Algebraic Java classes for optimization

Matthias Schwab and Joel Schroeder[1]

keywords: *objected-oriented, Java, optimization, geophysics, seismic image processing*

ABSTRACT

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.

Jest (Java for estimations) separates optimization and application software without imposing limitations on either. Consequently, experts independently implement complementary and reusable optimization and application software, irregardless of the algorithms' internal complexity. The Jest library is accompanied by a growing number of examples and solvers. This chapter is an advertisement for the Jest software: my current version of Jest is available at my website[2].

Off-the-shelf optimization packages are usually limited to particular implementations and to somewhat arbitrary calling sequences. Off-the-shelf optimization packages usually implement 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 complicated 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

[1]**email:** matt@sep.stanford.edu, schroeder@alumni.stanford.org

[2]`http://sepwww.stanford.edu/oldsep/matt/jest.`

and application style while simultaneously preventing any collaboration with laboratories of a different standard. Additionally, any new routine has to comply with the calling sequences of existing routines. A new routine can easily take advantage of old programs by invoking their known calling sequence. However, a new routine is not easily substituted for an existing alternative unless the two programs share identical calling sequences: a coincidence that is not easily maintained among a diverse group of contributing programmers.

Jest is centered around a set of abstract mathematical interface classes – *Jam* (Java for mathematics) – that resolve the implementation- and invocation-compatibility problem of traditional optimization libraries. Rather than defining a specific data representation, Jam defines the functionality of basic numerical analysis objects, such as vectors, vector spaces, operators, and solvers. Jam 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. The central Jam package is the only part of Jest that is shared by all users (see Table 1). All other Jest packages implement Jam interfaces. The individual The Javadoc pages available at our web page are

Jam interfaces do not prescribe any particular implementation, such as a vector representation. Collaboration among classes relies on the method invocations defined by the Jam interface. For example, Jam's vector interface requires 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. A Jam-compatible class implements a Jam interface and limits itself to use only Jam-defined interface methods. a programmer of a Jest class is not concerned with the implementation or invocation of other Jest classes, but only with the invocation of the relevant Jam interfaces. Furthermore, the Java compiler ensures that any class that implements a Jam interface does indeed supply all necessary methods. Consequently, a programmer can, in principle, interchange all classes that implement the same Jam interface.

But how can one define a complete and general set of Jam interfaces? Mathematics! Jam defines names and method invocations for basic mathematical entities (vectors, operators, and solvers) and their standard operations. Ultimately, the Jam interface is as complete and general as the original mathematical definition. Furthermore, the mathematical framework of Jest is shared by researchers in numerical analysis, engineering, and natural sciences. A perfect interface design! However, only a full-blooded object-oriented language allows the necessary abstract formulation.

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. Finally, Jam is not only general by not imposing a specific implementation, it is also extendible by allowing new mathematical object interfaces to be added.

Jest packages (class libraries) are separated by user community. The top package, Jam, is shared by all users and only includes interfaces for the abstract mathematical entities, such as a vector and its operations (add, scale, etc.). User groups (e.g., my laboratory's geophysicists) author packages the classes of which implement Jam interfaces. For example, Jag's Rsf class is my laboratories data structure and implements the Jam vector interface.

Package name	User Group	Class type	Vectors	Operators	Solvers
Jam	**Anyone**	Interfaces of abstract math objs	Vector	Operator CompoundOperator	Solver IterativeSolver
Juice	Mathematicians	Classes of numerical analysis	Rn	MatrixMultiplication	CG solver, Newton-Raphson ..
Jag	SEP geophysicists	Classes of seismic processing	Rsf, Isf	Nmo, convolution ...	
Wave	2nd research geophysicists	Classes of seismic processing		Kirchhoff migration ...	Arnoldi solver ...
Benzol	Chemists	...	...	...	...

Table 1: Package hierarchy. Jest packages (class libraries) are separated by user community. The top package, Jam, is shared by all users and only includes interfaces for abstract mathematical entities, such as a vector and its operations (add, scale, etc.).

The Juice package contains my elementary numerical analysis classes (e.g., my conjugate gradient solver). The bottom two packages do not exist but represent potential extensions by other user groups. Packages are bound to be compatible since they implement Jam interfaces. Consequently, any given user package only implements the classes that are unique to its user community. Jest's extendible structure and mathematical objects aims to garner collaboration across disciplines.

