Stanford Exploration Project, Report 94, November 11, 1997, pages 361–??

A seismic inversion library in Java

Matthias Schwab and Joel Schroeder[1]

keywords: *objected-oriented, Java, HCL, optimization*

ABSTRACT

Jag is a Java library for numerical optimization of geophysical problems. It shares the fundamental class hierarchy with HCL, a C++ library. We found writing Java easier than writing C++. Java freed us from garbage collection and pointer arithmetic and gave us multiple inheritance of interfaces. During the development we guided our design decisions on a small set of research scenarios. We are confident Jag will excel in prototyping solutions to geophysical inversion problems. Furthermore, we are at the verge of delivering Jag results wrapped in reproducible documents on the World Wide Web. Unfortunately, Java's current performance is inferior to even C++, which might restrict Jag to small- and medium-sized research projects.

INTRODUCTION

Jag (Java Applied to Geophysics) is our new library for numerical optimization of geophysical problems. It is a Java implementation of the earlier C++ libraries, CLOP and HCL.

Jag consists of an abstract interface level, *Jam* (Java Applied to Mathematics), and a set of concrete, implemented classes, Juice[2]. Jam and Juice contain abstract algebraic and numerical objects, such as vectors, spaces, operators, and solvers. Both packages could be shared between researchers of almost any computational scientific field. On the other hand, the packages *Rsf* (Regularly Sampled Function) and *Isf* (Irregularly Sampled Function) comply with Jam's interfaces, but implement vectors, spaces, and operators that are particular to computational research in geophysics. Additionally, Jag includes a *Book* package that tests and illustrates Jag's capabilities on some small- to medium-sized research scenarios.

Jag takes full advantage of Java's object-oriented features, such as abstraction, inheritance, and encapsulation, to ensure the collaboration of its current and future classes. We found it much easier to implement our classes in Java than in C++.

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

[2]Juice is not an acronym, but it is the juice behind Jam.

Specifically, Java's lack of pointers, its automatic garbage collection, and its interface approach to multiple inheritance solved many practical problems we encountered in earlier C++ versions. Most importantly, programming Java was fun.

Unfortunately, Java programs are slow in comparison to C or Fortran implementations. However, new compiler technologies, such as just-in-time compilers, are reducing the performance gap drastically.

Since Jag is implemented in Java, we hope that soon we will routinely publish reproducible research documents (Schwab et al., 1996; Schwab and Schroeder, 1995) on the Web. Reproducible Web documents could change the way researchers publish and collaborate since it makes computational research tools and results as accessible as a button click (Figure 1).

PREVIOUS WORK

Dave Nichols et al (1993) implemented an object-oriented, geophysical inversion library, known as CLOP. CLOP never overcame its experimental character, however. It dealt almost exclusively with the inversion of linear operators, and it stored its data in a proprietary and inefficient C++ package (M++). In summary, CLOP's lack of speed, its programming errors, and some confusing defaults made the package unattractive for routine processing. However, CLOP enabled Dave Nichols (Lumley et al., 1994) to prototype a new multiple inversion scheme rapidly; a work that might not have been attempted without CLOP. Simultaneously, Dave Nichols and Lester Dye designed an improved class hierarchy that directly expressed the abstraction inherent in vector algebra.

Mark Gockenbach and Bill Symes independently developed a strikingly similar C++ library. In 1994, the efforts at Rice and Stanford merged into the Hilbert Class Library (HCL) (Schwab, 1994). Mark Gockenbach is HCL's primary author (Gockenbach and Symes, 1996) and maintains a detailed web-page (1996). Our new library, Jag, is a Java implementation of HCL's algebraic class hierarchy.

Lydia Deng (1996) of the Colorado School of Mines chose a very different approach, called COOOL, when developing a C++ system for geophysical inversion. Rather than casting the inversion in terms of fundamental algebraic expressions, such as vectors, she concentrated directly on more complex classes, such as objective functions and solvers.

Sergey Fomel (1996) implemented an object-oriented inversion library in Fortran90. While Fortran90 delivers high computational performance, it does not provide a classic object-oriented environment. Fortran90 modules are not classes and lack the notion of instances. Furthermore, Fortran90 does not provide inheritance. Nevertheless, Fomel successfully created Fortran90 solutions to all inversion problems in Jon Claerbout's book *Three Dimensional Filtering* (1994).

