CONCURRENCY: PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper., Vol. **10**(11–13), 1117–1129 (1998)

Developing numerical libraries in Java

RONALD F. BOISVERT[1]*, JACK J. DONGARRA[2], ROLDAN POZO[1], KARIN A. REMINGTON[1] AND G. W. STEWART[1,3]

[1]*Mathematical and Computational Sciences Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA*
(e-mail: {boisvert,pozo,karin}@nist.gov)

[2]*Computer Science Department, University of Tennessee at Knoxville, Knoxville, TN 37996, USA and Oak Ridge National Laboratory, Oak Ridge, TN, USA*
(e-mail: dongarra@cs.utk.edu)

[3]*Department of Computer Science, University of Maryland, College Park, MD 20742, USA*
(e-mail: stewart@cs.umd.edu)

SUMMARY

The rapid and widespread adoption of Java has created a demand for reliable and reusable mathematical software components to support the growing number of computationally intensive applications now under development, particularly in science and engineering. In this paper we address practical issues of the Java language and environment which have an effect on numerical library design and development. Benchmarks which illustrate the current levels of performance of key numerical kernels on a variety of Java platforms are presented. Finally, a strategy for the development of a fundamental numerical toolkit for Java is proposed and its current status is described. ©1998 John Wiley & Sons, Ltd.

1. INTRODUCTION

Mathematical software libraries were introduced in the 1960s both to promote software reuse and as a means of transferring numerical analysis and algorithmic expertise to practitioners. Many successful libraries have since been developed, resulting in a variety of commercial products, as well as public repositories of reusable components such as *netlib* and the Guide to Available Mathematical Software[1].

Library users want components which run fast, are easily moved among computing platforms, invariably produce the right answer, and are easy to understand and integrate with their applications. Thus, efficiency, portability, reliability and usability are of primary concern to library developers. Unfortunately, these properties are often competing, portability and reliability often taking a toll on performance, for example. Hence, the development of high-quality portable mathematical software libraries for widely differing computing environments continues to be a challenging task.

Java technology[2,3] is leading to a revolution in network-based computing. One of the main reasons for this is the promise of new levels of portability across a very wide range of platforms. Java is only beginning to affect the scientific computing world. Some of

*Correspondence to: R. F. Boisvert, Mathematical and Computational Sciences Division, Information Technology Laboratory, National Institute of Standards and Technology, Gaithersburg, MD 20899, USA. (e-mail: boisvert@nist.gov)

Contract grant sponsor: US Army Research Office; Contract grant number: DAAH04-95-1-0595.

CCC 1040–3108/98/111117–13$17.50 *Received 28 February 1998*
©1998 John Wiley & Sons, Ltd. *Revised 27 March 1998*

the barriers to its adoption in this domain are the perception of inadequate efficiency, language constraints which make mathematical processing awkward, and lack of a substantial existing base of high-quality numerical software components.

In this paper we assess the suitability of the Java language for the development of mathematical software libraries. We focus on features of the language and environment which may lead to awkward or inefficient numerical applications. We present case studies illustrating the performance of Java on key numerical kernels in a variety of environments. Finally, we outline the Java Numerical Toolkit[*] (JNT), which is meant to provide a base of computational kernels to aid the development of numerical applications and to serve as a basis for reference implementations of community defined frameworks for computational science in Java.

2. NUMERICAL COMPUTING IN JAVA

Java is both a computer language and a runtime environment. The Java language[2] is an object-oriented language similar to, but much simpler than, C++. Compilers translate Java programs into bytecodes for execution on the Java Virtual Machine (JVM)[3]. The JVM presents a fixed computational environment which can be provided on any computer platform.

The resulting computing environment has many desirable features: a simple object-oriented language, a high degree of portability, a runtime system that enforces array bounds checking, built-in exception handling, and an automated memory manager (supported by a garbage collector), all of which lead to more reliable software.

In this Section we review key features of the Java language, assessing their effect on both performance and convenience for use in numerical computing. In doing this we point out a number of deficiencies in the language. It is important to note, however, that many of Java's desirable features, such as its portability, are derived from the JVM rather than the language itself. Other languages can be compiled into Java bytecodes for execution by the JVM, and several compilers for Java extensions are under development.[†] Precompilers which translate other languages, including C++, into pure Java are also under development. If such tools achieve a high enough level of maturity and support, they too can provide the basis for a Java-based development environment for scientific computing.