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.

Finally, the *Jag* (Java for geophysics) group of packages contains vectors and operators for seismic image processing. Jag is centered around the Rsf (Regularly Sampled Function) vector class. An Rsf vector represents a multi-dimensional regularly sampled, physical functions and is similar to a traditional float array type. An array of axis objects describes the Rsf vector space. Additionally, Jag comprises several seismic image processing packages based on Rsf vectors, e.g., convolution, edge detection, nonstationary patching, and missing data estimation. Within the Jest optimization framework, Jag's vector and operator implementations constitute a budding seismic processing library.

Java is a fun programming language. We struggled for years to develop an objected-oriented optimization and imaging package in C++, but our Fortran-programming colleagues found C++ too complex and difficult to master. We consider Java easy to program and hope, as Jest grows and prospers, some of our colleagues will join our effort.

We have little experience with off-the-shelf optimization packages since our laboratory, the Stanford Exploration Project, maintains its own small collection of optimization routines. It is this isolation of researchers and laboratories that Jest sets out to overcome. However, I will first list some brief and brash historic and technical comments on a few alternative libraries for contrast and comparison.

Jest is not the first library that attempts to separate solver and application software. Examples of well-known high-quality optimization libraries are MINOS (**?**) and MINPACK (**?**). 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. Our laboratory developed CLOP (1993), an experimental geophysical inversion library implemented in C++. CLOP enabled Lumley et al. (Lumley et al., 1994) to prototype a new multiple suppression scheme that probably would not have been attempted in our laboratory's traditional Fortran environment. 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) (Schwab, 1994). 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.

Lydia Deng et al. (Deng et al., 1996; **?**) developed a C++ optimization library for geophysical inversion that addresses similar problems as Jest. Fomel (Fomel and Claerbout, 1996; Fomel et al., 1996) focused on performance and implemented a geophysical optimization library in Fortran90 using its modular features.

In the summer of 1996 (before the development of Jest), SEP held an informal conference on objected-oriented numerics (Claerbout and Biondi, 1996). Afterwards, several participants cooperated to develop a framework to combine HCL's C++ classes with high-performance Fortran90 routines. The required linking of Fortran90 and C++ proved difficult but doable(Schroeder and Schwab, 1996; Urdaneta and Karrenbach, 1996).

Fall 1997, Jacob and Karrenbach (1997) implemented a multi-threaded velocity estimation based on Jest.

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 modeled as the convolution of the original image $f(t)$ by a blurring filter $b(t)$:

$$y(t) = b(t) * x(t) = \int b(t - t')x(t')dt'.$$

If the image is digitized, convolution can be expressed discretely as

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

The discrete version can be formulated as a matrix multiplication $\mathbf{y} = \mathbf{B}\,\mathbf{x}$:

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} b_1 & 0 & 0 & 0 & 0 \\ b_2 & b_1 & 0 & 0 & 0 \\ b_3 & b_2 & b_1 & 0 & 0 \\ 0 & b_3 & b_2 & b_1 & 0 \\ 0 & 0 & b_3 & b_2 & b_1 \\ 0 & 0 & 0 & b_3 & b_2 \\ 0 & 0 & 0 & 0 & b_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}$$

which in this case implements transient boundary conditions.

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 $\mathbf{x}$, 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 1 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 their own vector space. Vector and vector space are Java interfaces that define the basic vector operations, such as listed in Table 2 and Table 3.

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 2 and Table 3. 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.

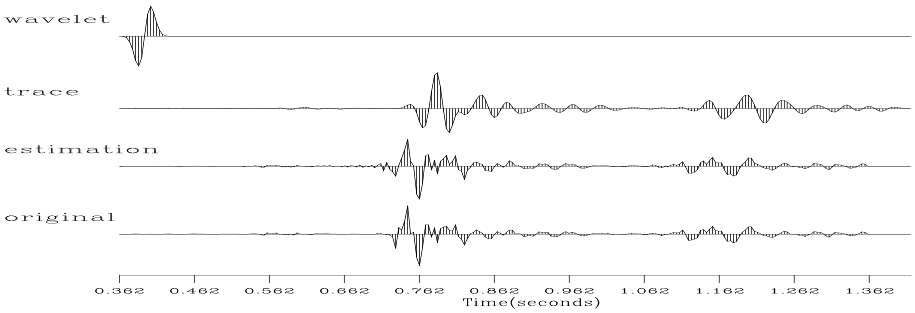

Figure 1: Seismogram deconvolution. The top signal is the known filter. The second signal is the received, blurred signal. The third signal is the deblurred signal estimated from the signals above. The bottom signal is the correct answer. $\boxed{\text{tcai}}$ [ER]

