is not set by default and is created on demand thus saving memory if no meta-information is required. Internally, an annotation object stores the values in a `HashMap` with the key being the position of the label. However, custom implementations are also allowed, since the structure of an annotation object is defined in its own interface.

Annotation may not only be used for labeling data, but also for specifying e.g. data type or default value for a column, such as in a table of an SQL database. Consequently, annotation objects are not limited to storing `String` values but arbitrary objects can be assigned as label, axis label, or row/column label and also higher dimensional data can be labeled in this scheme.

Depending on the operation performed on a matrix, the annotation is copied, modified or discarded. E.g. when the sum of two matrices of the same size is calculated, the resulting matrix will contain the unmodified annotation data of the first matrix. A transpose function will rotate annotation in an appropriate way, while other operations may eliminate annotation in the result, e.g. axis labels are lost when a 1x1 matrix is returned which contains the maximum value of all entries in a matrix.

2.9 Visualization

All of the matrix libraries shown in table 1 do not provide visualization methods. We feel, that this is a major drawback, as it leaves the user with two choices. He can either include a visalization library into his software project, or start another program such as Matlab each time a graphical representation is required. The first option would require to convert the data into a format useable by the library with its interfaces and classes, while, for the second option, he would have to export the data to disk and load it into another software. However, both methods have to copy the data in some way, which becomes impractical for large matrices.

Therefore, we provide our own visualization methods with a strict adherence to the MVC (Model-View-Controller) pattern [14]. Since all data is comprised

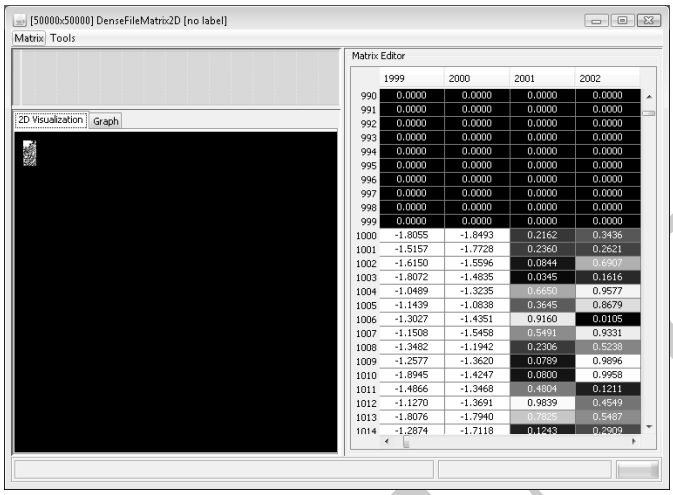

Figure 6: Visualization of a very large matrix.

within a matrix, the visualization routine must merely perform a mapping from this data to the pixels on the screen, what requires no additional storage and allows large matrices to be displayed.

Listing 2 creates a very large matrix and displays an editor together with a two-dimensional structure plot, where cells are marked in different colors according to their values (see Figure 6). In this example, the largest part of the matrix remains empty, with the small area containing data clearly visible in the upper left corner. This demonstrates, that visualization is an extremely helpful tool, when information must be extracted from large files, as errors in the data become obvious and areas with relevant information can be discerned from unuseable data.

Listing 2: Visualizing a very large matrix.

```
Matrix m1, m2, m3;
long rows = 50000; long cols = 50000;

// create a 20GB matrix in a file on disk
m1 = new DenseFileMatrix2D(rows, cols);

// select a small portion of the matrix
// and fill it with random values
m2 = m1.select(Ret.LINK,"1000-5000;1000-3000");
m2.rand(Ret.ORIG);

// select another submatrix and subtract 2.0
m3 = m1.select(Ret.LINK,"1000-2000;1000-2000");
m3.minus(Ret.ORIG, false, 2.0);

m1.showGUI(); // visualize the matrix
```

Although the example matrix requires 20GB of disk space, it can be displayed

in a few seconds, since only a small part of the data is actually relevant to create the visualization. Size and speed of the hard disk are the only limiting factors.

3 Summary and Discussion

In this paper, we have presented the *Universal Java Matrix Package (UJMP)*, a novel and innovative matrix library for Java. Our requirement to cope with large amounts of data, and the fact that no existing library was able to fulfill all our demands, has convinced us to design a new library which is able to handle very large dense, sparse and multi-dimensional matrices as well as graphs within the same framework, even when the data does not fit into main memory. A major feature of our software is the extendable architecture, which integrates various data sources as well as existing matrix libraries, incorporating their individual strenghts. In particular, the import and export functions have to be mentioned which support a wide range of file formats and databases. Calculations can be performed in different "modes" (see section 2.4) to give the user the utmost control on how data is processed. In addition, visualization methods are provided, with the ability to deal with large data sets.