2.1. Arithmetic

The idea that results produced on every JVM should be bitwise identical[2] on all platforms threatens the usability of Java for high-performance scientific computing. While there may be some scientific applications where such certainty would be useful, its strict implementation could severely degrade performance on many platforms. Systems with hardware that supports extended precision accumulators (which enhance accuracy) would be penalized, for example, and certain code optimizations (including runtime translation to native code) would be disallowed.

It is also unfortunate that Java has not provided programmers with full access to the facilities of IEEE floating-point arithmetic. Programmers do not have control of the rounding mode, for example (although this is rarely available in high-level languages). Also, the

[*]`http://math.nist.gov/jnt/`.
[†]Many of these are listed at `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`.

ability to (optionally) have floating-point arithmetic throw exceptions (on the generation of a NaN, for example), would simplify coding and debugging.

2.2. Complex arithmetic

Complex arithmetic is essential in scientific computing. Java does not have a complex data type, although this is not a fatal flaw since new types are easy to define. However, since Java does not support operator overloading, one cannot make such types behave like the primitive types `float` or `double`. More important than syntactic convenience, however, is that not having complex arithmetic in the language can severely affect the performance of applications. This is because compilers, as well as the JVM, will be unable to perform conventional optimizations on complex arithmetic because they are unaware of the semantics of the class.

Since complex arithmetic is so pervasive it is necessary to establish community consensus on a Java interface for complex classes[4].

2.3. Memory model

Perhaps the biggest difference in developing numerical code in Java rather than in Fortran or C results from Java's memory model. Numerical software designers typically take information about the physical layout of data in memory into account when designing algorithms to achieve high performance. For example, LINPACK[5] used column-oriented algorithms and the Level 1 BLAS in order to localize memory references for efficiency in paged virtual memory environments. LAPACK[6] used block-oriented algorithms and the Level 2 and 3 BLAS to localize references for efficiency in modern cache-based systems. The ability to do this hinged on the fact that Fortran requires two-dimensional arrays be stored contiguously by columns.

Unfortunately, there is no similar requirement for multidimensional arrays in Java. Here, a two-dimensional array is an array of one-dimensional arrays. Although we might expect that elements of rows are stored contiguously, one cannot depend upon the rows themselves being stored contiguously. In fact, there is no way to check whether rows have been stored contiguously after they have been allocated. The row-orientation of Java arrays means that, as in C, row-oriented algorithms may be preferred over column-oriented algorithms. The possible non-contiguity of rows implies that the effectiveness of block-oriented algorithms may be highly dependent on the particular implementation of the JVM as well as the current state of the memory manager.

The Java language has no facilities for explicitly referencing subvectors or subarrays. In particular, the approach commonly used in Fortran and C of passing the address of an array element to a subprogram which then operates on the appropriate subarray does not work.*

2.4. Java's vector class

Despite its name, Java's vector class `java.util.Vector` is not really appropriate for numerics. This class is similar in spirit to those found in the Standard Template Library

*In C one would explicitly pass an address to the procedure, but address arithmetic does not exist in Java. In Fortran one passes an array element, which amounts to the same thing since all parameters are passed by address, but scalars are passed by value in Java.

of C++, that is, they are merely containers which represent objects logically stored in contiguous locations.

Because there is no operator overloading, access to vectors via this class must be through a functional interface. Also, `Vector` stores generic `Objects`, not simple data types. This allows a vector to contain heterogeneous data elements – an elegant feature, but it adds overhead, and, unfortunately, complicates its use for simple data types. To use a vector of double, for example, one needs to use Java's wrapper `Double` class and perform explicit coercions.

Consider the difference between using native Java arrays:

```
double x[] = new double[10];     // using native Java arrays
double a[] = new double[10];
...
a[i] =  (x[i+1] - x[i-1]) / 2.0;
```

and Java's `Vector` class:

```
Vector x = new Vector(10);     // using Java's Vector class
Vector a = new Vector(10);
...
a.setElement(i, new Double((((Double) x.ElementAt(i+1)).doubleValue()
      - ((Double) x.ElementAt(i-1)).doubleValue()) / 2.0);
```

Deriving a `VectorDouble` class from `Vector` which performed these coercions automatically would clean the code up somewhat, but would introduce even more overhead by making each reference `x[i]` a virtual functional call.