Table 2: 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 3: 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

Table 4: 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

Operator

Theoretical considerations or problem descriptions often represent linear transformations by matrices. In scientific and engineering applications, as in the convolution example above, these matrices are often large and sparse. An implementation of such a transformation as an explicit matrix multiplication is inefficient. Instead Jest utilizes the more general concept of operator interfaces and leaves implementation details to each application. The operator concept, of course, includes the specific implementation of a linear operator as a matrix but also generalizes to nonlinear transformations. 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 4. 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. 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.

Deconvolution

The code fragment `tcaiDecon` on the following 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 the example's operator and vectors according to the normal equations

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

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 1 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 yn = B'y
    CompoundLinOpAdj convBn =                      // create Bn = B'B
      new CompoundLinOpAdj(convB, convB.adjoint());
    Rsf xx = (Rsf) convBn.getDomain().newMember(); // create x
    new CGSimple(500).solve(convBn,yn,xx);         // solve  yn = Bn 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 $\mathbf{Bx} = \mathbf{y}$. The `solve()` invocation requires a starting solution for $\mathbf{x}$ 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 for the conjugate-gradient algorithm as:

$$p_k = -r_k + \beta_{k-1} p_{k-1} \tag{1}$$

$$\alpha_k = \frac{|r_k|_2^2}{p_k^T B p_k} \tag{2}$$

$$x_{k+1} = x_k + \alpha_k p_k \tag{3}$$

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

Each pseudo-code step corresponds to one or two lines of Jest instructions, as the program's comments indicate[3]. 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 is not a 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.

[3]Gill's algorithm and the implementation trivially differ in the representation of the operator by the letter **B** and the residual sign convention.

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 2 shows the result of a medium-sized missing data estimation (Claerbout, 1994). The estimation almost exclusively involves tools used in the deconvolution example above.

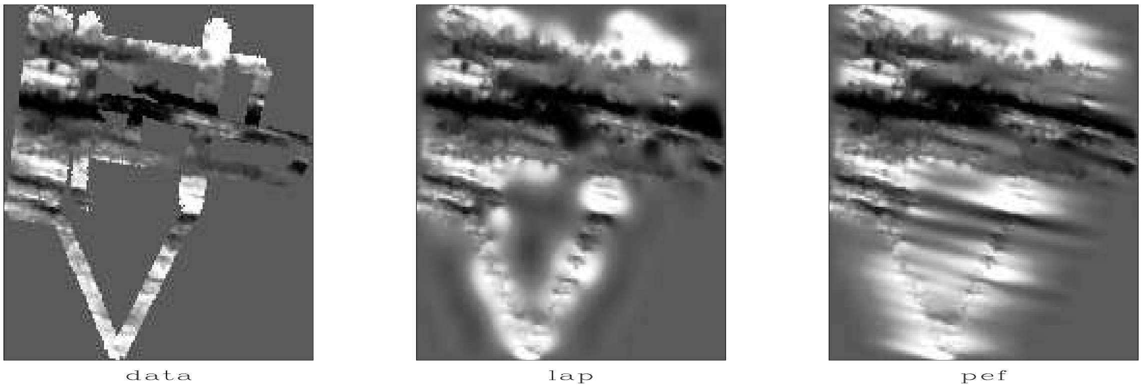

Figure 2: Estimation of missing bathymetry data. The image in the left panel is the bathymetry data collected by a side-scan sonar along a vessel's path across the Pacific mid-ocean ridge. The image in the center panel shows an estimation of the missing data based on the convolution with a Laplacian. The image in the right panel shows an estimation by a two-stage linear prediction-error filter technique. seabeam [ER]

The panel on the left shows bathymetry (water depth) measurements $\mathbf{d}_k$ 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, $\mathbf{M}_k \mathbf{d} = \mathbf{d}_k$ from the total data set $\mathbf{d}$.

