Università di Pisa
Dipartimento di Informatica

Technical Report: TR-03-09

MAJA: Matrix Computation in Java

F. Romani
Dipartimento di Informatica
Via Buonarroti 2, 56127 Pisa, Italy. *E-mail: romani@di.unipi.it*

Address: Corso Italia 40, 56125 Pisa, Italy. Tel: +39 050 2212700 Fax: +39 050 2212726

MAJA: Matrix Computation in Java*

F. Romani
Dipartimento di Informatica
Via Buonarroti 2, 56127 Pisa, Italy. *E-mail: romani@di.unipi.it*

Abstract

MAJA is a package for MAtrix computation in JAva. The package allows defining various kinds of structured matrices operating in an unified way; General, Symmetric, Sparse, Band, Toeplitz, Circulant matrices are implemented among the others. It is possible also to operate on Block matrices, Additive Structures, Tensor Products, Implicit Inverses and so on. The package organization is similar to the Java Collections API. It is easy to exploit the presence of more than one processor without affecting the final user code. The source code is platform independent and has been tested on Linux, Windows, and Macintosh computers; test benchmarks are presented, both on single and double-processor computers. Also examples of implementation of involved linear algebra algorithms are presented.

1 Goals of the project

In the practice of matrix computations, several software high level tools are available which greatly reduce the programming efforts (e.g. MATLAB o Mathematica). At the other end, the use of low level languages like FORTRAN or C allows exploiting at the best the computational power of the machine at the expense of a rather involved programming.

The goal of this package is to join the ease of high level structured matrix manipulation with the advantages of a low level language. In this way it is possible to test special algorithms on structured matrices from the point of view of behavior, stability and efficiency in an environment as close as possible to the environment of the final user.

As programming language we chose to use Java for its intrinsic advantages:

- cleanness of object implementation;
- portability in any platform;
- ease of integration with user interface, image processing and web-computing;

*This research was partially supported by MURST, within the Project PRIN "Inverse Problems in Medical Imaging", prot. 2002013422_001.

- simple exploiting of a multiprocessor machine capabilities, when available.

Much of the project has been inspired by the two similar packages JAMA [2] and JAMPACK [4].

2 Package structure and class hierarchy

2.1 Interfaces

At the root of the class hierarchy there are two interfaces, representing the matrix and vector abstract type, respectively.

- `interface Matrix`. Represents the matrix abstract type. Some methods defined are `get` which returns the value of an element, `set` which changes the value of an element (allowed only for subclasses with variable elements), `getRowDimension`.

- `interface vector`. Represents the vector abstract type (the name is in all lowercases to distinguish it from the built-in class vector). Some methods defined are `get`, `set`, `scalarMultiplyInPlace`.

2.2 Concrete classes implementing `Matrix`

Here are some classes implementing `Matrix`.

- `GeneralMatrix`: matrix without structure represented as the double dimensional array of its elements.

- `GeneralSparseMatrix`: sparse matrix represented as an array of hash tables, this type can be used as a `GeneralMatrix` but only the non zero elements are stored. The complexity of accessing elements is high but it is possible any manipulation of elements.

- `FixedSparseMatrix`: sparse matrix represented in compressed row format, no modification of structure is allowed. The complexity of computing matrix vector products is low, this structure is suitable for implementing iterative solution of large linear systems.

- `RankOneMatrix`: rank one matrix stored as two vectors.

- `LowRankMatrix`: the low rank matrix stored as two rectangular matrices of any type.

- `FixedBlockMatrix`: matrix represented as a double dimensional array of matrices of any type, but with the same shape.

- `AdditiveMatrix`: matrix represented as the sum of two matrices of any type. It is possible to combine additive matrices to represent sums of more than two elements.

- `TensorProductMatrix`: matrix represented as the tensor product of two matrices of any type.
- `CompositeMatrix`: matrix are represented via an auxiliary class `FilterMatrix` that determines for any value of the indices the matrix from which the element has to be taken out. Below it is shown, as an example, how to represent in this way the inverses of tridiagonal matrices.
- `IdentityMatrix`.
- `DiagonalMatrix`.
- `BandMatrix`.
- `LowerTriangularMatrix`.
- `UpperTriangularMatrix`.
- `SymmetricMatrix`.
- `CirculantMatrix`.
- `ZeroMatrix`.

The last classes have a self-explaining name. For any class the proper memory allocation is used.

It is possible to trasform a matrix type into another one. If the conversion is compatible, the matrix remains the same, otherwise some elements are set to zero. For example, if `D` is a `DiagonalMatrix`, the matrix `new GeneralMatrix(D)` is the same matrix represented in the general format. On the other hand, if `G` is a `GeneralMatrix`, the matrix `new GeneralMatrix(G)` is the diagonal part of `G`.

2.3 Concrete classes implementing `vector`

There are two implementations for the type vector.

- `SimpleVector` is a vector represented as the array of its values.
- `BlockVector` is a vector represented as the array of vectors (this representation is useful when dealing with block matrices).

2.4 Static classes

Like in the Java Collections API, here are available two static classes which allow operating on vectors and matrices.