2.5. I/O facilities

Java's JDK 1.1 defines over 40 I/O classes, many of them with only subtle differences, making it difficult to choose the right one for a given task. For example, the average Java user may not immediately recognize the difference between parsing tokens from strings via `StringTokenizer (String)` and `StreamTokenizer (StringReader(String))`.

Ironically, despite these numerous I/O classes there is little support for reading floating point numbers in exponential notation. Even if one resorts to low-level parsing routines to read floating point numbers, the internal class `java.io.StreamTokenizer` parses '2.13e+6' as four separate tokens ('2.13', 'e', '+', '6.0'), even when the `parseNumbers()` flag is set. Furthermore, no formatted output facilities are provided, making it very difficult to produce readable tabulated output.

2.6. Other inconveniences

A number of other conveniences which library developers have come to depend upon are not available in Java. *Operator overloading*, which would be particularly useful for user-defined matrix, vector, and array classes, as well as for complex arithmetic, would be quite useful. Finally, *templates*, such as those in C++, would eliminate the need to create duplicate functions and classes for double, float, complex, etc. The loss of compile-time polymorphism can also lead to inefficiencies at runtime. While these omissions are not fatal, they significantly increase the burden of numerical library developers.

Extensions to Java which provide such facilities are under development by various groups, implemented as either precompilers producing pure Java, or as compilers for the JVM.* This may provide a reasonable approach for a Java-centric development environment for scientific computing.

3. JAVA PERFORMANCE ON NUMERICAL KERNELS

For numerical computation, performance is a critical concern. Early experience with Java interpreters has led to the common perception that applications written in Java are slow. One can get good performance in Java by using native methods, i.e. calls to optimized code written in other languages[7]. However, this comes at the expense of portability, which is a key feature of the language. Advances in just-in-time (JIT) compilers, and improvements in Java runtime systems, have changed the landscape in recent months, suggesting that Java itself may indeed be suitable for many types of numerical computations. In this Section we provide a variety of benchmarks which provide a brief glimpse of the current performance of Java for simple numerical kernels.

3.1. Run-time array bounds checking

Runtime array bounds checking is a requirement of the JVM, and improved reliability of applications makes this feature very desirable. Fortunately, this does not necessarily imply significant performance degradation in practice. Modern processors have the ability to overlap index checks with other computation, allowing them to cost very little. Experiments performed in C++ using the NIST Template Numerical Toolkit[8] on a Pentium Pro with the Watcom C++ compiler (version 10.6) show that array bounds checking can add as little as 20% overhead.

3.2. Elementary kernels

Timings for several BLAS 1 kernels with various unrolling strategies in several environments are presented in Table 1. The baseline kernel `daxpy` (unroll 1) is written with no optimizations, i.e.

```
public static final void daxpy(int N, double a, double x[], double y[]) {
   for (int i=0; i<N ; i++) y[i] += a*x[i]; }
```

The unroll 4 variant of daxpy uses the kernel

```
     y[i  ] += a * x[i  ];  y[i+1] += a * x[i+1];
     y[i+2] += a * x[i+2];  y[i+3] += a * x[i+3];
```

while the unroll 4-inc variant uses

```
     y[i] += a * x[i]; i++;  y[i] += a * x[i]; i++;
     y[i] += a * x[i]; i++;  y[i] += a * x[i]; i++;
```

The latter can provide performance improvements if the JVM's `inc` opcode is used. The `ddot` schemes are similar.

*See `http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`.

Table 1. Performance of BLAS level 1 kernels in various environments. Vector length is 200. Also varied is the depth of loop urollings. Results in MFLOPS

	daxpy				**ddot**		
	Unroll depth				**Unroll depth**		
Environment	1	4	4-inc	8	1	4	8
Pentium II, 266 MHz, Intel BLAS, Win95	96†				193†		
Pentium II, 266 MHz, gcc 2.7.1 -O3, Linux	88.1	134.2	120.0	132.5	147.1	147.1	148.1
Pentium II, 266 MHz, Microsoft SDK 2.0, Win95	45.0	67.0	80.0	81.0	41.0	80.0	81.0
Pentium Pro, 200 MHz, Sun JDK 1.1.3, Linux	10.4	14.6	15.7	14.7	13.5	21.6	22.0
SGI R10000, 194 MHz, java 3.0.1, IRIX 6.2	12.0	15.0	16.7	16.1	14.4	22.0	24.4
SGI R10000, 194 MHz, f77 -O3, IRIX 6.2	128.7	128.9	129.3	133.3	188.4	188.7	186.3

†Actual loop unrolling strategy unknown.