The solution in the center panel minimizes the output of the Laplace operator applied to the data $\mathbf{d}$, $\min_{\mathbf{d}}(\nabla^2 \mathbf{d})$, under the constraint that the known data does not change, $\mathbf{M}_k \mathbf{d} = \mathbf{d}_k$.

The solution in the right panel uses a two stage prediction-error technique. First, we minimize the convolution $\min_{\mathbf{p}}(\mathbf{p} * \mathbf{d}_k)$ 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 $\mathbf{M}_k$.

DISCUSSION

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 numerical analysis into its solver classes, scientific application into its application classes, and data structure into its 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 only need to program a single class. The new class can then be combined with Jest library items to build complex tests and examples. A small set of well-defined mathematical class interfaces ensures that such project-specific extensions are fully compatible with the entire Jest framework.

To help maintain Jest, its classes are accompanied by simple test routines. Furthermore, Jest includes a series of research-type examples that, due to our laboratory's reproducible research mechanism, are recomputed at the push of a button. To maintain Jest, we routinely and automatically execute all of Jest's test and example routines. Furthermore, the test and research cases complement Jest's Javadoc pages by offering a programmer illustrative 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 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.

For the future, I envision reproducible electronic documents that distribute scientific research on the World Wide Web. Reproducible research documents (Schwab and Claerbout, 1996; Schwab et al., 1996; Schwab and Schroeder, 1995) organize an researcher's writings and his software implementation of his ideas into easily accessed and maintained packages. The online version of an electronic document, as shown in Figure 3, offers readers buttons that initiate the recomputation of the document's results.

Due to Java's security features, an electronic document implemented as a Java applet can be downloaded and safely executed by any standard Web browser. Thereby, Java enables us to combine a scientific article with its underlying software while avoiding the complications