In the summer of 1996, SEP held an informal conference on Objected-Oriented

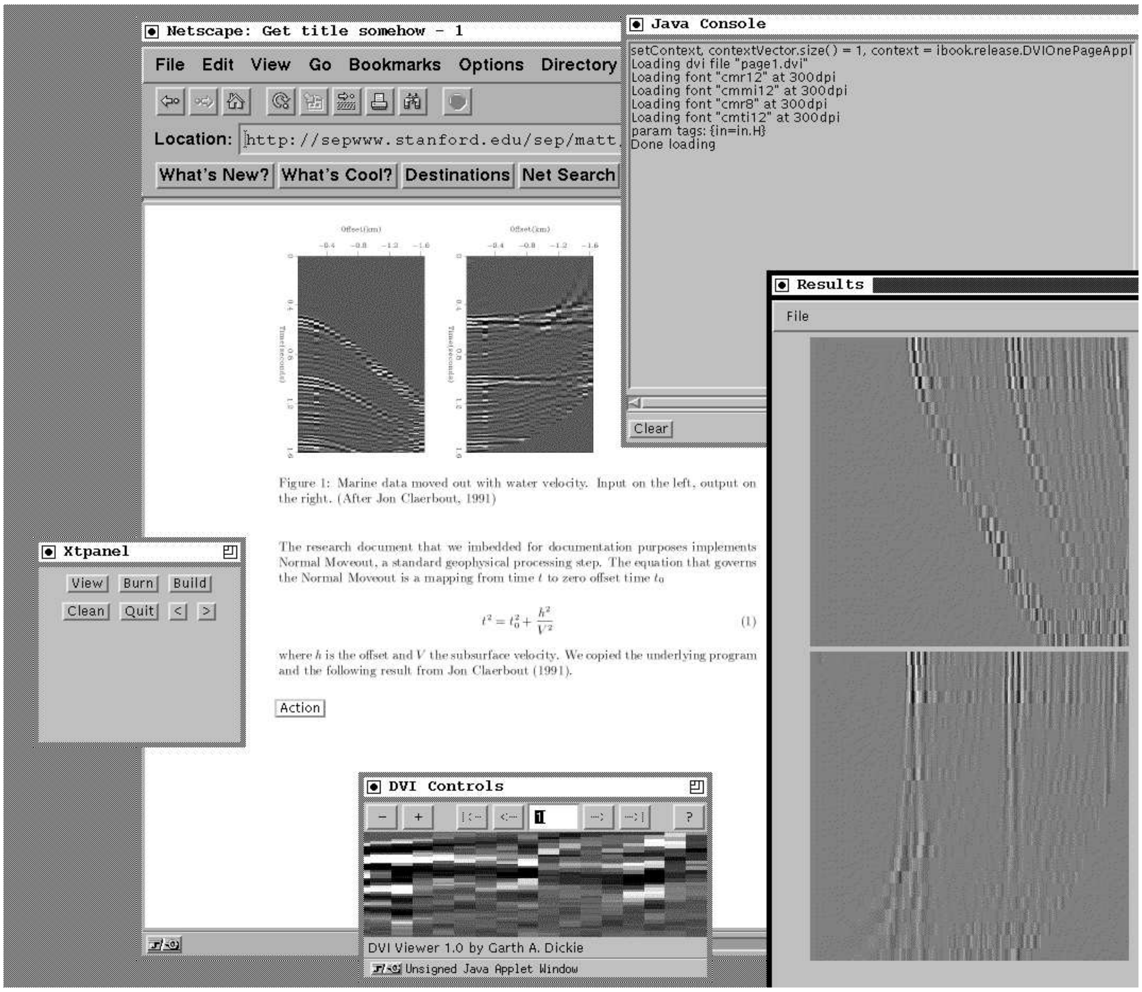

Figure 1: A Java DVI viewer, IDVI, displays an SEP research document in a Netscape browser. The window at the bottom center helps a reader to navigate the IDVI viewer. The Action button at the bottom of the page activates a Java xtpanel applet (left next to the button) that permits a reader to reproduce the document's result. The browser downloads the necessary Java programs from the server and executes them on the reader's machine. The Java console at the top right shows the executed commands. The final NMO result is displayed in the window at the bottom right. Soon we hope to be able to publish such reproducible World Wide Web documents routinely. matt4-idvi [NR]

Numerics (Claerbout and Biondi, 1996). Afterwards, several participants cooperated to develop a framework in which high-level, abstract HCL classes, (e.g. solvers), were combined with high-performance Fortran90 routines (e.g. operators) (Urdaneta and Karrenbach, 1996). The required linking of Fortran and C++ proved difficult but doable.

Independent of any geophysical inversion work, Ken Lenga and Bob Clapp (1995) wrote an impressive Java graphics package to display seismic data on the web. Unfortunately, their efforts were finally thwarted by backward incompatible changes in Java's AWT's library.

Currently, at SEP, most researchers use the traditional SEPlib library (C-Fortran) in their research. Researchers that work on large problems with irregular geometry additionally use the SEP90 library (C-Fortran90) (Biondi et al., 1996; Clapp and Crawley, 1996). A few researchers experiment with Sergey Fomel's Fortran90 library.

So why did we write another library? In the past, we actively participated in the development of HCL. We liked its algebraic class hierarchy, its object-orientation, and its opportunity of collaboration beyond SEP. However, C++'s complexity and Java's promises lured us one Sunday afternoon into starting a Java library for geophysical optimization.

