Algebraic Java classes for numerical optimization

Matthias Schwab and Joel Schroeder

SUMMARY

Our software, Jest, comprises a general and extendible library for numerical optimization for science and engineering, *Jam*, and a particular extension of that framework for seismic image processing, *Jag*. Jest successfully separates optimization and application software without imposing limitations on an application's vector representation, on an application's operator implementation, or on a solver's ability to manipulate vectors and operators. To ensure the compatibility of solver and application, Jest includes a set of simple interfaces that define method invocations for the fundamental mathematical objects of numerical optimization, such as vectors, vector spaces, operators, and solvers. Jest's solvers are implemented in terms of these mathematical objects and consequently possess the generality of the original abstract mathematical algorithm. Jest's seismic imaging library, Jag, demonstrates that Jest incurs no restrictions and little or no programming overhead for scientific or engineering software packages. While generally implemented in Java, Jest links some computationally intensive applications to C-routines to improve performance. Once Jest is integrated in our laboratory's electronic book, our research becomes reproducible by the push of a button in any world-wide-web browser.

INTRODUCTION

Jest (Java for estimations) separates optimization and application software without imposing restrictive limitations on either. Consequently, experts independently implement complementary and reusable optimization and application software, disregardless of the algorithms' internal complexity. The Jest library is accompanied by a growing number of examples and solvers. This article is an advertisement for the Jest software and future reproducible electronic web documents. Preliminary versions are available at `http://sepwww.stanford.edu/sep/matt/jest`.

Off-the-shelf optimization packages usually need their vectors in simple generic data structures, such as one-dimensional Fortran arrays. Unfortunately, generic data structures are often inconvenient or unacceptable in the case of complex vectors or vectors too large to hold in-core. Alternatively, laboratories develop their own in-house optimization software that they adapt to their local data structure and application type. This approach, however, locks a laboratory into a fixed data structure and application style while simultaneously preventing any collaboration with laboratories of a different standard.

Rather than defining a specific data representation, Jest defines the functionality of basic numerical analysis objects, such as vectors, vector spaces, and operators, and solvers. Jest's most abstract package, Jam (Java for mathematics), does not contain any implementations but consists of a collection of Java interfaces that assign names and methods to each of Jest's mathematical objects. This interface level ensures that any implementation of an interface will be compatible with any other existing or future Jest object. Consequently, Jam does not impose any implementation detail, such as a vector representation or a subroutine invocation. Instead, Jam's interfaces require all vectors to implement the simple Hilbert vector operations such as addition of two vectors, scaling of a vector, or the computation of the norm of a vector. Similarly, the interfaces require that every operator implements standard methods to image a vector to another vector or to return a reference to the vector space of its domain or range. Jam is not only general by not imposing a specific implementation, it is also extendible by allowing new mathematical object interfaces to be added. Overall, Jam offers a plug-and-play quality: a programmer can accept or replace any part of Jest and preserve compatibility by implementing the corresponding Jam interfaces.

Jest's Juice package contains implementations of Jam interfaces: general solver classes (e.g. a conjugate gradient solver), simple vector and space classes (e.g. a space of real numbers), simple operator classes (e.g. a zero and an identity operator), and classes of compound vectors and operators that wrap their components into a single vector respectively operator (e.g. product vector or block operator). Since Juice classes are mostly trivial, light-weight implementations, we expect most Jest programmers to use them independent of their particular application.

Besides Jest's optimization framework, Jag's vector and operator implementation constitutes a budding seismic imaging library in its own right. We hope that its object orientation will permit programming complex but reusable processing algorithms and data structures easily.

Since Jest is implemented in Java, we hope that soon we will routinely publish reproducible research documents (Schwab and Claerbout, 1996) on the Web. Reproducible Web documents could change the way researchers publish and collaborate since they make computational research tools and results as accessible as a button click (Figure 1).

Obviously, Jest is not the first library that attempts to separate solver and application software. Examples of well-known high-quality optimization libraries are MINOS (Murthag and Saunders, 1983 revised 1995) and MINPACK (Moré et al., 1980). However, these libraries are defined in procedural languages and impose severe restrictions on the application software. To the contrary, Jest does not require any particular implementation of any of its objects. It only requires that an object's mathematical operations are implemented somehow.