`Matrices`, contains static methods to perform, matrix multiplication, matrix linear combination, etc.

`Vectors`, contains static methods to perform dot products, linear combination, the unity vectors, etc.

2.5 Internal representation of results

When operating on matrices of different type, the resulting type is the more convenient in terms of representation. For example, in Table 1 the internal representation of the results of matrix multiplication between various classes of matrices are presented.

`General`	$\times$	`LowRank`	$\rightarrow$	`LowRank`
`LowerTriangular`	$\times$	`LowerTriangular`	$\rightarrow$	`LowerTriangular`
`LowerTriangular`	$\times$	`Diagonal`	$\rightarrow$	`LowerTriangular`
`LowerTriangular`	$\times$	`LowRank`	$\rightarrow$	`LowRank`
`UpperTriangular`	$\times$	`UpperTriangular`	$\rightarrow$	`UpperTriangular`
`UpperTriangular`	$\times$	`Diagonal`	$\rightarrow$	`UpperTriangular`
`UpperTriangular`	$\times$	`LowRank`	$\rightarrow$	`LowRank`
`Symmetric`	$\times$	`LowRank`	$\rightarrow$	`LowRank`
`Band`	$\times$	`Diagonal`	$\rightarrow$	`Band`
`Band`	$\times$	`LowRank`	$\rightarrow$	`LowRank`
`Circulant`	$\times$	`Circulant`	$\rightarrow$	`Circulant`
`Circulant`	$\times$	`LowRank`	$\rightarrow$	`LowRank`
`Diagonal`	$\times$	`LowerTriangular`	$\rightarrow$	`LowerTriangular`
`Diagonal`	$\times$	`UpperTriangular`	$\rightarrow$	`UpperTriangular`
`Diagonal`	$\times$	`Band`	$\rightarrow$	`Band`
`Diagonal`	$\times$	`Diagonal`	$\rightarrow$	`Diagonal`
`Diagonal`	$\times$	`FixedSparse`	$\rightarrow$	`FixedSparse`
`FixedSparse`	$\times$	`Diagonal`	$\rightarrow$	`FixedSparse`
`LowRank`	$\times$	`General`	$\rightarrow$	`LowRank`
`LowRank`	$\times$	`LowerTriangular`	$\rightarrow$	`LowRank`
`LowRank`	$\times$	`UpperTriangular`	$\rightarrow$	`LowRank`
`LowRank`	$\times$	`Symmetric`	$\rightarrow$	`LowRank`
`LowRank`	$\times$	`Band`	$\rightarrow$	`LowRank`
`LowRank`	$\times$	`Circulant`	$\rightarrow$	`LowRank`
`LowRank`	$\times$	`Diagonal`	$\rightarrow$	`LowRank`
`LowRank`	$\times$	`FixedSparse`	$\rightarrow$	`LowRank`
`LowRank`	$\times$	`LowRank`	$\rightarrow$	`LowRank`
$AnyMatrix$	$\times$	`Zero`	$\rightarrow$	`Zero`
`Zero`	$\times$	$AnyMatrix$	$\rightarrow$	`Zero`
`RankOne`	$\times$	$NonZeroMatrix$	$\rightarrow$	`RankOne`
$NonZeroMatrix$	$\times$	`RankOne`	$\rightarrow$	`RankOne`
`Identity`	$\times$	$AnyMatrix$	$\rightarrow$	$SameMatrix$
$AnyMatrix$	$\times$	`Identity`	$\rightarrow$	$SameMatrix$
	$Otherwise$		$\rightarrow$	`General`

Table 1: Internal representation of results of matrix multiplication.

2.6 Algorithms

Any linear algebra algorithm can be implemented, either as a method in the proper class or as a static method in `Matrices` or `vectors`. Special attention is paid to the implementation of iterative methods for the solution of large linear systems like Gauss-Seidel, Conjugate Gradient and other Krylov methods.

3 Examples

3.1 A block Matrix

The matrix deriving from the discretization of the Poisson operator in a square is a large, sparse structured matrix, which can be seen as the sum of two tensor products between tridiagonal Toeplitz and identical matrices. The following sample code, constructs a matrix of this kind of any size without allocating unnecessary memory.

```
...
double[ ] d =2, -1;
Matrix A = new SymmetricToeplitzMatrix(n, d);
Matrix I = new IdentityMatrix(n);
Matrix D1 = new TensorProductMatrix(A,I);
Matrix D2 = new TensorProductMatrix(I,A);
Matrix T = new AdditiveMatrix(D1,D2);
Matrices.print(T);
...
```

3.2 A more sophisticated example

As a more involved example of the use of the MAJA package we will show how to perform the explicit representation of the inverse of a symmetric tridiagonal matrix, by using a `CompositeMatrix`.