OUR GOALS FOR JAG

We, of course, did not formulate our goals when we began work on Jag. At the time we were merely curious about Java and tired of C++. However, here are the goals that we should have had:

Jag should enable a researcher to

- **prototype** complex inversion problems,
- **scale** prototyped solutions to various data input sizes,
- **reuse** Jag code for different projects,
- **collaborate** with other researchers, possibly at remote sites or in other disciplines.
- **publish** results in various forms, particularly on the World Wide Web.

Did we succeed? We think we did, with the notable exception that solutions may not scale well to extremely large problem sizes, because of Java's inferior performance. We will revisit these criteria in our conclusion section.

PROGRAMMING ALGEBRA

Fundamentally, the Jag library, as well as its predecessors CLOP and HCL, is designed to find estimates for a Hilbert vector $\mathtt{x}$ subject to

$$\mathbf{A}\mathbf{x} \approx \mathbf{b}, \tag{1}$$

where $\mathbf{b}$ is a known vector, and $\mathbf{A}$ is a linear or nonlinear operator In general, Jag does not assume that $\mathbf{A}$ is square, a matrix, or full rank, that the data is consistent, over-determined, or under-determined. Furthermore, Jag does not assume that a solution is unique or that a solution even exists. Jag only requires that the problem is of the form stated in equation 1.

To address the problem $\mathbf{A}\mathbf{x} \approx \mathbf{b}$ a programmer needs three basic object types or classes: vector objects that contain the data $\mathbf{b}$ and the model $\mathbf{x}$, an operator object that implements the operator $\mathbf{A}$ and a solver object that, given the vector and operator objects, approximates a solution.

Vectors

In Jag, data, such as a seismic CMP gather, is represented as a *Vector* and a *Space* object. Any object of the Vector class implements the abstract algebraic definition of a Hilbert vector. Fundamentally, a Hilbert vector is able to add another (compatible) vector to itself, to scale itself by a given scalar, and to compute the inner product between itself and another vector. A Jag vector also contains a reference to its corresponding vector space object. Figure 2 shows the HTML documentation page for Jag's Vector interface. A Java utility program, *Javadoc*, generates such pages from an author's comments in the Java source file.

Any object of the Space class represents an abstract algebraic space. A Jag programmer can query if a given vector is a member of the space, or if another space is equal. A programmer can also request from a Space object to return a new vector member.

While it is true that most Jag vectors will store their data as an array of numbers and will represent their space as an array of axes or a set of key values, the notion of Jag vectors is not limited to such implementations. As we will discuss later, the vector and space classes are interfaces and Jag only specifies their method invocations.

Regularly sampled function

An example of a concrete vector is a SEP data cube, a set of real values on a regular grid. In Java, the *Rsf* (Regular Sampled Function) class stores a SEP data cube as a native Java array. An Rsf supports all vector methods that are implied by the

All Packages Class Hierarchy This Package Previous Next Index

---

Interface jam.vector.Vector

public interface **Vector**
extends Object
extends JamObject

A `Vector` is a mathematical vector with an inner product.
Warning: there is a potential conflict with `java.util.Vector`.
One should avoid using `import java.util.*;`
Import object from `java.util` on an individual basis, and never import `java.util.Vector`.
Instead use `java.util.Vector` in full where needed.

Method Index

o **add**(Vector)
  this = this + vec2
o **add**(Vector, Vector)
  this = vec1 + vec2
o **addScale**(float, Vector)
  this = this + a*vec2
o **addScale**(float, Vector, Vector)
  this = vec1 + a*vec2
o **copyFrom**(Vector)
  this = src (without reallocating this)
o **dot**(Vector)
o **getSpace**()
o **neg**()
  this = −this
o **neg**(Vector)
  this = −vec2
o **norm**()
o **norm2**()
o **scale**(float)
  this = a*this
o **scale**(float, Vector)
  this = a*vec
o **sub**(Vector)
  this = this − vec2
o **sub**(Vector, Vector)
  this = vec1 − vec2
o **zero**()
  this = 0 (the zero vector)

Figure 2: Javadoc, a Java utility program, generates HTML pages from Java interface and class files. The Vector HTML page lists the methods that a programmer needs to supply for each implementation of the Vector interface, such as an Rsf. Type declarations are hyperlinks to the type's Javadoc HTML page. This Figure shows only part of the more detailed original HTML page. $\boxed{\text{matt4-javadocVector}}$ [NR]

mathematical definition of a vector. For example, the expression `data1.add(data2)`[3] adds vector `data2` to vector `data1` by adding the underlying array elements. Or `data1.scale(alpha)` scales all elements of vector `data1` by the floating point number `alpha`.

The Rsf vector space, *RsfSpace* is an array of axes. Each axes object knows its number of samples `n1`, its increment `d1`, its offset `o1`, its physical `unit`, and a `label`. An RsfSpace can identify an equal vector space by checking the other spaces type (RsfSpace or not) and, in the case of an RsfSpace, by comparing their individual axes. Furthermore, an RsfSpace object gives access to its axes and their parameters.