Historically, Jest is the latest in a series of optimization libraries. Nichols and Dye redesigned CLOP's class hierarchy to express abstract objects of vector algebra. Independently Gockenbach and Symes of the Rice Inversion Project developed a similar C++ library. In 1994, the efforts of the two laboratories merged into the Hilbert Class Library (HCL). Gockenbach published HCL (Gockenbach and Symes, 1996) and since maintains a detailed web-page (Gockenbach, 1996). Jest began as a Java implementation of HCL. We found Java much easier to use and learn than C++: an important fact for traditional Fortran or C programmers. Fall 1997, Jacob and Karrenbach (1997) implemented a multi-threaded velocity inversion based on Jest and processed a seismic field example.

A DECONVOLUTION EXAMPLE

To introduce Jest we will discuss what probably is the *Hello World* program of image processing: image restoration by deconvolution. Many physical transmission systems blur their input signal. For example, atmospheric turbulences blur satellite and telescope images. The broadband wavelet in a seismic experiment blurs the reflectivity series of the Earth. Instruments themselves cause a degradation of their input which often is well described by a blur.

Mathematically, a blurred image $y(t)$ is usually described as

$$y_i = \sum_j b_{i-j} x_j ,$$

where x_i is the original unblurred image, and b_i is the blurring filter. The discrete convolution is a linear operator and can be written as a matrix multiplication $\mathbf{y} = \mathbf{B}\,\mathbf{x}$.

In our example, we deconvolve the time series $\mathbf{y}$ recorded by a seismograph with known blur filter $\mathbf{b}$. To estimate the unblurred image $\mathbf{x}$,

