The jCrunch™ Java Numerical Libraries

William N. Reynolds
Least Squares Software LLC
PO Box 91405
Albuquerque, NM 87199
bill@leastsquares.com
http://www.leastsquares.com

1. Introduction

The Java™ programming language has been greeted with a great degree of enthusiasm by the IT community. It's ease of use, portability, robustness and object-oriented design allow for rapid development of reusable application components. Java allows the developer to web-enable his applications – easily distributing his code to the next cubicle or to the next continent. The promise of distributed, web-enabled, applications is especially appealing to the fields of technical computation such as engineering, quantitative finance, research and simulation.

However, like any young technology, Java is still lacking the tools that developers expect in a mature programming environment. In particular, the lack of mathematical routines in Java is particularly troubling for the developer doing technical computations. While the underlying mathematical technology has been well established for over forty years (indeed, it was one of the original forces driving the development of the digital computer) the effort to re-implement this technology for Java has been prohibitive.

A number of initiatives have been undertaken to address this issue [1,2,3]. Most notable among these is the Java Grande Forum; however, four years after introduction of Java, there is still a pronounced absence of high quality numerical libraries.

2. The jCrunch Numerical Libraries

The jCrunch project was initiated to address this lack of high quality numerical libraries. The jCrunch libraries are set of proprietary, commercial-grade Java numerical libraries. Our approach is very similar to the F2J project at UTK [1]. Rather than "reinvent the wheel" by re-implementing all known numerical algorithms, we have instead decided to leverage the existing body of Fortran numerical libraries by implementing an automatic Fortran-to-Java translator.

This approach has advantages and disadvantages. The advantages are obvious: if the translator is done well, familiar, high-quality, robust numerical algorithms immediately become available in Java. However, this is offset by some significant disadvantages:

1. The translated routines will by necessity have "Fortran-like" interfaces – requiring that the programmer keep track of many messy details (such as array leading dimensions) that would be unnecessary in a modern object-oriented implementation.

2. The generated code is unlikely to be very efficient, since the original code was written with

We address the second issue in several ways. First, by an appropriate choice of data storage and by designing the translator such that expensive operations, such as object instantiation, are minimized, very efficient Java code can be generated.

Second, we have extended the translator to generate Java Native Interface (JNI) wrappers that call out to native Fortran libraries. These libraries may reside in either a shared library, or a browser plugin (using the very similar JRI). Using these Native Objects, near-Fortran execution speeds may be achieved.

Finally, we have implemented a set of *Native Broker* classes that dynamically select either the pure Java or JNI objects at runtime. All three classes, pure Java, JNI and Native Broker, expose the same interface. In addition, the Native Broker provides methods for setting and getting the current run mode, as well as for determining why a given run mode was or was not selected.

3. The jCrunch API

Every jCrunch object has a consistent calling interface that allows it to be easily integrated into a numerical application. Since jCrunch objects are Java classes that are translations of subroutines and functions from the Fortran programming language, each object encapsulates a particular operation on member variables that correspond to Fortran arguments. When a jCrunch object is a translated Fortran function, a return value is also computed. Translations of Fortran subroutines return void.

Using a jCrunch object consists of four steps:

1. Instantiating an object.
2. Initializing the object with arguments.
3. Executing the computation.
4. Retrieving the result.

Although the first three steps are conceptually separate and can each be done with a separate line of code, convenience methods are provided that can combine some of these steps.

Instantiating the Object

All of the jCrunch objects are provided with an argumentless constructor. For example, to instantiate a Dgesvd object (this is a translation of the Lapack [5] routine `dgesvd`, which executes a Singular Value Decomposition, or SVD), we use:

```
Dgesvd svd = new Dgesvd();
```

This instantiates a new object, `svd`, which is ready to be assigned arguments.