In most cases, an Rsf and its space are created with the help of an existing Rsf or by invoking a method that reads a traditional SEPlib file from a file or input stream. In return, Rsfs can write their contents to a file or output stream in SEP IO format. Using SEPLib format allows us to use the existing SEPlib programs for preprocessing and graphical display. We also wrote a simple Matlab IO method.

The `progVector` on the current page class creates two Rsf, one from a local SEPlib file and the other one from an URL that points to a SEPlib file. The program adds the two vectors and scales the result. Finally, it writes the result to a local file.

The class fragment `progNmo` on page 370 implements the NMO operator's arithmetic core. In the beginning, the apply method queries various axis parameters from the vector space object of a given Vector `x`. Later the operator accesses the Rsf vector elements through array referencing.

```
package book.web;
import  rsf.vector.*;
import  java.net.*;
import  java.io.*;

public class progVector {
  public static void main(String[] args) {
    // Read data from a stanford URL and from a local file "cmp.HH"
    Rsf data1 = new Rsf();
    try {
      URL  data = new URL("http://sepwww.stanford.edu/matt/book/web/cmp.HH");
      data1    = RsfFactory.newRsf(data.openStream());
    } catch (Exception e) { System.err.println(e); }
    Rsf data2 = new Rsf("./cmp.HH");
    //  data1 = data1 + data2; data1 = data1 / |data1|
    data1.add(data2);
    data1.scale( 1f / data1.norm() );
    // Write data1 to a file "out.H"
    data1.write("./out.H");
  }
}
```

[3]In contrast to HCL, we adopted the Java convention to spell method and variable names lower case and to spell class names upper case. The only exception is the operator variable **A**, since we want it to resemble its mathematical format.

We feel that geophysical algorithms require simple data indexing for clarity and performance. Since Java does not allow the overloading of its indexing operator `[]`, we decided to permit *native* access to an Rsf's vector elements. Unfortunately, such native access violates the object-oriented goal of encapsulation; the class access depends on the classes internal representation. For example, we currently experiment with two Rsf versions: An *RsfF* uses a Java one-dimensional array to store all its elements. A programmer writes a simple arithmetic indexing expression, `dat(i1 + i2 * n1)`, to access an element. Alternatively, *RsfC* uses a multi-dimensional array and a programmer writes the standard indexing expression `dat[i2][i1]`. Additionally, both Rsf versions offer a programmer a pair of encapsulating indexing methods, `float getDatum(int[] index)` and `void setValue(int[] index, float value)`. We currently favor the multi-dimensional array implementation of Rsf.

Other primitive vectors

The simplest case of a vector is probably a single Real number. The Real class is useful since its objects represent the range of any functional, $f(\mathbf{x}) = y$. Functionals are formulated in many (but not all) minimization problems. A Real object contains a single Java float number and implements all mathematical vector operations. In Juice, all Real numbers have the same vector space. Alternatively, an author can implement a Real number class with physical dimensions as an Rsf with a single element and an Rsf space with a single axis.

Additionally, we implemented n-dimensional real vectors, Rn. Our Rn vectors can have a shape and are not necessarily a one-dimensional array of floats. We mostly use Rn for testing within the Juice part of Jag.

Jag also includes an experimental version of an Irregularly Sampled Function, *Isf*. The Isf class is best thought of as an array of vectors which themselves can be of Isf, Rsf, or any other vector type. The Isf handles most vector operations by invoking the corresponding methods of its elements. The parameters that describe each individual vector element are stored in a two dimensional table, a sort of object-oriented relational data base. The implementation of the parameter table permits a programmer to allocate additional keys and vector elements easily.

Composite vectors

Simple vectors can often be combined to build more complicated ones. The Isf implementation above is a composite vector since it is basically an array of vectors, but that implementation is hidden from the programmer.

On the other hand, the compositional character of a ProductVector, $\mathbf{x} = (\mathbf{x_1}, \mathbf{x_2})$, is not hidden from a programmer. A programmer can address the ProductVector as a single vector, $\mathbf{x}$ or he can address each individual vector $\mathbf{x_i}$ by itself. A ProductVector object contains the Cartesian product of the vector $\mathbf{x_1}$ and $\mathbf{x_2}$ and is handy when we need to formulate a minimization problem, such as

$$\begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{21}} \\ \mathbf{A_{12}} & \mathbf{A_{22}} \end{bmatrix} \begin{bmatrix} \mathbf{x_1} \\ \mathbf{x_2} \end{bmatrix} \approx \begin{bmatrix} \mathbf{b_1} \\ \mathbf{b_2} \end{bmatrix}$$

in the canonical form $\mathbf{Ax} \approx \mathbf{b}$. We will talk about composite operators such as $\mathbf{A}$ in the next section.

Operators

An operator object images an input vector to an output vector when its `image()` method is invoked. Furthermore, an operator knows about its domain and range, which are vector space objects.