Most of the features described in this paper have already been implemented and are part of our library, facilitating our work with data sources in Java. The here described matrix library also serves as the mathematical back-end of our second Java project, the *Java Data Mining Package (JDMP)* [17]. However, weaknesses are still present regarding calculations or the graphical interface, and also the sometimes incomplete integration of external data sources is still an issue.

We also lack some standard matrix implementations and calculations, e.g. compressed row storage (CRS) for sparse matrices or decomposition methods. However, it is not planned to include those in the near future, since we do not want to invent the wheel twice and our primary goal is the integration of other libraries making their features and performance available in our framework.

Another important issue is the integration of parallel computations, since current processors usually have more than one core already and will defnitely be multi-core in the future. Currently, we can employ other matrix libraries such as Parallel Colt for this purpose, but we also think about integrating additional frameworks, e.g. Hadoop [18], Terracotta [19] or the Java Parallel Processing Framework [20]. Other interersting concepts such as Parallel Arrays are discussed in JSR166 [21] for future Java versions, which may open up new optimization opportunities for matrix calculations.

Another issue we would like to improve is the availability of documentation, which is rather limited at the moment, making it unnecessarily hard to get acquainted with our library.

It has to be emphasized, that our software is licensed under LGPL, which allows it to be included in commercial applications. Source code and jar files are freely available through our website [15] in the hope that the here introduced matrix library will attract many users and developers.

References

[1] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing, 2nd edition.* Cambridge: Cambridge University Press, 1992.

[2] G. H. Golub and C. F. V. Loan, *Matrix Computations, 3rd edition.* Johns Hopkins University Press, 1996.

[3] University of Tennessee, "LAPACK," 2008, http://www.netlib.org/lapack/.

[4] D. Doolin, J. Dongarra, and K. Seymour, "JLAPACK - compiling LAPACK Fortran to Java," *Scientific Programming*, vol. 7, no. 2, pp. 111–138, 1999.

[5] J. E. Moreira, S. P. Midkiff, and M. Gupta, "A standard Java package for technical computing," *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.

[6] P. V. Artigas, M. Gupta, S. P. Midkiff, and J. E. Moreira, "High performance numerical computing in Java: Language and compiler issues," *Proceedings of the 12th Workshop on Language and Compilers for Parallel Computers*, 1999.

[7] J. E. Moreira, "Java multiarray package," 2005, http://jcp.org/en/jsr/detail?id=083.

[8] The MathWorks and the National Institute of Standards and Technology (NIST), "JAMA: A Java matrix package," 2005, http://math.nist.gov/javanumerics/jama/.

[9] CERN – European Organization for Nuclear Research, "Colt," 1999, http://acs.lbl.gov/~hoschek/colt/.

[10] Apache Software Foundation, "Apache commons mathematics library," 2008, http://commons.apache.org/math/.

[11] Java.net, "The vecmath package," https://vecmath.dev.java.net/.

[12] B.-O. Heimsund, "Matrix toolkits for Java (MTJ)," 2006, http://ressim.berlios.de/.

[13] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification, 3rd edition.* Boston, MA: Addison-Wesley, 2005.

[14] J. W. Cooper, *The Design Patterns Java Companion.* Addison Wesley, 1998.

[15] H. Arndt, M. Bundschus, and A. Naegele, "Universal Java Matrix Package (UJMP)," 2009, http://www.ujmp.org.

[16] S. Otto and J. P. Denier, *An Introduction to Programming and Numerical Methods in Matlab*. Berlin: Springer, 2005.

[17] H. Arndt, M. Bundschus, and A. Naegele, "Java Data Mining Package (JDMP)," 2009, http://www.jdmp.org.

[18] Apache Software Foundation, "Apache hadoop," 2008, http://hadoop.apache.org/.

[19] Terracotta Inc., "Terracotta," 2009, http://www.terracotta.org/.

[20] Laurent Cohen, "Java parallel processing framework," 2009, http://www.jppf.org/.

[21] D. Lea, "JSR-166," 2008, http://gee.cs.oswego.edu/dl/concurrency-interest/.