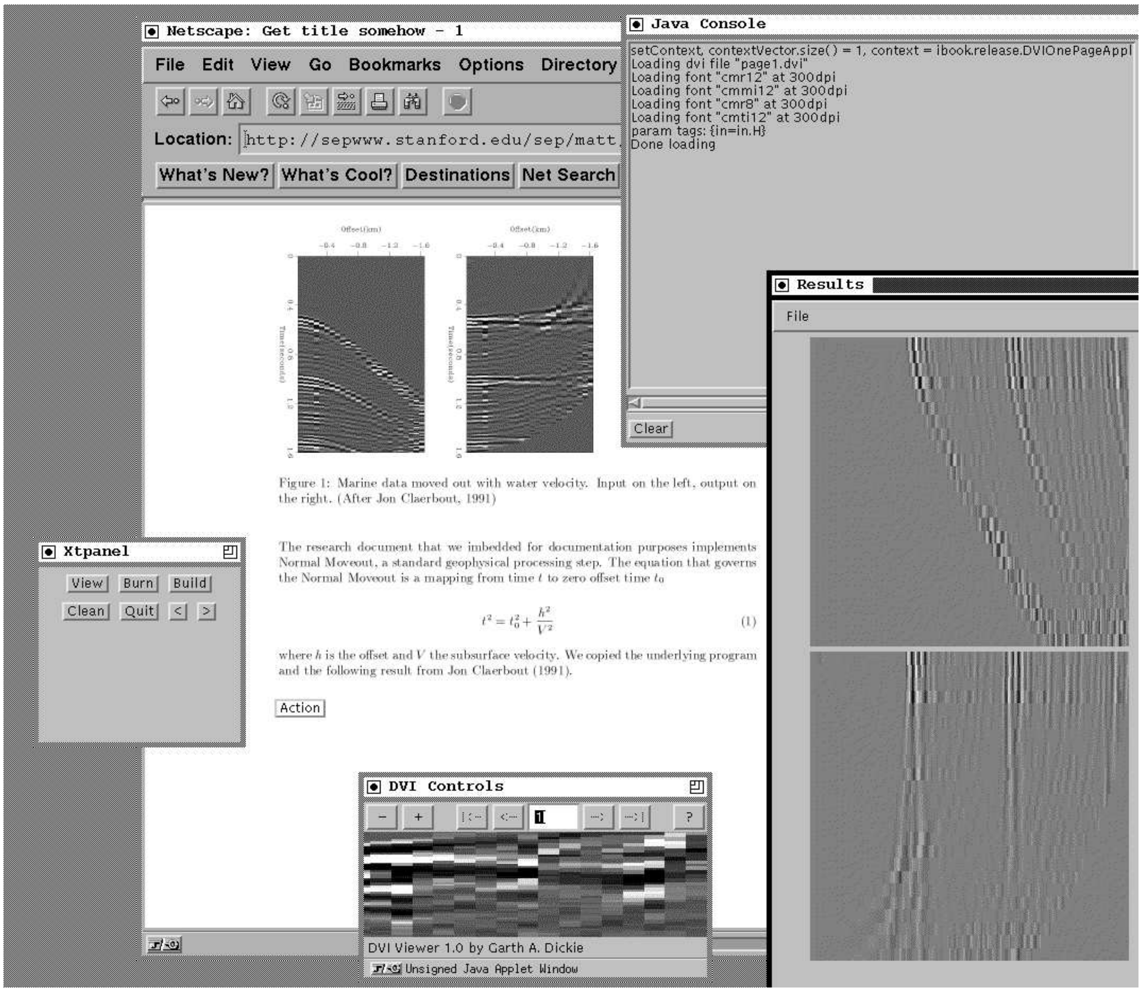

Figure 3: Reproducible electronic document on the Internet. A Java DVI viewer, IDVI (Dickie, 1997), displays a TEX research document in a Netscape browser. The window at the bottom center helps a reader to navigate the IDVI viewer. The Action button at the bottom of the white page activates a Java menu 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. idvi [NR]

of traditional software installation and compilation. The resulting convenience and completeness of reproducible Web documents should foster collaboration among researchers and increase research efficiency. Jest currently contains Java applications. However, first experiments (Figure 3) show how Jest's application can be extended to Java applets and how they can be integrated into a reproducible Web document.

... 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 times slower. Experts predict that standard Java will soon be nearly as fast as C or C++, but probably never as fast as Fortran. Java's run time resolution of all methods that are not declared final and its array bound checks for indexing operations are probably the two features that hurt Jest's performance most. However, we have not yet investigated Jest's performance systematically.

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's native method mechanism easily facilitates the linking of C and Java routines. A Java program that invokes a C routine remains platform-independent if the method invocation is programmed to default to the Java version, when a native version is unavailable.

Typically, a Jest programmer prototypes and tests his program in an all-Java version. If the performance is insufficient, the programmer can profile the application and decide which part of the code to replace by a native method. We expect that straightforward numerical code, such as a convolution kernel, might need to be translated from Java to C, while algorithmically complex code, such as a solver, will not.

Alternatively, Jacob and Karrenbach (1997) report a performance comparable to Fortran when they combine Jest, a specialized compiler (JavaParty), and parallel computers in the optimization of a a large-scale geophysical inversion problem. Unfortunately, to yield that performance, their compiler required a non-standard class modifier regarding the distribution of parallelized objects.

Besides better performance, I 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[4]: (1) a complex-number primitive is important in many applications, (2) a class implementation incurs unacceptable inefficiencies, and (3) 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.

For example, the methods below compute one root of the quadratic equation $ax^2 + bx + c = 0$. The concise first method does not exist in Java, since it uses a complex primitive type. The second method computes the same root but uses a user-defined complex class: It

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

unfortunately obfuscates the simple, original mathematical operation.

Desired method:

```
static Complex root(Complex a, Complex b, Complex c) {
  return  (b - Complex.sqrt(b*b - 4*a*c)) / (2*a) ;
```

Available method:

```
static Complex root(Complex a, Complex b, Complex c) {
  Complex b2      = Complex.multiply(b,b);
  Complex ac4     = Complex.multiply(4,Complex.multiply(a,c));
  Complex discrim = Complex.sqrt(Complex.subtract(b2,ac4));
  return Complex.divide(Complex.subtract(discrim,b)),
                        Complex.multiply(2,a))); }
```

Without a primitive type for complex numbers, we must choose between an encapsulating complex-number class or an error-prone representation by a pair of floats. We partially side-stepped this difficulty by implementing a class for an array of complex numbers. Such an array justifies the class overhead, which was prohibitive for a single complex number, and simultaneously hides the representation of a complex number as a pair of floats. However, using individual complex numbers remains inefficient or tedious unless Java adds a complex-number primitive type.