For example, the linear NMO operator shifts a CMP gather's traces according to a hyperbolic time-offset relationship. An NMO input vector is an Rsf vector that contains the CMP gather. The operator's output vector is a moved-out CMP gather. The input and output space are identical except that the first axis of the output lists move-out corrected time, τ rather then true two-way traveltime, t. Figure 1 shows a reproducible World Wide Web document that invokes the Jag NMO operator. The class fragment `progNmo` on the next page implements an NMO operator's arithmetic core. The apply method combines the operator's forward and adjoint image operation in a combined loop, just as we are used to doing it at SEP. However, a programmer never invokes the apply method directly. Instead, he invokes the `image()` method of an operator or its adjoint.

```
public void apply(boolean add, Vector in, Vector out) {
  Rsf2   x = (Rsf2) in;
  Rsf2   y = (Rsf2) out;
  int   nx =  ((RsfSpace) x.getSpace()).getNSamples(1);
  float dx = (((RsfSpace) x.getSpace()).getAxes()[1]).delta;
  float ox = (((RsfSpace) x.getSpace()).getAxes()[1]).offset;
  int   nz =  ((RsfSpace) x.getSpace()).getNSamples(0);
  float dz = (((RsfSpace) x.getSpace()).getAxes()[0]).delta;
  float oz = (((RsfSpace) x.getSpace()).getAxes()[0]).offset;

  float[][] yy = y.d;
  float[][] xx = x.d;

  for (int ix=0; ix < nx ; ix++) { float xs = (ox + dx * ix) * 1f/vel  ;
  for (int iz=0; iz < nz ; iz++) { float zs =  oz + dz * iz;
    float ts = (float) Math.sqrt(zs * zs + xs * xs) + 1.e-20f;
    float wt = (float) (zs/ts * (1./Math.sqrt(ts)));
    int   it = (int) (.5 + (ts - oz) / dz);
    if (it < nz) {
      if(isFwd)
        yy[ix][iz] += xx[ix][it] * wt;
      else
        xx[ix][it] += yy[ix][iz] * wt;
```

```
    }
  }}
}
```

The NMO operator is often combined with the *Stacking* operator. The Stacking operator takes a 2-D Rsf object and sums element-wise over the second axis. The Stacking operator is linear, too, and its adjoint fills a 2-D Rsf with a copy of the adjoint's 1-D input Rsf.

Composite Operators

Similar to vectors, composite operators can be built from primitive ones. The *CompoundOperator* combines two operators so that the output of the first operator is the input to the second. For example, an NMO-and-Stack operator can be constructed as a Compound Operator that combines the NMO and Stack operator. Such a Compound Operator does not require any additional programming. A programmer can concentrate on implementing the simple constituent operators and can rely on the composite operators to build up more complicated operators properly.

The CompoundOperator type is not the only composite operator: The *Block Operator* arranges operators into operator matrices. For example, a Block Operator organizes four operators $\mathbf{A_{11}}, \mathbf{A_{21}}, \mathbf{A_{12}}, \mathbf{A_{22}}$ into an operator matrix

$$\begin{bmatrix} \mathbf{A_{11}} & \mathbf{A_{21}} \\ \mathbf{A_{12}} & \mathbf{A_{22}} \end{bmatrix}$$

In many cases, the programmer may be able to enhance the operator to compute additional information, such as the adjoint, the derivative, the gradient, or the Hessian. In general, such additional information allows a programmer to choose a more sophisticated solver. In Jag, every linear operator with adjoint automatically inherits its correct derivative, gradient, and Hessian method without the programmer having to add any additional code. If possible, composite operators combine their constituent's enhancements to enhancements of the composite. For example, the CompoundOperator of two linear operators with adjoint has an adjoint method as well.

In addition to the NMO operator, we wrote an entire family of convolution routines. We experimented with bilinear convolutions, recursive convolutions (for input and filters of any dimension), convolutions that are based on RsfC or RsfF, and convolutions that internally invoke native C subroutines. We also implemented a *Residual*, a *Norm*, and a composite *NormResidual* operator. These operators are useful for turning $\mathbf{Ax} - \mathbf{b}$ into a functional expression $f(x) = |\mathbf{Ax} - \mathbf{b}|$ when necessary.

Solvers

A solver's `solve(A, b, x)` method computes an estimate of the model $\mathbf{x}$ for the given operator $\mathbf{A}$ and a known data vector $\mathbf{b}$. To test Jag's ability to accommodate a wide variety of application-independent solvers, we implemented six solvers:

- Steepest Descent Solver
- Conjugate Gradient Solver (for linear operators)
- Conjugate Gradient Solver with pre-conditioning
- Conjugate Gradient Solver for nonlinear functionals
- Simplex Solver
- Simulated Annealing Solver

The program fragment of the Steepest Decent Solver in the Appendix demonstrates the simple but powerful abstract mathematical formulations that Jag's class hierarchy provides. The solver exclusively contains abstract interface objects and methods. Consequently, such a solver will work with *any* implementation of those interfaces.