Initializing the Object with Arguments

Every jCrunch object has a public `setArgs()` method for assigning values to its member arguments. In the case of `Dgesvd`, there are 14 arguments. We assign these arguments as:

```
JCDMatrix u = new JCDMatrix(n, n);
JCDMatrix vt = new JCDMatrix(m, m);

svd.setArgs(jobu, jobvt, a.rowCount(), a.colCount(),
   a, a.getLDim(), s, u, u.getLDim(), vt,
   vt.getLDim(), new double[5*n], 5*n, info);
```

The `setArgs` method has been overloaded to allow offsets to be passed with arrays to allow computations to be performed on subarrays. This is described in section 5.

Executing the Computation

The actual computation is invoked with the object's `run()` method. In the event that the object represents a function computation, this method will return a result, otherwise, this method is void, and results are obtained only from the new values of the member variables.

Retrieving the Result

For a function object, the most interesting result is that returned by the `run()` method. More complicated computations return data by modifying the member arguments. Which arguments return interesting data varies from routine to routine; however, jCrunch provides `getArg#()` methods for retrieving all of them. The `getArg#()` methods are numbered up from 0. For example, the new value of the `jobu` argument could be obtained from the `svd` object, after its `run()` method was called, using the `getArg0()` method. We could also retrieve the data for the VT matrix using the `getArg9()` method.

```
 String nuJobu = svd.getArg0();
 double vtData[] = svd.getArg9();
```

Convenience Methods

Convenience methods are provided for instantiating objects with arguments or for running methods with arguments. For example, the first two steps above could have been done with the following constructor:

```
JCDMatrix a = new JCDMatrix(n, m);

JCDMatrix u = new JCDMatrix(n, n);
JCDMatrix vt = new JCDMatrix(m, m);

Dgesvd svd = new Dgesvd(jobu, jobvt, a.rowCount(), a.colCount(),
   a, a.getLDim(), s, u, u.getLDim(), vt,
   vt.getLDim(), new double[5*n], 5*n, info);
```

Alternatively, one could eliminate the `setArgs()` by using the `run()` method with arguments:

```
Dgesvd svd = new Dgesvd();
```

4. Interfaces and Object Arguments

Every jCrunch object implements a particular interface that shows exactly what its arguments are and what its run method returns. Every interface is named **JCInterface_*<objectCode>*.** An object codes is a string of characters representing both the objects return value and the arguments to `setArgs()`. Return value codes are set apart from the argument codes with an underscore character. For example, `Dgesvd` implements the following interface: `JCInterface_v_ssiiEiEEiEiEii.` The type code is the first letter of the Java type for a scalar, or the capitalized next alphabetical letter for an array. The following table gives the mapping of codes to argument types:

Code	Argument Type
i	`int`
J	`int[]`
f	`float`
G	`float[]`
d	`double`
E	`double[]`
b	`boolean`
C	`boolean[]`
s	`String`
O	Begin passed object
P	End passed object
v	`void` return type

The O and P codes bracket arguments that themselves implement jCrunch interfaces. In this way, one can pass objects that perform computations themselves. An example would be an object representing a fitness in an optimization routine. Note that one can quickly interface legacy objects to a jCrunch routine simply by implementing the appropriate jCrunch interface.
For example, the `Dgees` object implements the following interface:

```
JCInterface_v_ssOb_ddPiEiiEEEiEiCi
```

This means it takes as an argument a jCrunch object that returns a boolean and takes two double arguments.

Methods mandated by the jCrunch interfaces are `run(), setArgs()` (with array offsets, as described in section 5), `getArg#()` and `clone().` All jCrunch objects are Cloneable and Serializable. The `clone()` method will return a deep copy of the object, returning clones of any member objects.