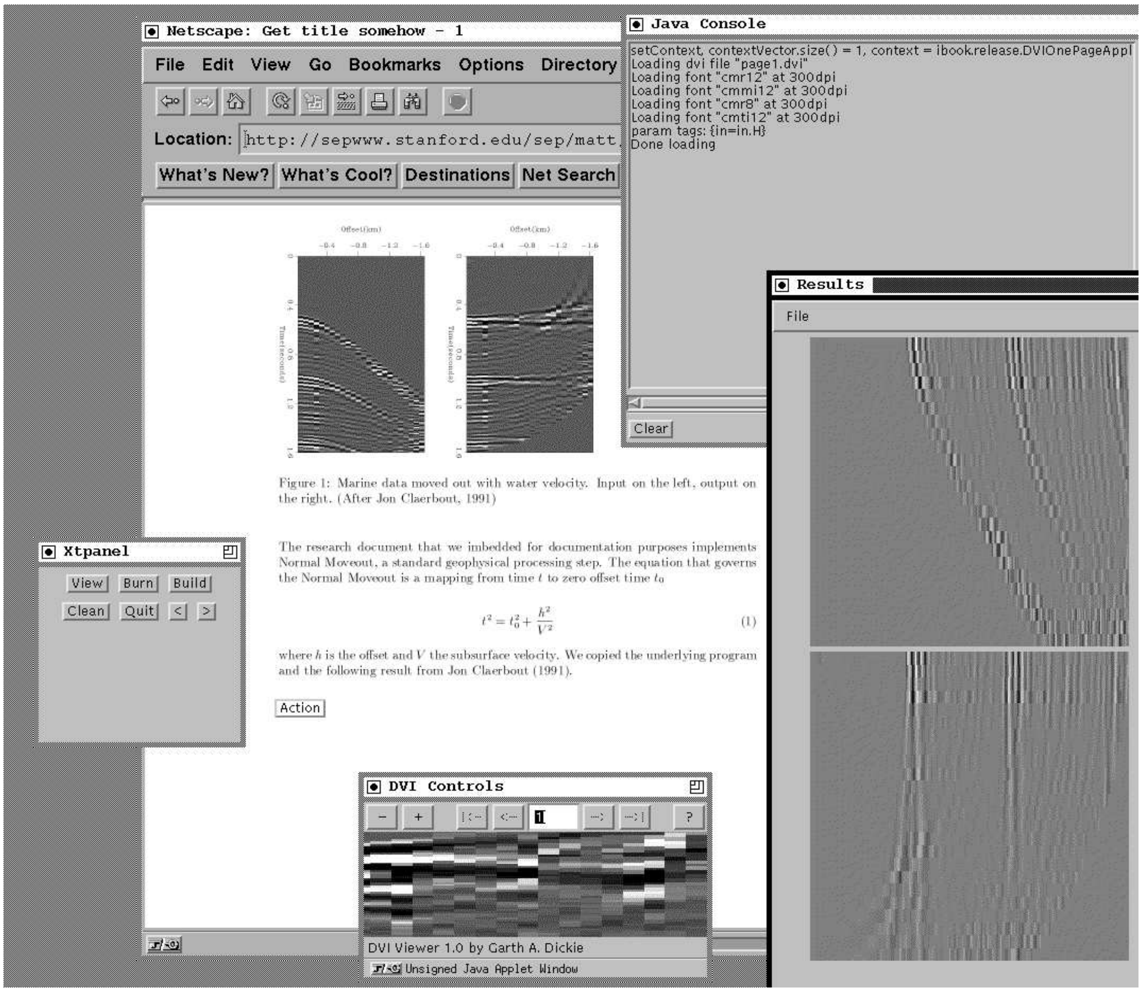

Figure 1: A Java DVI viewer, IDVI (Dickie, 1997), displays a TEXresearch document in a Netscape browser. The window at the bottom center helps a reader to navigate the IDVI viewer. The Action button at the bottom of the page activates a Java Xtpanel applet (left next to the button) that permits a reader to recompute the document's result. The browser downloads the necessary Java byte code from the server and executes them on the reader's machine. The Java console at the top right shows the executed commands. Another button enables the reader to download the document's corresponding application package. The final result figure is displayed in the window at the bottom right. Soon we hope to be able to publish such reproducible World Wide Web documents routinely.

Table 1: Some Jest vector methods (name, syntax, effect)

add	`x.add(y)`	$x \leftarrow x + y$
scale	`x.scale(a)`	$x \leftarrow ax$
dot	`a = x.dot(y)`	$a \leftarrow \langle x, y \rangle$
getSpace	`x.getSpace()`	returns vector space

Table 2: The Jest vector space methods (name, syntax, effect)

hasMember	`s.hasMember(x)`	true if vector x is member
newMember	`x = s.newMember()`	creates a new member x

we seek the unconstrained minimum of

$$\min_{\mathbf{x}} \|\mathbf{y} - \mathbf{B}\,\hat{\mathbf{x}}\|$$

which yields the least square solution

$$\hat{\mathbf{x}} = (\mathbf{B}^T\mathbf{B})^{-1}\mathbf{B}^T\mathbf{y}.$$

The routine that deconvolves the time series combines Jest vectors that represent $\mathbf{y}$, $\mathbf{b}$ and $\hat{\mathbf{x}}$, a Jest linear-operator-with-adjoint to implement the convolution $\mathbf{B}$, and finally Jest's conjugate gradient solver to estimate the solution vector $\hat{\mathbf{x}}$. Figure 2 shows the known filter, the blurred input time series, and the deconvolved result.

Vectors
The observed data $\mathbf{y}$, the known blurring filter $\mathbf{b}$, and the unknown deblurred image $\mathbf{x}$ are vectors. Each of them is accompanied by a vector space. Vector and vector space are Java interfaces that define the basic vector operations, such as listed in Table 1 and Table 2.

In the deconvolution example, the various vectors are represented by Rsf (Regularly Sampled Function) objects. The Rsf class is implemented as a multi-dimensional, in-core array. The RsfSpace class is an array of physical Axis descriptions. Rsf and RsfSpace are, of course, implementations of the general vector and space interface. and consequently implement the basic vector methods of Table 1 and Table 2. The Rsf and RsfSpace classes provide additional features, but for the moment it suffices to know that an Rsf contains a `write(filename)` method that stores a representation of itself and its vector space in a disk file. A class RsfFactory can read such a disk file and create the corresponding Rsf.