I 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 permit overloading operators[5], only classes with an internal Java array representation can offer the desired `[]` indexing operator.

For example, the program fragments approximate the second derivative of a one-dimensional array. The first formulation with its natural index expressions is only available if the array is implemented as a native Java array. If the array is a user-defined array (e.g., an out-of-core, a complex number array) or a Java vector, Java's inability to overload the index operator leads to the verbose formulation of the second program fragment.

Desired method:

```
float x[] = new float[10];   // only native Java arrays
float y[] = new float[10];
...
y[i] =  (-x[i+1] + 2*x[i] - x[i-1]) / 4.0;
```

Available method:

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

```
Myarray x = new Myarray(10); // using Java's Vector
Myarray y = new Myarray(10); // or complex array class
...
y.setElement(i,(- x.ElementAt(i+1)
                + x.ElementAt(i  )
                - x.ElementAt(i-1)) / 4.0);
```

In our applications, we could not forego the conciseness and simplicity 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. Therefore, the array-like Rsf vector and its operators are more closely linked than I wish. Additionally, our classes offer function calls to access individual array elements. However, I find such calls unintuitive, especially when the arguments include index and element values (`f.setDatum(3,3.f)`). I was not able to devise a general solution based on enumerations. In summary, I do not know how to write application classes with concise, but general, indexing operators for array-type data implemented by structures other than the original Java array.

I 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.

Future

I continue to improve and enhance Jest by implementing my seismic imaging research in Jest. Jest needs more sophisticated numerical solvers. We may try to translate HCL's limited memory BFGS, implicitly-restarted Arnoldi, or augmented Lagrangian solver using a C++ to Java conversion script. Simultaneously, we seek collaboration with other scientists, engineers, and especially numerical analysists to share and develop the Jest framework. The current version of Jest is available at `http://sepwww.stanford.edu/oldsep/matt/jest`.

ACKNOWLEDGMENTS

Jon Claerbout suggested the separation of solvers and applications using object orientation. Dave Nichols guided me among half-a-dozen other programmers in the implementation of CLOP, a first implementation of Jon's idea. Independently, Dave Nichols and Lester Dye as well as Mark Gockenbach and Bill Symes developed a more rigorous framework that based its class design on mathematical entities. I later helped Lester and Mark to unite the two frameworks into the Hilbert Class Library. Frustrated by HCL's inability to attract programmers among our laboratory's researchers, Joel Schroeder and I designed a Java library that rivals HCL but takes advantage of Java's simpler object-oriented features and its comprehensive library support. Thanks to all of you. It was great fun!

REFERENCES

Claerbout, J., and Biondi, B., 1996, Geophysics in Object-Oriented Numerics (GOON): An informal conference: SEP–**93**, 241–252.

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

Deng, L. ., Gouveia, W., and Scales, J., 1996, The CWP object-oriented optimization library: `http://timna.Mines.EDU/cwpcodes/coool`.

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

Fomel, S., and Claerbout, J., 1996, Simple linear operators in Fortran 90: SEP–**93**, 317–328.

Fomel, S., Crawley, S., and Clapp, R., 1996, A generic NMO program: SEP–**93**, 305–316.

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

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

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`.

Lumley, D., Nichols, D., and Rekdal, T., 1994, Amplitude-preserved multiple suppression: SEP–**82**, 25–46.

Nichols, D., Urdaneta, H., Oh, H. I., Claerbout, J., Laane, L., Karrenbach, M., and Schwab, M., 1993, Programming geophysics in C++: SEP–**79**, 313–471.

Schroeder, J., and Schwab, M., 1996, HCL and regular data: SEP–**93**, 263–272.

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

Schwab, M., and Schroeder, J., 1995, Reproducible research documents using GNUmake: SEP–**89**, 217–226.

Schwab, M., Karrenbach, M., and Claerbout, J., 1996, Making scientific computations reproducible: SEP–**92**, 327–342.

Schwab, M., 1994, Birth of a C++ project: SEP–**82**, 251–256.

Urdaneta, H., and Karrenbach, M., 1996, An IGF90 tutorial: SEP–**93**, 273–292.