We additionally devised a scheme that allows a programmer to supply his own report and termination methods to an iterative solver. Both methods are executed at the end of each iteration. The report method might print a statement about the state of the estimation and the termination method decides to continue the iteration or not. By supplying a report and termination method, programmer gains tight control over an iterative solver.

Finally, a *NormalEquations* class permits a programmer to conveniently set up the normal equations $(\mathbf{A}'\mathbf{A})\mathbf{x} = \mathbf{A}'\mathbf{b}$ for a given minimization problem $\mathbf{A}\mathbf{x} = \mathbf{b}$.

RESEARCH SCENARIOS

To test Jag during its development, we implemented a small set of research scenarios:

- **Deconvolution** finds the wavelet for a given reflectivity and seismic trace
- **Missing PE Filter** finds the coefficients for a Prediction Error (PE) from given input data
- **Missing Data (linear)** reconstructs missing data in a two-step linear approach. First, the filter is found where the data is known. Once the filter is known, the missing data is found with the PE filter from the first step. This missing data problem involves compounding the convolution operator with a

MaskingOperator that distinguishes the data locations where the data is known and where it is missing. We used two input data sets: a synthetic section and the Seabeam side-sweep sonar data from Jon Claerbout's book (1994).

- **Missing Data (nonlinear)** reconstructs missing data by a nonlinear approach. The filter and data are optimized simultaneously.
- **Coherency with Patching** computes a coherency attribute for a 3-D image volume. The process splits the image volume into subvolumes, finds a PE filter in each subvolume, filters the subvolume with the locally optimal PE filter, and finally merges the subvolumes back into a single image volume.
- **TvDecon** deconvolves a non-stationary time series. The process includes a leaking integration pre-processing step, and, since the problem is under-determined, a composition with the regularization operator, $\mathbf{Ix} \approx \mathbf{0}$.

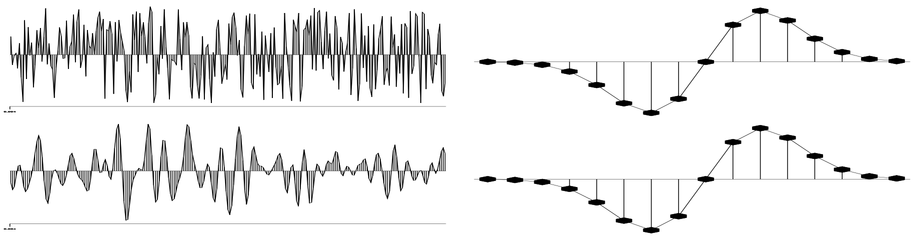

Figure 3: The signal at the upper left is a random reflectivity series. We compute the lower left signal by convolving the reflectivity series with a 16 point Ricker wavelet, which is shown in the upper right. We compute the lower right wavelet by inverting for an unknown 16 point wavelet given the two traces on the left. The CG solver finds the same zero-phase Ricker wavelet. matt4-decon [ER]

The code fragment `progDecon` on this page is the main routine of the deconvolution problem. The class instantiates two Rsf. One contains a random time series, the other one the result of a convolution of the time series with a 16 point Ricker wavelet. Next the program instantiates an internal convolution operator. Since we are going to use a Conjugate Gradient solver, we create a positive-definite operator by recasting the operator and vectors according to the normal equations

$$\begin{aligned} \mathbf{Ax} &= \mathbf{b} \\ \mathbf{A'Ax} &= \mathbf{A'b}. \end{aligned}$$

Finally, we instantiate a CG solver, solve for the unknown wavelet, and write the wavelet to a file. Figure 3 shows the input and the result of the inversion.

```
/**
 * <BR><!--#include file="./source.inc"--><BR>
 */

/**
```

```
* I solve b = M x for x.
* M is a    known convolution operator of the time series m (reflectiviy).
* x is an unknown filter (wavelet).
* b is a    known time series (seismic trace).
*
* I create the normal equations: b_L = M'b and L = M'M and
* call a CGSolver to solve M'b = M'M x or b_L = L x.
*/
public class IcafDecon {
  public static void main(String[] args) {
    Rsf trace = new Rsf("./trace.H");
    Rsf rflct = new Rsf("./rflct.H");

    hasAdjoint icaf        = new IcafFactory(rflct, trace.getSpace());
    Vector     traceNormal = icaf.adjoint().image(trace);
    CompoundLinearOperatorAdjoint
               icafNormal  = new CompoundLinOpAdj(icaf, icaf.adjoint());
    // Create an initial wavelet
    Rsf wave = (Rsf) icafNormal.getDomain().newMember();
    wave.zero();
    // Solve with CG solver and default solver settings
    new    CGSolver().solve(icafNormal, traceNormal, wave);
    // Write
    wave.write(System.out);
  }
}
```

WHY OBJECT ORIENTATION?