The data in Table 1 show that Microsoft's SDK 2.0, for example, appears to deliver about half the performance of C code for daxpy and ddot on the Pentium II. This is encouraging. The results also indicate that unrolling can favorably affect the performance of numerical kernels in Java, but that the effect varies widely among JVMs. (No JIT is currently available for the Sun JDK 1.1 under Linux.)

3.3. Dense matrix multiply

We next consider a straightforward matrix multiply loop, i.e.

```
for (int i=0; i<L; i++)
  for (int j=0; j<M; j++)
    for (int k=0; k<N; k++)
       C[i][j] += A[i][k] * B[k][j];
```

By interchanging the three `for` loops one can obtain six distinct matrix multiplication algorithms. We consider three which contain row operations in the innermost loop, denoted as (i,j,k), (k,i,j) and (i,k,j) according to the ordering of the loop indices. The first is the loop displayed above; it computes each element of **C** in turn using a dot product. The second sweeps through **C** N times row-wise, accumulating one term of the inner product in each of **C**'s elements on each pass. The third uses a row-wise daxpy as the kernel.

In Table 2 we display the result in MFLOPS for these kernels on a 266 MHz Pentium II using both Java SDK 2.0 under Windows 95 and C compiled with the Gnu C compiler Version 2.7.2.1 under Linux. The C kernels were coded exactly as the loop above, and compiled with the options `-O3 -funroll-loops`. We consider the case $L = N = M = 100$, as well as the case where $L = M = 100$ and $N = 16$. The latter represents a typical rank K update in a right-looking LU factorization algorithm.

In Table 3 we explore the effect of two additional loop optimizations for this kernel in Java on the Pentium II. In the first case we attempt to reduce overhead using one-dimensional indexing. That is, we assign rows to separate variables (e.g. `Ci[j]` rather than `C[i][j]`), while in the second we use both one-dimensional indexing and loop unrolling (to a level of 4).

Efficient implementations of the Level 3 BLAS in Fortran use *blocked algorithms* to make best use of cache. Because the memory model used in Java does not support contiguously stored two-dimensional arrays, there has been some speculation that such algorithms would

Table 2. Performance of matrix multiplication in C and Java. 266 MHz Pentium II using Gnu C 2.7.2.1 (Linux) and Microsoft Java SDK 2.0 (Windows 95). **C** = **AB**, where **A** is $L \times N$ and **B** is $N \times M$. $L = M = 100$. Results in MFLOPS

	Environments			
	Gnu C		**Microsoft Java**	
Loop order	$N = 100$	$N = 16$	$N = 100$	$N = 16$
(i, j, k)	82.2	90.6	20.4	29.1
(k, i, j)	60.4	49.0	11.4	13.9
(i, k, j)	74.1	60.0	7.6	9.4

Table 3. Effect of loop optimizations on matrix multiplication in Java. 266 MHz Pentium II using Microsoft Java SDK 2.0 (Windows 95). **C** = **AB**, where **A** is $L \times N$ and **B** is $N \times M$. $L = M = 100$. Results in MFLOPS

	Loop Optimizations			
	1D indexing		**plus unrolling**	
Loop order	$N = 100$	$N = 16$	$N = 100$	$N = 16$
(i, j, k)	30.4	36.4	38.3	49.2
(k, i, j)	18.2	20.8	22.0	26.0
(i, k, j)	10.4	11.2	15.8	18.5

be less effective in Java. We have implemented a blocked variant of the (i,j,k) matrix multiply algorithm above with two-level blocking: 40×40 blocks with 8×8 unrolled sub-blocks. Figure 1 compares the simplistic (i,j,k) algorithm with its blocked variant for matrices of size 40×40 to 1000×1000 on a 266 MHz Pentium II system running Microsoft Java SDK 2.0 under Windows 95. The performance of the blocked algorithm is far superior to the unblocked algorithm, achieving 82.1 MFLOPS for the largest case. This is only 10% slower than the smallest case, which fits completely in Level 1 cache.