Operator
The convolution $\mathbf{B}$ in our example is a linear operator that implements Jest's operator interface. A few operator methods are listed in Table 3.

The interface for a linear operator with adjoint is called *hasAdjoint* and extends the operator interface by a method `adjoint()` that returns the adjoint operator of the original operator. Many solvers require a linear operator's adjoint.

Table 3: Some Jest operator methods (name, syntax, effect)

image	`y = f.image(x)`	$y \leftarrow f(x)$
image	`f.image(x,y)`	$y \leftarrow f(x)$
addImage	`f.addImage(x,y)`	$y \leftarrow y + f(x)$
getDomain	`f.getDomain()`	returns vector space

The deconvolution example uses a simple convolution class `Tcai` and its adjoint; both implement the hasAdjoint interface. An auxiliary class TcaiFactory creates the needed transient convolution object. Note, that the linear operator is not necessarily formulated as an explicit matrix multiplication, but as a general operator object.

Deconvolution
The code fragment `tcaiDecon` on this page is the main routine of the deconvolution problem. The class instantiates two Rsf objects from corresponding data files. One contains the blurred time series, the other one the blurring filter. Next, the program instantiates a transient convolution operator, `convB`. Since the Conjugate Gradient Solver requires a positive-definite operator, we recast example's operator and vectors according to the normal equations

$$\begin{aligned} \mathbf{B}\mathbf{x} &= \mathbf{y} \\ \mathbf{B}^T\mathbf{B}\mathbf{x} &= \mathbf{B}^T\mathbf{y}. \end{aligned}$$

The convolution's `convB.adjoint()` method yields the adjoint operator. Jest's CompoundLinearOperatorAdjoint class chains the forward and adjoint of `conv` to implement $\mathbf{B}^T\mathbf{B}$ in a single operator `convBn`. Jest provides a family of such compound operator classes. To create a starting vector for the solution, we request a new member from the domain of the operator. The values of the vector elements default to zero. Finally, we instantiate a simple CG solver that estimates the unblurred image. The integer argument at the solver's instantiation is the number of iterations and the arguments of the solve method are the elements of the normal equation: the compound operator $\mathbf{B}^T\mathbf{B}$, the known data vector $\mathbf{B}^T\mathbf{y}$, and the initial guess $\mathbf{x} = \mathbf{0}$ of the unknown model vector. The solver updates the vector $\mathbf{x}$ to its estimate of the model. The last line of the deconvolution routine writes the estimated image to the system's standard output stream. Figure 2 shows the inputs and the result of the inversion.

```
package conv.decon;
import jam.vector.*;
import juice.solver.*;
import juice.operator.*;
import rsf.vector.*;
import conv.convt.*;
/**
* I solve y = B x for x.                                       <BR>
* y is a    known time series                          (trace).<BR>
* B is a    known convolution operator of the blur (wavlt).<BR>
* x is an unknown relectivity.                                 <BR>
* This creates the normal equations: y_n = B'y and B_n = B'B and
* calls a CGSolver to solve y_n = B_n x, respectively B'y = B'B x.
*/
public class TcaiDecon {
  public static void main(String[] args) {
    Rsf  yy = RsfFactory.newRsf("./trace.H"); // read   y
    Rsf  bb = RsfFactory.newRsf("./wavlt.H"); // read   b
    Tcai convB = new TcaiFactory(bb).         // create B
      getTcaiFromOutputSpace((RsfSpace) yy.getSpace());
    Vector     yn = convB.adjoint().image(yy);// create y_n = B'y
    CompoundLinOpAdj convBn =                 // create B_n = B'B
      new CompoundLinOpAdj(convB, convB.adjoint());
    Rsf xx = (Rsf) convBn.getDomain().newMember();     // creates x
    new CGSimple(500).solve(convBn,yn,xx);    // solve  y_n = B_n x
    xx.write( System.out );                   // write  x
  }
}
```

Solver
The conjugate gradient solver that estimates the unblurred image vector $\hat{\mathbf{x}}$ implements the Jest solver interface. The solver interface defines a single method, `solve(operator B, vector y, vector x)`. The arguments stem from the equation $\mathbf{B}\mathbf{x} = \mathbf{y}$. The `solve()` invocation requires a starting solution for x which it then will update to its best estimate. The solver's termination criteria are usually set by the solver's constructor.