It is well known [1, 3] that the inverse $Z = \{z_{ij}\}$ of a proper symmetric tridiagonal matrix can be put in the form

$$z_{ij} = \begin{cases} u_i v_j & \text{if } i \leq j, \\ u_j v_i & \text{if } i > j. \end{cases}$$

The class `TriInv` implements the interface `FilterMatrix` and is used to build a `CompositeMatrix` A. Any time an element value is required from A, the method `whichMatrix` is called and the element is taken from one of the component matrices (in this case the component matrices are the two rank one matrices $U = uv^T$ and $V = vu^T$). The composite matrix A can be used as any other matrix and its inverse results to be tridiagonal.

```
import maja.*;
public class TriInv implements FilterMatrix{
   Matrix U,V;
   public TriInv (vector u, vector v) {
      super(n,n);
      U = new RankOneMatrix(new vector(u), new vector(v));
```

```
      V = new RankOneMatrix(new vector(v), new vector(u));
   }
   public Matrix whichMatrix (int i, int j){
      if(i<j) return U;
         else return V;
   }
}

static public void main(String[ ] args) {
   double[ ] va =1,2,3,4,5;
   double[ ] ua =1,1,1,1,1;
   vector u = new SimpleVector(va);
   vector v = new SimpleVector(ua);
   CompositeMatrix A = new CompositeMatrix(new TriInv(u, v));
   Matrix A1= A.inverse( );
   Matrices.print(A1);
}
```

4 Benchmarks

In Table 2 a sample of timings for matrix multiplication of size 500 is presented. The experiment were carried out using a bi-processor 650MHz Pentium III computer running Windows 2000 Professional and a 867MHz PowerPC G4 running Apple MacOS 9.2. In the first two columns we see the first and second operand, the third column is the time in seconds obtained using only one Pentium processor, the fourth column is the time obtained using both the processors (when applicable) and the last column represents the time on the mono-processor Macintosh machine.

It is self evident that the package is able to exploit both the presence of two processors and the structure of the operands.

In Table 3, a sample of timings for matrix inversion, (size 500) is presented. In this case too, the package is able to exploit the matrix structure to gain efficiency.

5 Availability

The prototype of the package will be available at the end of 2002.

References

[1] R. Bevilacqua, M. Capovani, Proprietà delle matrici a banda ad elementi ed a blocchi, Boll. Un. Mat. Ital., B13 (1976) 844-861.

			PC seq.	PC par.	Mac
General	×	General	9.422	5.578	9.596
General	×	Band	9.453	5.718	9.753
General	×	Circulant	9.453	5.703	9.797
General	×	Diagonal	0.047		0.053
General	×	LowerTriangular	9.438	5.828	9.719
General	×	LowRank	0.032	0.031	0.026
General	×	RankOne	0.015		0.018
General	×	Symmetric	9.453	5.719	9.843
Band	×	Band	9.453	5.688	9.782
Band	×	Circulant	9.484	5.688	9.751
Band	×	Diagonal	0.0		0.01
Band	×	LowerTriangular	9.468	5.765	9.842
Band	×	LowRank	0.11	0.094	0.066
Band	×	RankOne	0.047		0.04
Band	×	Symmetric	9.484	5.703	9.83
Circulant	×	Circulant	0.015		0.018
Circulant	×	Diagonal	0.047		0.04
Circulant	×	LowerTriangular	9.5	5.765	9.828
Circulant	×	LowRank	0.062	0.078	0.066
Circulant	×	RankOne	0.047		0.041
Circulant	×	Symmetric	9.485	5.703	9.778
LowerTriangular	×	Diagonal	0.031		0.027
LowerTriangular	×	LowerTriangular	2.297		1.801
LowerTriangular	×	LowRank	0.11	0.093	0.078
LowerTriangular	×	RankOne	0.031		0.046
LowerTriangular	×	Symmetric	9.484	5.671	9.795
LowRank	×	LowRank	0.0		0.0040
LowRank	×	RankOne	0.0		0.0030
LowRank	×	Symmetric	0.125		0.104
RankOne	×	RankOne	0.0		0.01
RankOne	×	Symmetric	0.094		0.065
Symmetric	×	Symmetric	9.516	5.719	9.833

Table 2: Timings for matrix multiplication with some couples of operands.

Matrix Type	→	*Inverse Type*	PC seq	Mac
General	→	General	13.84	10.06
LowerTriangular	→	LowerTriangular	3.17	2.19
UpperTriangular	→	UpperTriangular	3.14	2.21
Symmetric	→	Symmetric	13.90	10.14
Band	→	General	14.00	9.95
Circulant	→	Circulant	3.29	1.09
Diagonal	→	Diagonal	0.00	0.00
GeneralSparse	→	General	14.00	10.14
Identity	→	Identity	0.00	0.00

Table 3: Timings for matrix inversion with different structure.

[2] J. Hicklin, C. Moler, et al. JAMA : A Java Matrix Package,
`http://math.nist.gov/javanumerics/jama/`

[3] P. Rósza, R. Bevilacqua, P. Favati, F. Romani, On band matrices and their inverses, Linear Algebra Appl, 150 (1991) 287-296.

[4] G. W. Stewart, Jampack (JAva Matrix PACKage).
`ftp://thales.cs.umd.edu/pub/Jampack/Jampack/AboutJampack.html`
`ftp://gams.nist.gov:/pub/Jampack/Jampack/AboutJampack.html`