The performance of the blocked algorithm is somewhat dependent on details of the implementation. We observed a subtle tradeoff, for example, between including more temporary variables and the use of explicit indexing. The choice of blocksize also has a big effect; selecting 64×64 blocks only yields 63 MFLOP performance for the 1000×1000 case, for example. This indicates that it may be necessary to have a standard method call which returns the optimal blocksize for the current virtual machine.

These results indicate that Java performance is still inferior to that obtained from C for Level 3 BLAS operations, but that optimized Java can approach half the speed of C. (Note that the highly optimized Intel BLAS ran this kernel (dgemm) at about 190 MFLOPS for $L = M = N = 200$.) Also, relative performance of kernels in Java may be quite different than in C. As expected, however, kernels based on dot products may be preferable in Java, and strategies such as unrolling, one-dimensional indexing and blocking will help, although the strength of the effect will undoubtably be highly dependent on the particular JVM and JIT compiler.

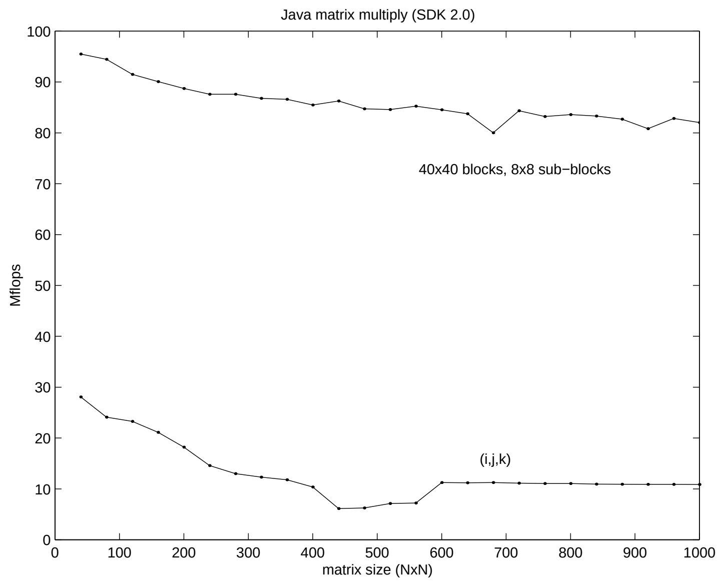

Figure 1. Blocked versus unblocked matrix multiplication in Java

3.4. Sparse matrix–vector multiply

Finally, we consider sparse matrix–vector multiplication based on a sparse coordinate storage format[9]. In particular, we compare an implementation in Java with those based upon the NIST Sparse BLAS Toolkit reference implementation[10] in C on a 266 MHz Pentium II running Windows 95. The test cases, taken from the Harwell-Boeing collection[11,12], represent fairly small sparse matrices, but may provide an indication of the relative performance of these languages on kernels which contain indirect index computations. The results are presented in Table 4. Note that the higher levels of performance for WEST0156 are due to the fact that the matrix is small enough to completely fit in cache. The results indicate that optimized Java code which is rich in indexing such as in sparse matrix operations can perform at from 65% to 75% of the speed of optimized C code.

4. NUMERICAL LIBRARIES IN JAVA

Several development strategies exist for building numerical libraries in Java. First, numerical classes can be coded directly in the language. This is, of course, labor-intensive, and could lead to less than optimal code due to inefficiencies in the language. Nevertheless, several groups have begun to undertake such projects[4,13,14]. A second option is to develop tools to translate existing Fortran libraries into Java[15,16]. While this provides easy access to a wealth of existing software, the severe mismatch between Fortran and Java semantics is likely to lead to converted library source which is unnatural and inefficient. A third option is to use Java's *native methods* facility to provide an interface to existing code in other languages such as Fortran and C. This requires one to develop a Java wrapper to each library routine, although this is far simpler than recoding. The problem here, of course, is that Java's greatest advantage, its portability, is compromised.