5. Array Storage Conventions

Column-Major Order

In general, jCrunch follows the Fortran convention for matrix storage, namely 2-Dimensional matrices are stored in a 1-Dimensional array in column-major order. This means that the following matrix:

$$two\_D\_Matrix_{ij} = one\_D\_Array_{i+j*ld}$$

Here, `ld` is the **leading dimension**, or column length, of the matrix. Note that this formula assumes that matrices are accessed with elements indexed from 0. Fortran uses a 1 based offset. To access array elements indexed with row and column offsets `r` and `c`, use the following:

$$two\_D\_Matrix_{ij} = one\_D\_Array_{(i-r)+(j-r)*ld}$$

Offsets

Subarrys can be accessed by providing the dimension of the subarray, the leading dimension of the parent array and the offset within the parent array. For example, we would like to access the submatrix marked by X's in this 4x4 array:

```
O0 O4 O6 O8
O1 X0 X2 O9
O2 X1 X3 OA
O3 O5 O7 OB
```

which is stored in a 1D Array, O, as

```
O = { O0 O1 O2 O3 O4 X0 X1 O5 O6 X2 X3 O7 O8 O9 OA OB };
```

Using a *global* offset of 5 (the zero-based index of the first element of the submatrix, X0) and a leading dimension of 4, the column length of the parent matrix, the submatrix X could be accessed as:

$$X_{ij} = O_{i + 5 + j*4}$$

then X0 = X[0][0] becomes O[0 + 5 + 0*4] = O[5] (element X0) and X3 = X[1][1] becomes O[1 + 5 + 1*4] = O[10] (element X3).

Offsets and jCrunch

Every array argument that is passed to a jCrunch object can be treated as a subarray. For this reason, every method for a jCrunch object that takes an array argument is overloaded to take array arguments plus offsets (actually, it's the other way around - the interfaces require methods with offsets, the offset-less methods are the overloads). Using a `Dgesvd` object named `svd` as an example, the `setArgs()` method is usually used as:

```
svd.setArgs(job, jobvt, m, n, a, lda,
   s, u, ldu, vt, ldvt, work, lwork, info);
```

Note that every matrix argument has an associated leading dimension. To pass subarrays there is a second `setArgs()` method that uses an array offset:

jCrunch.array

As a first step toward a high-level, object-oriented interface, we have implemented a set of Matrix objects which perform all of the access and offset bookkeeping [4]. To pass data to a jCrunch object from a jCrunch array class, use its `data` member and `getLDim()` and `getGOffset()` methods:

```
JCDMatrix A = new JCDMatrix(m, n);

for(i=0;i<A.getRowCount();i++)
  for(j=0;j<A.getColCount();j++)
    A.set(i,j) = f(i,j);

svd.setArgs(jobu, jobvt, A.getRowCount(),
  A.getColCount(), A.data, A.getGOffset(),
  A.getLDim(), singularValues, 0, U.data,
  U.getGOffset(), U.getLDim(), VT.data,
  VT.getGOffset(), VT.getLDim(), work, 0,
  workDim, info);
```

The jCrunch Array class has a number of other features, please refer to [4] in this volume for details.

6. Object Brokers - Transparently Accessing Native Code

Because native speed is of such paramount importance to the numerical programmer, a *NativeBroker*™ package is available to complement the Java objects in the basic jCrunch packages. Each NativeBroker package transparently allows the jCrunch objects to call native code whenever it is available. Native code can be either a browser plugin (currently plugins are available for Netscape™ and Internet Explorer™) or a shared library (or DLL). In the event that the native code is not available, the pure Java object will be instantiated and used to perform the computation. All of this occurs without intervention by the user or developer.

Using Native Brokers

Using a NativeBroker object is simple - the programmer simply changes the `import` statement that imports the jCrunch object to use the NativeBroker version of the object. For example, a code that was using the Java object, `Dgesvd` would use the import statement:

```
import com.leastsquares.jcrunch.lapack.Dgesvd;
```

To use the NativeBroker object, this statement is changed to:

```
import com.leastsquares.jcrunch.lapack.nativebroker.Dgesvd;
```

and the class recompiled. For each jCrunch object, there is a corresponding NativeBroker object. Note

the NativeBroker, which then throws a Java exception. However, runtime errors not properly handled by the native code can cause the JVM to crash. For this reason, during development and testing, DLL's should be disabled to exploit the robustness and flexibility of the JVM.