The solver program literally translates the Conjugate Gradient algorithm into Java code. Gill et al. (1981) list pseudo-code[1] for the con-

[1] Gill's algorithm and the implementation trivially differ in the representation of the operator by the letter $\mathbf{B}$ and the residual sign convention.

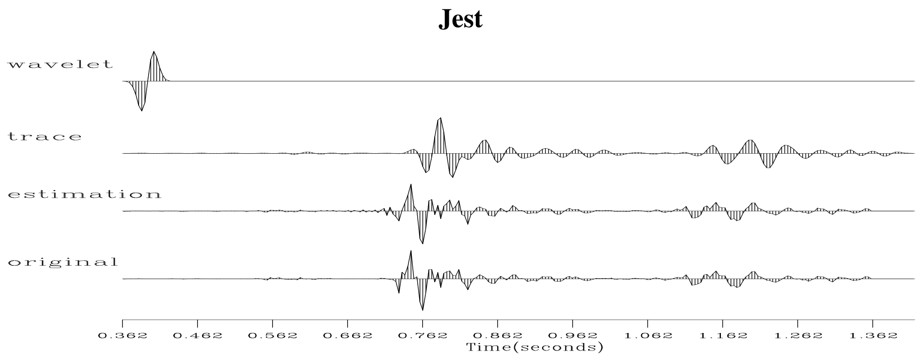

Figure 2: The top signal is the known filter. The second signal is the received, blurred signal. The third signal is the deblurred signal estimated from the signals above. The bottom signal is the correct answer.

jugate gradient algorithm as:

$$
\begin{aligned}
p_k &= -r_k + \beta_{k-1} p_{k-1} && (1)\\
\alpha_k &= \frac{|r_k|_2^2}{p_k^T B p_k} && (2)\\
x_{k+1} &= x_k + \alpha_k p_k && (3)\\
r_{k+1} &= r_k + \alpha_k B p_k && (4)\\
\beta_k &= \frac{|r_{k+1}|_2^2}{|r_k|_2^2}. && (5)
\end{aligned}
$$

Each pseudo-code step corresponds to one or two lines of Jest instructions, as the program's comments indicate. More importantly, the program's instructions are as general as the algorithm, since the program's objects – vectors and operators – are of the same abstraction as the algorithm's mathematical symbols.

```
package juice.solver;
import  jam.solver.*;
import  jam.vector.*;
import  jam.operator.*;
/** Conjugate Gradient Solver. */
public class CGSimple implements LinearSolver {
  private int maxIter;
  /** @param maxIter maximum number of iterations. */
  public CGSimple(int maxIter) { this.maxIter = maxIter; }
  /**
   * @param A operator to be inverted
   * @param b known vector
   * @param x starting estimate of unknown vector
   */
  public void solve(Operator A, Vector b, Vector x ) {
    if (! (A instanceof LinearOperator))
      System.err.println("Error: A not LinearOperator");
    Vector r = A.getRange().newMember(); // init r = b - Ax
    A.residual(x,b,r);
    r.neg();
    Vector p = (Vector) r.copy();        // init p = r
    Vector v = A.getRange ().newMember();// init v = 0
    float rtrNew = r.norm2();

    for (int iter=0; iter< maxIter; iter++) {
      A.image(p, v);                // v = Ap
      float ptv   = p.dot( v);      // ptv = p^t Ap
      float alpha = rtrNew/ptv;     //   a = |r|^2 / p^t Ap
      if (ptv == 0f) System.err.println("Error: ptv=0.");
      x.addScale(  alpha, p);       //   x = x + a p
      r.addScale( -alpha, v);       //   r = r - a Ap
      float rtrOld = rtrNew;
      rtrNew       = r.norm2();
      float beta = rtrNew/rtrOld;   //   b = |r|^2 / |r|^2
      if (rtrOld == 0f) System.err.println("Error: rtrOld=0.");
      p.addScale( beta, r, p);      //   p = -r + b p
  }}}
```

CGSimple is a primitive member of Jest's solver family. Its entire interaction with a program is restricted to setting the number of iterations and invoking its solve routine. Standard iterative Jest solvers accept an iterator object that contains application specific initializations, termination criteria, report instructions, and final clean up code.