To implement Jag, we needed abstraction, encapsulation, and inheritance. Perhaps a similar package could be implemented in a more procedural language, but we do not know how. Abstraction enforces the collaboration of Jag's diverse classes. Encapsulation gives each Jag programmer a maximum of independence when implementing a class. Finally, inheritance permits a programmer to implement and maintain the bare minimum of code.

Abstraction

Jag objects interact in very intricate ways: a solver may need to execute the image function of an operator $\mathbf{A}$ and add an update to the estimated solution $\mathbf{x}$. The solver, as well as its programmer, has no knowledge of the internals of $\mathbf{A}$ or $\mathbf{x}$. Obviously not every solver will work with every operator, nor will every operator be able to image every vector. It is important that a programmer and a compiler can easily verify if a set of objects will collaborate. A solver should also operate with as many operators and vectors as possible. Furthermore, a solver should not only collaborate with the appropriate *existing* operators or vectors, but with *any* appropriate operator or vector, even future ones.

We use Java's interfaces to separate solver, operator, and vector classes, despite their sophisticated interactions. Java interfaces, not unlike C++ virtual classes, define a set of method prototypes. Any class that implements an interface has to implement all the interface's methods. Any class that interacts with implementations of an interface can rely upon the fact that any particular implementation offers all the methods listed in the interface. Interfaces are Java's purest form of class abstraction.

We isolated Jag's most important interfaces in a separate package called Jam. These interfaces specify the methods expected from Jag's mathematical entities. For example, all the operators are derived from the interface *jam.operator.Operator* and therefore implement the interface method `public Vector image(Vector d)`. Similarly, all Jam vectors are derived from an interface called *jam.vector.Vector* and, consequently, implement a method `public void add(Vector vec2)`. As long as any implemented class, e.g., a solver, restricts itself to methods listed in Jam's interfaces, then such a class will find the methods it expects.

Encapsulation

Jag requires that a vector performs all the operations that are part of the algebraic vector definition. For example, a vector can add itself to another vector of the same vector space or a vector can scale itself with a scalar. Similarly, a Jag operator can apply itself to a compatible input vector and return an output vector. The curious phrase that an inanimate vector *adds itself* or *scales itself* expresses the fact that the object-oriented implementation of the addition is carried by each object. Object orientation allows us to wrap algorithms and data structures into functional units, such as vectors that add themselves. Consequently, a solver does not need to know about the implementation details of a vector's addition method, but merely trusts that the standard `add()` method invokes the correct computations that result in an addition.

Inheritance

Inheritance allows a programmer of an object-oriented class hierarchy to share code: any subclass inherits all methods and variables of its super class. The subclass may overwrite the inherited methods or use them unmodified.

In Jag, linear operators inherit almost all their methods from a super class *SepLinearOperator*. A programmer usually only implements the operator-specific parts, such as

- the **constructors**, which cannot be inherited,
- the **apply() method**, which implements the forward and adjoint operation, and

- the **toString() method**, which returns a Java string representation of the object.

SepLinearOperator supplies methods to image the operator or to access the operators adjoint, its gradient, its Hessian, and more.

WHY JAVA?

Java is simple to learn and to use. Jag's predecessors, HCL and CLOP, were based on C++, a language whose basics are considerably more difficult to learn than Java.

Why is Java easier then C++? Java uses built in types and references but no pointers. Java has built in garbage collection. Java's interfaces allow multiple inheritance. Java has no C legacy features and imitates the C and C++ syntax. Java has no header files but its own package inclusion scheme. Java has some high level programming features such as hash tables or dynamic arrays. Java's strong typing and its array bound checking help a programmer to find bugs early.

Furthermore, Java's naming scheme is intuitive. Java's decision to derive every class from a single root class, *Java.Object*, gives the package a conceptual consistency. However, Java includes wisely a few primitive data types, such as floats or integers, for which an object implementation is often inefficient and superfluous.[4] Finally, Java wraps object-orientation, a window-, and a networking-toolkit into a single language.

Overall, we had fun coding Java. We hope that Jag will enable us to program software solutions that we would not attempt in a less sophisticated (Fortran) or more complicated (C++) programming language. However, Java undergoes rapid developments currently, and it is difficult to guess where it might go.

WHY NOT JAVA?

Java is slow. Consequently, programs implemented in Java are slow. We have not done any serious performance measures, nor have we tried to finalize and optimize Jag. Standard Java is about 30-60 times slower than C when interpreted. When Java is compiled by a just-in-time compiler (jit), it seems to be about 3-6 times slower then C. Java compiler technology is improving rapidly, but how efficient it can get relative to other languages is unclear. Hank Schiffman (Schiffman, 1997) believes that Java's native code will be about as fast as C++ in most cases, but hardly ever faster.

For the time being, we find Java's just-in-time performance acceptable for prototyping small- to medium-sized problems. However, for large field data examples, we

[4]Unfortunately, Java does not have a primitive complex data type. The word complex is not even reserved for future use.

may have to re-implement our application in a better performing language, after we developed it in Java.

