JCArray – the jCrunch™ Java Array Classes

David S. Dixon
Least Squares Software
Albuquerque, NM

ABSTRACT

In connection with jCrunch™ Lapack, Least Squares Software has developed Java array classes to encapsulate a Fortran-like data array implementation. These classes are designed to provide 1-D and 2-D arrays directly to Native methods while presenting the Java programmer with a useful, well-behaved, object-oriented API. The proposed representation has much in common with other proposals, such as JAMA[1], JNL[2] and NINJA[3]. The principal differences among these proposals are: degree of exposure to the internal data representation; persistence; reliance on specific implementations of Blas, Lapack, Linpack, etc.; and utility methods useful to "array jockeys" familiar with APL, Python, etc. Design goals of the JCArray classes include providing the same API to both pure Java and Native methods, and to support special matrices such as tri-diagonal, banded, etc.

1 About the JCArray Classes

Three principal goals drove the design of the JCArray Class and its descendents – JCFVector, JCDVector, JCFMatrix and JCDMatrix:

1. Provide a well-mannered Java wrapper for Fortran-like column-major numerical arrays to be passed to Native routines. These arrays may have a number of attribute parameters of interest to the Native procedures, such as an internal leading dimension (`lDim`), offset pointers into the array (`gOffset`), etc. The wrappers must provide access methods to those parameters.

2. Provide a robust framework for high-performance methods supporting the Lapack BLAS (basic linear algebra subprograms) array operations such as multiplication, determinants, eigenvalue calculations, systems of equation solvers, etc. The framework must be able to support either pure Java or Native BLAS.

3. Provide easy to use methods for performing typical array operations, such as transposing or reversing, taking a submatrix or subvector, computing a trace, etc.

2. The JCArray API

Class JCArray

A JCArray object is instantiated with a rank, shape, and type, where rank = shape.length, and type is (in Rev 1.1) either **float** or **double**. JCArray is intended as a general array class, supporting arrays of an arbitrary number of dimensions and arbitrary data types. In JCArray Rev 1.1, however, this class serves only as a way to pass any object of the JCxVector or JCxMatrix classes which must then be cast to the appropriate class. Future releases may support a data array of arbitrary dimensions and data-type.

Utility methods are included to get and set all parameters, such as getRank(), getShape() and setShape(int[]).

Class JCxVector

JCxVector objects are instantiated with

```
rank = 1
shape = (int[1]){length}
datatype = x(x:float for JCFVector, double for JCDVector).
```

Objects of this class are primarily intended to support vector operations in Native Fortran. Additionally, they are the receptacle objects for row, column, and diagonal slicing operations in the JCxMatrix classes.

Terse example:

```
JCFVector v = new JCFVector(3); // instantiate an empty,
                                // 3-element float vector
```

Verbose example:

```
double[] d = new double[] {1.0, 2.0, 3.0, 4.0};
JCDVector v = new JCDVector( d,          // data array
                             4,          // array length
                             1,          // leading dimension (lDim)
                             0,          // global offset (gOffset)
                             1,          // index origin (indexOffset)
                             1)          // global spacing (gStep)
```

There is a copy constructor, called by the `clone()` method, which makes a deep copy of the JCxVector object. That is, the underlying array `data[]` is copied from one object to the other.

Deep copying (`COPY`) is the default mode, but some methods support a `NOCOPY` mode. These methods return a pointer to the original data array, and may set the `gOffset` to point to a subset (`subVector`) within the array. These methods are intended to support extremely large data arrays for which time and memory constraints make multiple copies prohibitive.

JCxVector methods which support the `NOCOPY` mode include: `getDataNoCopy()`, `setDataNoCopy(double[])`, `subVectorNoCopy(int, int)`, and `transposeNoCopy()`.

Utility methods are included to get and set all parameters, such as `v.getSize()`, `v.getGOffset()`, etc.

There are also `print()` methods (taken from the Jama[1] specification):

```
v.print(10,3);

      1.000     2.000     3.000     4.000
```

Class JCxMatrix

JCxMatrix objects are instantiated with

> `rank = 2`,
> `shape = (int[2]){mrows, ncols}` and
> `datatype = x` (`x`: `float` for JCFMatrix, `double` for JCDMatrix).

The constructors range from terse and simple to verbose, setting dimensions, offsets, and data array.

Terse example:

```
JCDMatrix m = new JCMatrix(4,4); // instantiate an empty,
                                 // 4x4 double matrix
```

Verbose example:

```
float[] f = new float[] {1.0, 3.0, 2.0, 4.0);

JCFMatrix m = new JCFMatrix( m,   // data array
                             2,   // row count (mrows)
                             2,   // column count (ncols)
                             1,   // leading dimension (lDim)
                             0,   // global offset (gOffset)
                             0,   // row index origin (rowOffset)
```

```
1)    // column index origin (colOffset)
```

Utility methods are included to get and set all parameters, such as `m.getShape()`, `m.getRowCount()`, `m.setLDim(int)`, etc.

Deep copying (`COPY`) is the default mode, but some methods support a `NOCOPY` mode. These methods return a pointer to the original data array, and may set the `gOffset` to point to a subset (`subMatrix`) within the array. These methods are intended to support extremely large data arrays for which time and memory constraints make multiple copies prohibitive.

JCxMatrix methods which support the `NOCOPY` mode include: `getColNoCopy(int)`, `getDataNoCopy()`, `setDataNoCopy(double[])`, `subMatrixNoCopy(int, int)`, and `transposeNoCopy()`.

There are `print()` methods (also from Jama[1]):

```
m.print(10,3);

        1.000     2.000
        3.000     4.000
```

Additional utilities for the JCxMatrix Classes include: `take()`, `reshape()`, `ravel()`, `resize()`, `repeat()`, `choose()`, `concatenate()`, etc.