A MISSING DATA EXAMPLE

Jest is designed for complex, large-scale problems. To hint at the problems we had in mind when designing Jest, Figure 3 shows the result of a medium-sized missing data estimation (Claerbout, 1994). The estimation almost exclusively involves tools used in the deconvolution example above.

The panel on the left shows bathymetry (water depth) measurements $\mathbf{x}_k$ above the mid-ocean ridge in the Pacific. The measurements were taken by a side-scan sonar hanging off the bottom of a ship. The vessel's path did not cover the entire image area and consequently we attempt to estimate the missing data. The masking operator $\mathbf{M}_k$ extracts the known data, $\mathbf{M}_k\mathbf{x} = \mathbf{x}_k$ from the total data set $\mathbf{x}$.

The solution in the center panel minimizes the convolution $\min_{\mathbf{x}}(\mathbf{l} * \mathbf{x})$ of the Laplacian filter $\mathbf{l}$ and the data $\mathbf{x}$ under the constraint that the known data does not change, $\mathbf{M}_k\mathbf{x} = \mathbf{x}_k$.

The solution in the right panel uses a two stage Prediction Error filter technique. First, we minimize the convolution $\min_{\mathbf{p}}(\mathbf{p} * \mathbf{x}_k)$ to estimate the Prediction Error filter $\mathbf{p}$. Next, we replace the Laplacian in the earlier formulation by the Prediction Error filter and solve again for the missing data. The missing data estimation is easily built from a combination of the objects shown in the deconvolution example, a specialized Prediction Error filter class, and the mask operator $\mathbf{M}_k$.

A BRIEF EVALUATION OF JEST

Jest was designed to enable researchers to prototype complex inversion problems rapidly, to scale the prototyped solutions to large data examples, to maintain and reuse the software developed for different projects, to publish the results effectively, and to encourage collaboration among researchers of the same laboratory, of distant laboratories, and across disciplines.

Jest's advantages

Jest's high-level, mathematical encapsulation successfully separates the expertise of numerical analysis, scientific application, and data structure into its solver, operator, and vector classes. Essentially, it is that natural separation that sets Jest apart from more common optimization packages. The back-of-an-envelope designs of a researcher will usually translate directly to Jest's mathematical objects. Since the innovation will probably lie within a single area of expertise, such as a solver, or a new operator, a researcher will usually need to program a single class. The new class can then be combined with Jest library items to build complex tests and examples.

In general, Jest's code is more readable than comparable C and Fortran implementations. On the numerical battlefield, when numbers are added and square roots are taken, Java is superior to C due to its lack

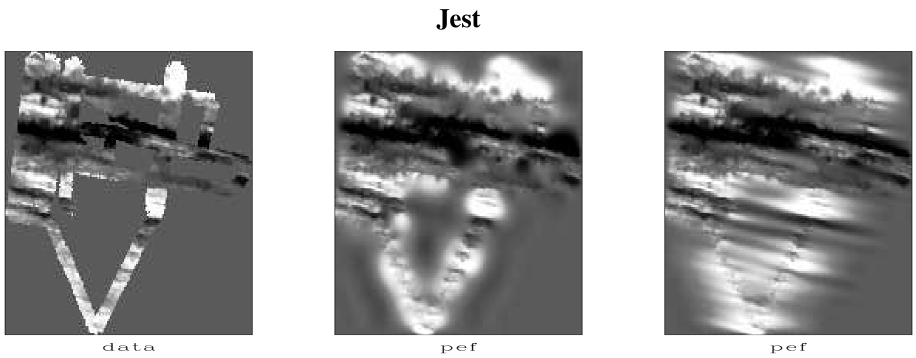

Figure 3: The image in the left panel is the bathymetry data collected by a side-scan sonar along a vessel's path across the Pacific mid-ocean ridge. The image in the center panel shows an estimation of the missing data based on the convolution with a Laplacian. The image in the right panel shows an estimation by a two-stage linear Prediction Error filter technique.

of pointers and addresses. Here Java's readability is generally comparable to Fortran. On the abstract algorithmic level of solvers, Jest's mathematical objects are unsurpassed in conciseness and generality. Overall, as in every object-oriented language, a lot of *dirty* details, e.g. an array's parameters or input and output details, are hidden in Jest's objects.