Table 4. Performance of sparse matrix–vector multiply in Java and C. 266 MHz Pentium II using Microsoft Java SDK 2.0 and Watcom C 10.6 (Windows 95). Results in MFLOPS

			Environments	
Matrix	**Order**	**Entries**	**Microsoft Java**	**Watcom C**
WEST0156	156	371	33.7	43.9
SHERMAN3	5,505	20,033	14.0	21.4
MCFE	765	24,382	17.0	23.2
MEMPLUS	17,758	126,150	9.1	11.1

In this Section we discuss issues involved in the design of numerical libraries coded directly in the Java language.

4.1. Basic design parameters

A number of elementary design decisions must be made when developing numerical libraries in Java.

- *Precision.* What floating-point precisions should be supported? Most numerical computations are currently carried out in IEEE double precision, and hence, support of double is necessary.
- *Naming.* What naming convention should be used for numerical classes? Should long descriptive names or short less cumbersome names be used? Should `Float` and `Double` explicitly appear in class names to distinguish between precisions, as has been done in early numeric class libraries?
- *Vectors and matrices.* Should native Java arrays be used instead of specialized classes for vectors and matrices? Native Java arrays have the advantage of efficient processing and automatic array bounds checking. If an elementary matrix class is devised, should matrices be represented internally as one-dimensional arrays to ensure contiguity of data for block algorithms? If this is done, how can we provide for efficient access to individual elements of arrays and vectors? (Preliminary experiments with Microsoft SDK 1.1 using a five-point stencil kernel showed that use of explicit get and set methods in a matrix class was five times slower than using native Java arrays.) Should indexing of vectors and matrices be 0-based or 1-based? Should packed storage schemes be supported? One can argue that storage is now so plentiful that for many problems the complexity of packed storage schemes for triangular and symmetric matrices is unnecessary in many cases.
- *Serializable classes.* Java provides a convention that allows for I/O of arbitrary objects. Classes which implement the `Serializable` interface promise to provide standard utilities for input and output of their instances. Should all numeric classes be made `Serializable`?
- *Functionality.* How much functionality should be built into classes? Experience has shown that extensive object-oriented design frameworks tend to restrict usability.

Because the design of object-oriented frameworks for numerical computing is a very difficult undertaking, and elaborate designs may, in fact, limit usability by being too complex and specialized for many users, we propose that a *toolkit* approach be taken to the development of numerical libraries in Java. A toolkit is a collection of 'raw' classes which are unencumbered by most of the trappings of object-oriented computing frameworks. They provide a rich source of numerical algorithms implemented mostly as static methods which need not be explicitly instantiated to be used. For simplicity and efficiency, native Java arrays are used to represent vectors and matrices. A toolkit provides a low-level interface to numerical algorithms similar to what one finds in C and Fortran. Toolkits provide a source of basic numerical kernels and computational algorithms which can be used in the construction of more facile object-oriented frameworks for particular applications[17].

4.2. Interface to the BLAS

Because the Java memory model for multidimensional arrays is different from the Fortran model, some consideration must be given to the meaning and interpretation of function interfaces for matrix/vector kernels like the BLAS which play a key role in any toolkit for numerical linear algebra.

As in C and C++, Java stores matrices by rows, without any guarantee that consecutive rows are actually contiguous in memory. Java goes one step further, however. Because there are no capabilities to manipulate pointers directly, one cannot 'alias' subvectors, or reshape vectors into matrices in a direct and efficient manner.

For example, in Fortran, if the function `SUM(N, X)` sums N elements from a given vector $\mathbf{X}$, then calling it `SUM(K, X(I))` sums the elements $x_i, x_i + 1, \ldots, x_i + K$. Unfortunately, no analogue exists in Java. We must reference subvectors explicitly by describing the whole vector and its offset separately, i.e. `SUM(K, X, I)`.