3. Special Matrix Forms

Lapack supports 28 matrix types[4], such as diagonal, band, Hermitian, triangular, *etc*. Many of these distinctions have come about because either a) some high-performance algorithms can only be performed on certain types of matrices, or b) especially large sparse matrices of these types can be more compactly stored (packed). To reduce confusion over the words **type** and **class**, in JCArray parlance the symmetry class is called the matrix ***form*** (i.e. general, diagonal, symmetric, triangular) while the packing is called ***storage format*** (i.e. conventional, band, or packed). A JCxMatrix object has an integer vector `storageFormat[]` the zeroeth element of which gives the form and storage format, the remaining elements specifying parameters of the storage format**.** For example, a general matrix has a one-element

```
storageFormat[] = int[]{JCxMatrix.GENERAL}.
```

A band matrix with two lower subdiagonals and one upper subdiagonal would have a three-element

```
storageFormat[] = int[]{JCxMatrix.GENERAL_BAND,2,1}
```

The `storageFormat` is set at instantiation from constructor arguments and can only be changed by methods that determine if the internal data meet the criteria for the requested form and storage format. If not, a new object will have to be instantiated with the desired form and storage format. Although JCArray Rev. 1.1 supports only GENERAL matrices (general form, conventional storage format), future releases will support all matrix types in Lapack.

4. Matrix operations and BLAS

The jCrunch™Lapack library contains low-level Java classes for both pure Java and NativeBroker Blas. The jCrunch line will be expanded to include high-level wrappers which manage all the Fortran-like details of the low-level classes, providing a pure object-oriented API. The jCrunch wrappers are designed to be easily adapted to custom Blas packages, taking advantage of high-performance, and application- or platform-tuned products. At that time, the JCArray classes will be enhanced to include the high-level Blas methods for array products, rotations, etc. In the meantime, the Numerics Working Group of the Java Grande Forum is completing proposals for a Java Array class which will include a standard API for array arithmetic. The JCArray classes will implement the standard API as arithmetic operations are added.

5. Relationship to JAMA, JNL, and NINJA

A joint proposal of The MathWorks and the National Institute of Standards and Techcnology (NIST), JAMA instantiates matrix objects with a two-dimensional Java array as the internal representation. The class contains a very basic set of access methods, a variety of elementary operations, and a number of Blas-like methods for decomposition and equation solvers. The package is focused on elegant object orientation and tuned specifically for Java two-dimensional arrays. As such, it does not support Native methods or arrays of higher or lower rank.

The general approach in JAMA is similar to JCArray, bet JAMA doesn't support one-dimensional arrays or the Fortran-like data representation that makes it possible for JCArray to work seamlessly with both pure Java and Native methods. The inclusion of Lapack-like and Blas-like computations in the class reduces the generality of a matrix object.

The JNL array classes from Visual Numerics, Inc. are made up entirely of static classes acting on Java one-dimensional and two-dimensional arrays. Thus, all array access is done via normal Java assignment and reference. Matrix methods include basic math, determinate, various norms, transpose, and solvers using LU factorization or QR decomposition. The package includes a Complex class from which Java arrays of Complex objects can be built, and classes for computation objects. For example, computating a

Complex Cholesky decomposition involves the instantiation of a ComplexCholesky object and execution of its R() method, which retains the various solution components as private members. These members are returned via access methods such as inverse(), condition(), determinant(), etc.

The approach to arrays in JNL is quite different from JCArray, but the computation objects are very similar to those of jCrunch Lapack. Because JNL does all basic array operations in Java, the bookkeeping to support a Fortran-like data representation can be done in Java.

NINJA package from the IBM Research includes a general array classes (e.g. doubleArray) supporting up to seven- dimensional arrays, with derived classes for one-, two-, or three-dimensional arrays (e.g. doubleArray1D, doubleArray2D, and doubleArray3D). NINJA supports a number of subarray methods and puts a great deal of focus on Index and Range objects. Other methods are included for transpose, permuteAxes, and reshape, but remain specific to arrays and array characteristics.

The NINJA approach is most similar to JCArray, going so far as to allow that there may be a column-major (i.e. Fortran-like) internal representation, but not for enough to allow that it be specified as such. The access and manipulation methods are very similar to JCArray.

6. The Java Grande Proposed Array Class

The Numerics Working Group of the Java Grande Forum is currently completing a proposed standard Array package for Java[5]. Many of the considerations that influenced JCArray have been included in the proposal, including the ability to present array data in column-major order at the Native level. Although it is unlikely the Java Array standard will include any of the special array forms considered in JCArray, a future version of JCArray -- which will implement the Java Array package when it is included in the Java standard -- will continue to connect the best of Java portability to the best native array mathematics on earth.

REFERENCES

1. R.F.Boisvert, J.Hicklin, B.Miller, C.Moler, R.Pozo, K.Remington, P.Webb, *JAMA: A Java Matrix Packag,* National Institute of Standards and Technology and The MathWorks, Inc., 1998. http://math.nist.gov/javanumerics/jama/

2. J.Brophy, *JNL 1.0 – A Numerical Library for Java,* Visual Numerics, Inc., 1998, http://www.jni.com/products/wpd/jnl/

3. J.E.Moreira, S.P.Midkiff, M.Gupta, *A Standard Java Array Package for Technical Computing,* Tech. Rep. 21369, IBM Research Division, 1998, http://www.research.ibm.com/ninja

4. E.Anderson et al, *LAPACK Users Guide, Second Edition,* Society for Industrial and Applied Mathematics, 1995.

5. J.E.Moreira, et al, *JSR - Java Array Package,* Java Numerics Working Group, The Java Grande Forum, 1999. http://www.javagrande.org/