7. Current Status

The first jCrunch library, jCrunch Lapack™, is at the time of this writing, in beta testing. We anticipate general release 3Q, 1999. Additional libraries, including optimizers, differential equation solvers and Fourier transform routines are in development. Currently, there are no "high-level" object wrappers for the translated routines. It is our hope to release these following the release of the basic translations. We have also not released any classes for handling complex numbers. While we have implemented complex numbers in the same way Fortran does, as arrays containing alternating real and imaginary components, we have deferred releasing any complex class libraries until the Java Grande forum settles on a specification for a Complex class.

References

1) The University of Tennessee at Knoxville's Fortran-to-Java project: http://www.cs.utk.edu/f2j/

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

3) The JavaNumerics website provides a variety of links to Java numerical initiatives: http://math.nist.gov/javanumerics

4) David S. Dixon, *JCArray, the jCrunch Java Array Classes,* this volume.

5) E. Anderson, *et. al. LAPACK User's Guide – Release 2.0* SIAM Press, Philadelphia, 1995.

Appendix A: Sample Application

We include here a listing of a sample "high-level" interface to the jCrunch Lapack class Dgesvd.

```
import com.leastsquares.jcrunch.lapack.Dgesvd;
import com.leastsquares.jcrunch.array.JCDMatrix;
import com.leastsquares.jcrunch.array.JCDVector;
```

```
 *  A = U*S*VT
 * <P>
 * A is the original matrix of dimension mRows by nCols. U is an
orthogonal
 * matrix of dimension mRows by mRows. S is a diagonal matrix of
dimension mRows
 * by nCols, the diagonal elements of S are A's singular values. VT is
the transpose of
 * an orthogonal matrix of dimension nCols by nCols.
 *
 * The routine takes as input the matrix A. By default, the singular
 * values are returned in member array <code>singularValues</code>,
 * the matrices U and VT are returned as DMatrix members.  Computation
 * is executed by the <code>run</code> method.  Optionally, an
 * argument can be passed to the constructor specifying whether or how
 * much of the matrices U and VT are computed.  The success or failure
 * of the computation is returned in the member <code>info</code>,
 * which is 0 for success and nonzero otherwise. See Dgesvd for
 * details.
 *
 * @see Dgesvd
 * @see tDSvd.java
 * @see DMatrix
 * @author David S. Dixon and Bill Reynolds, Appstar L.L.C.
 * @version $Id: DSvd.java,v 1.1 1998/09/12 16:51:03 bill Exp bill $ */

public class DSvd
{
 protected String id = "DSvd";
 protected String version = "1.0";
 protected String ident = id + " " + version;

 public static final int ALL_U_ALL_VT        = 0  ;
 public static final int ALL                 = 0  ;
 public static final int ALL_U_SOME_VT       = 1  ;
 public static final int ALL_U_OVERWRITE_VT  = 2  ;
 public static final int ALL_U_NO_VT         = 3  ;
 public static final int SOME_U_ALL_VT       = 4  ;
 public static final int SOME_U_SOME_VT      = 5  ;
 public static final int SOME_U_OVERWRITE_VT = 6  ;