... and disadvantages

Java is slower than C. When interpreted, Java is about 10 times slower. When compiled by a just-in-time compiler, Java is about 2 or 5 times slower. Experts predict that standard Java will soon be nearly as fast as C or C++, but probably never as fast as Fortran.

Nevertheless, we intend to use Jest for large scale problems. To improve Jest's performance, we implement computationally intensive routines in C and link them with Jest objects. Java easily facilitates the linking with its native method mechanism and the code remains platform-independent if the method invocation is programmed to default to the Java version, if a native version is unavailable.

Besides better performance, we wish Java contained a primitive type for complex numbers and the ability to overload the array index operator `[]`. For the same reasons that Java offers float and double primitive types, it should include a complex-number primitive type[2]: such a complex-number primitive is important in many applications, a class implementation incurs unacceptable inefficiencies, and only a primitive type offers the concise and intuitive arithmetic operators, such as '+'. Alternatively, a complex number can be simply represented by two floats. However, the lack of a single complex-number type and its associated methods is error-prone, tedious, and not object-oriented.

We wish the `[]` indexing operator could be overloaded in Java. Again, a multi-dimensional array is a fundamental structure in science and engineering applications. Internally the array might be stored as a Java array, a Java vector, or a huge out-of-core data file. Independent of the internal structure, applications need to index the array elements with a general indexing operation. Only the `[]` indexing operator offers the conciseness that matches the operation's simplicity. Since Java generally does not overload operators[3], only classes with an internal Java array representation can offer the desired `[]` indexing operator. In our applications, we could not forego the conciseness of the array index expression and consequently had to pierce the encapsulation of our array-like vector classes to ensure access by index for operators. Consequently, the array-like Rsf vector and its operators are more closely linked than we wish.

We believe Java's future in science and engineering will be greatly enhanced if those two shortcomings - a primitive complex-number type and overloading the index operator - were redressed. Java designers should not underestimate the importance of science and engineering as sources of innovation and education. A preliminary version of Jest is available at
`http://sepwww.stanford.edu/sep/matt/jest`.

[2] Fortran has a complex-number type, C and C++ do not.

[3] Java makes an exception for the '+' operator in string concatenations.

REFERENCES

Claerbout, J. F., 1994, Applications of Three-Dimensional Filtering: Stanford Exploration Project.

Dickie, G. A., 1997, About idvi: `http://www.geom.umn.edu/java/idvi`.

Gill, P. E., Murray, W., and Wright, M. H., 1981, Practical optimization: Academic Press.

Gockenbach, M. S., and Symes, W. W., 1996, The Hilbert Class Library: a library of abstract C++ classes for optimization and inversion: Computers and Mathematics with Applications, **32**.

Gockenbach, M., 1996, The Hilbert Class Library: A library of abstract classes for C++ optimization and inversion: `http://www.trip.caam.rice.edu/`.

Jacob, M., Philippsen, M., and Karrenbach, M., 1997, Large-scale parallel geophysical algorithms in java: A feasibility study: `http://wwwipd.ira.uka.de/~jacob`.

Moré, J., Garbow, B., and Hillstrom, K., 1980, User guide for minpack: Technical Report ANL-80-74, Argonne National Laboratory, Argonne, IL.

Murthag, B. A., and Saunders, M. A., 1983 (revised 1995), Minos 5.4 user's guide: Report SOL 83-20R, Systems Optimization Laboratory, Stanford University, CA.

Schwab, M., and Claerbout, J. F., 1996, SEP's reproducible electronic documents: `http://sepwww.stanford.edu/redoc`.