If we are to provide the same level of functionality as the Fortran and C BLAS then we must provide *several* versions of each vector operation. For example, the functionality of a Fortran BLAS with calling sequence

```
(..., N, X, INCX, ... )
```

would have to be supported by

```
(..., int n, double x[], int xoffset, ... ), and
(..., int n, double A[][], int Arow, int Acol, ... ),
```

the former for a subvector, the latter for part of a column in a two-dimensional array. Thus, a Fortran call to manipulate a subvector such as

```
CALL DAXPY(N, ALPHA, X(I), 1, Y(I), 1)
```

would be realized in Java as

```
BLAS.daxpy(N, alpha, x, i, y, i)
```

whereas a Fortran call to manipulate part of the column of an array such as

```
CALL DAXPY(N, ALPHA, A(I,J), LDA, B(I,J), LDB)
```

would be realized in Java as

```
BLAS.daxpy(N, alpha, A, i, j, B, i, j)
```

One might also want to provide simplified, and more efficient, versions which operated on entire vectors or columns, e.g.

```
(..., double x[], ...) and
(..., double A[][], int Acol, ... ),
```

the former for a vector, the latter for a column in a two-dimensional array.

Similarly, Level 2 and Level 3 BLAS which refer to matrices in Fortran as

```
(..., N, M, A, LDA, ...)
```

would require explicit offsets in Java to support operations on subarrays as in

```
(..., int n, int m, double A[][], int Arow, int Acol, ... )
```

whereas routines which manipulate whole Java arrays need only have

```
(..., double A[][], ... )
```

It is clear that providing efficient and capable linear algebra kernels in Java requires much more coding than in Fortran or C.

4.3. Interfaces

It is also necessary to identify common mathematical operations to be defined as Java interfaces. An interface is a promise to provide a particular set of methods. User-defined objects implementing a well-defined interface can then be operated on by library routines in a standard way.

Interfaces to generic mathematical functions are needed, for example, in order to be able to pass user-defined functions to zero finders, minimizers, quadrature routines, plotting routines, etc. If the following existed,

```
public interface UnivariateFunction {
   double eval(double x);}
```

then instances of user-defined classes implementing UnivariateFunction could be passed as arguments to zero finders, which in turn would use the eval method to sample the function. Many variants of mathematical function interfaces would be required. For example, it would also be necessary to define interfaces for bivariate, trivariate and multivariate functions. It would also be necessary to define interfaces for transformations from R^m to R^n. Versions for complex variables would also be required.

Interfaces are necessary to support iterative methods for the solution of sparse linear systems. These would define standard method invocations for operations such as the application of a linear operator (matrix–vector multiply) and preconditioner application, thus allowing the development of iterative solvers that are independent of matrix representation.

5. THE JAVA NUMERICAL TOOLKIT

In order to promote widespread reuse, community defined standard class libraries and interfaces are needed for basic mathematical operations. To promote the development of such class libraries, we have begun the construction of the Java Numerical Toolkit (JNT). JNT will contain basic numerical functions and kernels which can be used to build more capable class libraries. In particular, the initial version of JNT includes

1. elementary matrix/vector operations (BLAS)
2. dense LU and QR matrix factorizations
3. dense linear systems and least squares problems
4. sparse linear systems using iterative methods
5. elementary and special functions, such as sign and Bessel functions I_0, I_1, J_0, J_1, K_0, K_1, Y_0, Y_1
6. random number generators
7. solution of non-linear equations of a single variable.

The toolkit includes interface definitions to support mathematical functions and linear solvers as in Section 4.3. JNT will also include

1. solution of banded linear systems
2. dense eigenvalue problems
3. support for a variety of specialized matrix formats
4. additional special functions, such as hyperbolic functions, the error function, gamma function, etc.
5. one-dimensional quadrature rules.

The toolkit has been initially developed using the floating-point type `double`. Class and method names will not include the word Double (i.e., double will be the toolkit default). Versions based upon `float` will be defined and developed later if there is sufficient demand.

We are using the initial version of the toolkit to develop prototype user-level classes for numerical linear algebra. These will include explicit dense matrix classes which will contain class variables for matrix factorizations. When a user solves a linear system with such a matrix object, a factorization would automatically be computed if necessary and would be available for future use, without explicit action or knowledge of the programmer.

The Web page for the JNT project is `http://math.nist.gov/jnt/`.