 public static final int SOME_U_NO_VT        = 7  ;
 public static final int OVERWRITE_U_ALL_VT  = 8  ;
 public static final int OVERWRITE_U_SOME_VT = 9  ;
 public static final int OVERWRITE_U_NO_VT   = 10 ;
 public static final int NO_U_ALL_VT         = 11 ;
 public static final int NO_U_SOME_VT        = 12 ;
```

```
  * <code>A</code>.
  *
  * @see JCDMatrix */
 public JCDMatrix A ;


 /** Array of length Min(mRows, nCols) containing the singular values
     of <code>A</code>.
  *
  */
 public double singularValues[];

 /** The orthogonal matrix U. By default, this is  an mRows by mRows
matrix. If a
  * SOME_U instantiation option is specified, then only the first
Min(mRows, nCols)
  * columns are computed, and the matrix dimension is then  mRows by
Min(mRows, nCols).
  * If a NO_U or OVERWRITE_U instantiation option is specified, then
this matrix has
  * zero dimension and contains no data.
  *
  * @see JCDMatrix
  */
 public JCDMatrix U ;

 /** The orthogonal matrix VT. By default, this is  an nCols by nCols
matrix. If a
  * SOME_VT instantiation option is specified, then only the first
Min(mRows, nCols)
  * rows are computed, and the matrix dimension is then  nCols by
Min(mRows, nCols).
  * If a NO_VT or OVERWRITE_VT instantiation option is specified, then
this matrix has
  * zero dimension and contains no data.
  *
  * @see JCDMatrix
  */
 public JCDMatrix VT ;

 /** Return status indicator. A value of 0 indicates a successful
  * computation. A nonzero return implies a problem. See  Dgesvd.info.
  *
  * @see Dgesvd
  */
 public int info ;
```

```
 private int lDimVT ;
 private int nMin ;
 private int nMax ;
 private int workDim;

 // These are public, since users may need to get under the hood (to
 // access the work arrays, for example).
 /** The interface to the Lapack routine DGESVD. If technical
information is needed on
  * the computation, it may be accessed here.
  *
  * @see Dgesvd
  */
 public Dgesvd dgesvd ;


 private String jobu ;
 private String jobvt ;

 /** Construct and SVD object for the input matrix A. Upon invocation
     of the <code>run()</code> method, all of <code>U</code> and
  *  <code>VT</code> are computed  and returned.
  *
  * @param A Matrix to be decomposed.  */
 public DSvd(JCDMatrix A)
 {
  DSvd(A, ALL_U_ALL_VT) ;
 }

 /** Construct and SVD object for the input matrix A.  The number of
  * singular vectors returned and their storage schemes is specified
  * in the argument returnFlag. There are fifteen possible options,
  * each a member of DSvd. They are named using the format
  * OPTION_U_OPTION_VT. The possible options are:
  *
  * <P>
  * <dl>
  * <dt> ALL <dd> Compute all rows and columns of the matrix. Return as
DMatrix.
  * <dt> SOME <dd> Compute the first Min( mRows, nCols) columns of
  * <code>U</code> or rows of <code>VT</code>. Return as  DMatrix.
  * <dt> OVERWRITE <dd> Compute the first Min( mRows, nCols) columns of
  * <code>U</code> or rows of <code>VT</code>. Return in
  * <code>A</code>. Member is returned with dimensions 0.
  * <dt> NONE. <dd> No rows or columns are computed. Member is returned
with dimensions 0.
```

```
 * <dt> DSvd.SOME_U_ALL_VT <dd> <br>