A different reason to shun Java is its current rapid growth and change. Java is not standardized and past releases changed sufficiently to require considerable re-programming of already existing applications. In particular, Java's window toolkit, AWT, has been unreliable.

The development of Jag was not affected by changes in Java. In general, Jag relies on parts of Java that are traditional components of any computer language, such as arithmetic, string handling, stream IO, etc. We resisted the temptation to use more original Java features, such as its graphics and network capabilities. We hope, however, that soon we will be able to exploit these features as Java evolves and SUN begins to freeze the language's core. During the development of Jag, we changed between minor upgrades of Java without any complications. We developed our software on two platforms, Linux and SGI, without experiencing any portability problems. We are confident that the up-coming major upgrade to Java 1.1 will pose no problems for Jag.

SUMMARY AND CONCLUSIONS

Jag translates the abstract mathematics of inversion into a set of Java classes. It shares the algebraic class hierarchy with the HCL, a C++ library. We found Java a more elegant and flexible programming language than C++. In particular, C++'s lack of garbage collection and multiple inheritance burdens the programmer with very Gordian and pedestrian tasks. It is too early to decide if Jag will fulfill the goals that we mentioned earlier. However, here are a few preliminary comments:

- We implemented a large set of Jag classes in a relatively short time[5] and we are confident Jag will be an excellent library for prototyping.
- We have not yet used Jag for a large scale, geophysical inversion problem and we doubt that Jag's current performance would be competitive with the performance of alternative languages. However, we have not optimized Jag nor investigated Java's budding high-performance efforts.
- Currently, Jag reuses code very efficiently. In particular, the solvers are able to work with a wide range of vectors and operators.
- Jag's interface hierarchy and its Javadoc representation explicitly communicates to a programmer the requirements that any new class has to meet. The Java compiler enforces these requirements. We are confident that Jag will prove suitable for local as well as remote collaboration.

[5] the official Jag package currently comprises about 200 class and interface files.

- Finally, we feel that Jag's ability to publish computational research on the World Wide Web is the icing on the cake. We hope that long distance collaboration and exchange will become easier and we are looking forward to publishing reproducible documents on the Web.

ACKNOWLEDGEMENT

Martin Karrenbach brought IDVI to our attention and is eager to give Jag a try. Scott Morton of Cray-SGI boosted our optimism about Java's future performance. Mark Gockenbach, Dave Nichols, Bill Symes, Lester Dye, Hector Urdaneta, and Martin Karrenbach were directly or indirectly part of the design and implementation of HCL, Jag's older C++ cousin.

REFERENCES

Biondi, B., Clapp, R., and Crawley, S., 1996, Seplib90: Seplib for 3-D prestack data: SEP–**92**, 343–364.

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.

Clapp, R. G., and Crawley, S., 1996, SEPF90: SEP–**93**, 293–304.

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

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

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.math.lsa.umich.edu/˜gock/objects.html.

Lenga, K., Clapp, R. G., and Claerbout, J. F., 1995, Java: SEP–**89**, 227–248.

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.

Schiffman, H., 1997, Thoughts on Java: http://reality.sgi.com/shiffman/Java-QA.html.

Schwab, M., and Schroeder, J., 1995, Reproducible research documents using GNU-make: 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.

Shewchuk, J. R., 1994, An introduction to the conjugate gradient method without the agonizing pain: ftp://reports.adm.cs.cmu.edu/usr0/anon/1994/CMU-CS-94-125.ps.

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

APPENDIX

The `solve()` method of Jag's Steepest Decent Solver is almost literally translated from Shewchuk excellent paper (1994).

```
public void solve(Operator A, Vector b, Vector x ) {
  if (! (A instanceof LinearOperator)) {
    System.err.println("Error in SteepDescentSolver.solve(): " +
                       "operator A must be a LinearOperator");
  }

  this.A = A;
  this.b = b;
  this.x = x;

  // initialize the solver
  Vector v   = A.getRange().newMember();
  Vector res = A.getRange().newMember();
  A.residual(x,b,res);
  res.neg();

  rtr      = res.norm2();

  // iterator is supplied to the solver by the programmer
  iterator.start();

  while(iterator.hasMoreIterations()) {  // iterator decides about stopping
    doOneStep(v);
    iterator.nextIteration();            // iterator reports about iteration
    iter++;
  }
  iterator.finish();
```

```
}

protected void doOneStep(Vector v) {
  A.image(res, v);
  // compute step length alpha
  float rtv   = res.dot(v);
  float alpha = rtr/rtv;
  if (rtv == 0f) System.err.println("Error in SteepDescentSlv: " +
                                    "divide by rtv (=0).");
  // update x and r
  // every reset-th iteration I recalculate the exact residual to
  // remove the accumulated floating point error.
  x.addScale(  alpha, res);
  if (((iter+1) % reset) == 0) {
    A.residual(x,b,res);
    res.neg();
  }
  else {
    res.addScale( -alpha, v);
  }
  rtr = res.norm2();
}
```