6. CONCLUSIONS

The high level of portability, along with support for GUIs and network-based computing provided by Java is attracting interest from the scientific computing community. The Java language itself provides many facilities needed for numerical computing, but many others are lacking, such as complex arithmetic, operator overloading, a clear memory model, and formatted I/O. These will lead to much additional effort on the part of programmers, and brings the ability to achieve high levels of performance in some areas into doubt. On the other hand, rapid progress is being made in the development of JIT compilers, and the performance level of many Java systems are improving (delivering as much as 25–50% of optimized Fortran and C for key kernels in some cases). A major impediment to quick progress in this area is the lack of basic mathematical software which is plentiful in other environments. The construction of basic numerical toolkits for Java needs to be undertaken to bootstrap the development of more sophisticated numerical applications and to provide a basis for the development of community supported standard numerical class libraries and interfaces.

ACKNOWLEDGEMENTS

This paper is a contribution of the National Institute of Standards and Technology and is not subject to copyright. Certain commercial products are identified in this paper in order to adequately document computational experiments. Identification of such products does not constitute endorsement by NIST, nor does it imply that these are the most suitable products for the task. This work is supported in part by Defense Advanced Research Projects Agency under contract DAAH04-95-1-0595, administered by the US Army Research Office.

REFERENCES

1. R. F. Boisvert, S. Browne, J. Dongarra and E. Grosse, ‘Digital software and data repositories for support of scientific computing’, in N. Adam, B. K. Bhargava and M. Halem (Eds), *Advances in Digital Libraries*, no. 1082 in Lecture Notes in Computer Science, Springer-Verlag, New York, 1996, pp. 61–72.
2. J. Gosling, B. Joy and G. Steele, *The Java Language Specification*, Addison-Wesley, Reading, MA, 1996.
3. T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, Reading, MA, 1996.
4. Java language proposal, Visual Numerics, Inc., 9990 Richmond Ave., Ste. 400, Houston, TX 77042-4548, available at `http://www.vni.com/products/wpd/jnl/JNL/docs/intro.html`, 1997.
5. J. J. Dongarra, C. B. Moler, J. R. Bunch and G. W. Stewart, *LINPACK Users’ Guide*, SIAM, Philadelphia, 1979.
6. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov and D. Sorensen, *LAPACK Users’ Guide*, SIAM, Philadelphia, 1992.
7. A. J. C. Bik and D. B. Gannon, ‘A note on native level 1 BLAS in Java’, *Concurrency: Pract. Exp.*, **9**(11), 1091–1099 (1997).
8. R. Pozo, ‘Template numerical toolkit for linear algebra: High performance programming with C++ and the Standard Template Library’, *Int. J. High Perform. Comput. Appl.*, **11**(3), (1997). Further information about TNT may be found at `http://math.nist.gov/tnt/`.
9. R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1994.
10. K. Remington and R. Pozo, ‘Sparse blas’, Working document of the Basic Linear Algebra Subprograms Technical (BLAST) Forum, available at `http://www.netlib.org/utk/papers/blast-forum.html`.
11. R. F. Boisvert, R. Pozo, K. Remington, R. Barrett and J. J. Dongarra, ‘The Matrix Market: A web resource for test matrix collections’, in R. F. Boisvert (Ed.), *The Quality of Numerical Software: Assessment and Enhancement*, Chapman & Hall, London, 1997, pp. 125–137.
12. I. S. Duff, R. G. Grimes and J. G. Lewis, ‘Sparse matrix test problems’, *ACM Trans. Math. Softw.*, **15**(1), 1–14 (1989).
13. S. Russell, L. Stiller and O. Hansson, ‘PNPACK: Computing with probabilities in Java’, *Concurrency: Pract. Exp.*, **9**(11), 1333–1339 (1997).
14. T. H. Smith, A. E. Gower and D. S. Boning, ‘A matrix math library for Java’, *Concurrency: Pract. Exp.*, **9**(11), 1127–1137 (1997).
15. H. Casanova, J. Dongarra and D. M. Doolin, ‘Java access to numerical libraries’, *Concurrency: Pract. Exp.*, **9**(11), 1279–1291 (1997).
16. G. Fox, X. Li, Z. Qiang and W. Zhigang, ‘A prototype Fortran-to-Java converter’, *Concurrency: Pract. Exp.*, **9**(11), 1047–1061 (1997).
17. M. B. Dwyer and V. Wallentine, ‘A framework for parallel adaptive grid simulations’, *Concurrency: Pract. Exp.*, **9**(11), 1293–1310 (1997).