 * <dt> DSvd.SOME_U_SOME_VT <dd> <br>
 * <dt> DSvd.SOME_U_OVERWRITE_VT <dd> <br>
 * <dt> DSvd.SOME_U_NO_VT <dd> <br>
 * <dt> DSvd.OVERWRITE_U_ALL_VT <dd> <br>
 * <dt> DSvd.OVERWRITE_U_SOME_VT <dd> <br>
 * <dt> DSvd.OVERWRITE_U_NO_VT <dd> <br>
 * <dt> DSvd.NO_U_ALL_VT <dd> <br>
 * <dt> DSvd.NO_U_SOME_VT <dd> <br>
 * <dt> DSvd.NO_U_OVERWRITE_VT <dd> <br>
 * <dt> DSvd.NO_U_NO_VT <dd> (synonym: DSvd.NONE)
 * </dl>
 *
 * @param A Matrix to be decomposed.
 * @param returnFlag Flag specifying whether and how much of U and
 VT to compute. */

public DSvd(JCDMatrix A, int returnFlag)
{
 DSvd(A, returnFlag);
}

/** Get the current run mode of this DSVD object. Determines which
    submatrices will be computed by this object upon invocation of
    the <code>run</code> method. */

public int getRunMode() { return this.returnFlag; }

/** Set the current run mode of this DSVD object. Determines which
    submatrices will be computed by this object upon invocation of
    the <code>run</code> method. */

public void setRunMode(int returnFlag) { setupDSvd(returnFlag); }

void DSvd(JCDMatrix AArg, int returnFlag)
{
 A = AArg;
 mRows = A.getRowCount() ;
 nCols = A.getColCount() ;
 lDim = A.getLDim();
 dgesvd = new Dgesvd();

 setupDSvd(returnFlag);

 U = new JCDMatrix(mRowsU, nColsU);
 VT = new JCDMatrix(mRowsVT, nColsVT);
```

```
}

/** Execute the SVD on the input matrix. Compute the orthogonal
 *  matrices U and VT
 *
 */

public void run()
{
 dgesvd.run();
 jobu = dgesvd.getArg0();
 jobvt = dgesvd.getArg1();
 mRows = dgesvd.getArg2();
 nCols = dgesvd.getArg3();
 A.data = dgesvd.getArg4();
 lDim = dgesvd.getArg5();
 singularValues = dgesvd.getArg6();
 U.data = dgesvd.getArg7();
 VT.data = dgesvd.getArg9();
 work = dgesvd.getArg11();
 workDim = dgesvd.getArg12();
 info = dgesvd.getArg13();
}


private void setupDSvd(int returnFlag)
{
 this.returnFlag = returnFlag;

 if (mRows < nCols)
  {
   nMin = mRows ;
   nMax = nCols ;
  }
 else
  {
   nMin = nCols ;
   nMax = mRows ;
  }

 workDim = 3*nMin+nMax;
 if (workDim < (5*nMin-4))
  workDim = 5*nMin-4;

 if (workDim < 1)
  workDim = 1;
```

```
 mRowsU = mRows;
 nColsU = mRows;
 break;
case  SOME_U_ALL_VT:
case  SOME_U_SOME_VT:
case  SOME_U_OVERWRITE_VT:
case  SOME_U_NO_VT:
 jobu = "s";
 mRowsU = mRows;
 nColsU = nMin;
 break;
case  OVERWRITE_U_ALL_VT:
case  OVERWRITE_U_SOME_VT:
case  OVERWRITE_U_NO_VT:
 jobu = "o";
 mRowsU = 0;
 nColsU = 0;
 break;
case  NO_U_ALL_VT:
case  NO_U_SOME_VT:
case  NO_U_OVERWRITE_VT:
case  NO_U_NO_VT:
 jobu = "n";
 mRowsU = 0;
 nColsU = 0;
 break;
}

// If jobvt = "a", VT is nCols x nCols.
// If jobvt = "s", VT is nMin x nCols, where nMin = min(mRows, nCols)

switch(returnFlag)
{
case        ALL_U_ALL_VT:
case       SOME_U_ALL_VT:
case  OVERWRITE_U_ALL_VT:
case         NO_U_ALL_VT:
 jobvt = "a";
 mRowsVT = nCols;
 nColsVT = nCols;
 break;
case        ALL_U_SOME_VT:
case       SOME_U_SOME_VT:
case  OVERWRITE_U_SOME_VT:
case         NO_U_SOME_VT:
 jobvt = "s";
```

```
   case       SOME_U_NO_VT:
   case  OVERWRITE_U_NO_VT:
   case         NO_U_NO_VT:
    jobvt = "n";
    mRowsVT = 0;
    nColsVT = 0;
    break;
   }
 }


}
```