Stanford Exploration Project, Report 93, November 11, 1997, pages 263–273

Stanford Exploration Project, Report 93, November 11, 1997, pages 263–273

Short Note

HCL and regular data

Joel Schroeder and Matthias Schwab[1]

keywords: *deconvolution, nonlinear, missing data, linear, random*

INTRODUCTION

To support traditional SEPlib processing capabilities within the HCL (Hilbert Class Library) framework, we implemented a regular data class, a series of simple operators, and a few of Jon Claerbout's solvers.

The multi-dimensional regular data grid of an SEP Data Cube is implemented as an RGF (Regular Grid Function) class. In HCL's linear algebra paradigm, the grid size and its physical units define a vector space. Since the RGF class is derived from HCL's abstract vector class, we implemented the vector operations. We used existing I/O routines to ensure that RGF files are backward compatible with SEPlib files.

We coded a few simple linear and bilinear operators which act on RGFs. A recursive convolution operator works with data and filter of arbitrary dimension. A Fortran77 version, which is invoked from C++, runs faster than the strictly C++ version.

Finally we solved one of Claerbout's (1996) missing data and unknown filter problems with two different methods. The first method was a two-step data infill, in which our results agreed with Claerbout's. With the second method, using Mark Gockenbach's (1996) seasoned nonlinear minimization routine, we obtained accurate results without the need to understand the internals of his code.

REGULAR GRID FUNCTION

To initiate work at SEP within the HCL framework, we implemented the traditional SEP Data Cube as a vector class. We named the class RGF for regular grid function. Each instance of an RGF has an associated RGFSpace, which is a vector space in HCL.

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

An RGFSpace is defined by its dimension, and an Axis for each dimension. An Axis is a C++ class which has five attributes. These include: number of samples, offset of the first sample, distance between samples, and unit of measurement (in which the offset and distance are measured). The fifth attribute is a descriptive label, e.g. "geophone position". Thus, an RGFSpace consists of an integer dimensionality, and an array of Axis objects specifying the physical shape of the grid.

Along with the RGFSpace to which it belongs, an RGF contains an array of ordinates, implemented as a one-dimensional array. The user, however, can index the array as if it were multidimensional. We achieved this by using the C++ feature of overloading the parentheses operator.

Because an RGF is a vector in HCL, we coded the requisite vector operations. These include vector addition and subtraction, multiplication by a scalar, norm, and inner product. We also implemented I/O routines using the SEPlib C functions `sreed` and `srite` so we could use existing data files in the SEP Data Cube format.

RGF OPERATORS

The next step was to write a few simple operators. We coded the four basic convolution operators used at SEP: `tcai`, `tcaf`, `icai`, and `icaf`. These acronyms indicate convolution in which the boundary conditions are either transient or internal and in which the adjoint gives either the filter or the input.

We further realized that `tcai` and `tcaf` can be combined into a single bilinear operator. A bilinear operator takes two vector arguments and returns a vector value which is linear in each of the arguments considered by itself. We wrote a transient convolution class which is a subclass of HCL's bilinear operator class. In so doing, we unified the code for `tcai` and `tcaf`. (Likewise, we unified `icai` and `icaf` into a single internal convolution bilinear operator). Jon Claerbout stresses that the key code segments of an operator's forward and adjoint should be as close together as possible. However, by writing both `tcai` and `tcaf` as separate routines, he duplicates the forward operation and isolates the two adjoints in separate routines. Our transient convolution operator encapsulates the forward and both adjoints in a single routine. The following is pseudo-code for the key part of our transient convolution bilinear operator in one dimension:

```
% Pseudo-code for transient convolution as a bilinear operator (in 1D)

do fi = 0,m-1          % Loop over the filter components (index begins at 0)
do xi = 0,n-1          % Loop over the input  components
   yi = xi + fi        % Calculate the corresponding output components
   if (result = output) then y(yi) += f(fi) * x(xi)   % forward
   if (result = input ) then x(xi) += y(yi) * f(fi)   % adjoint of tcai
   if (result = filter) then f(fi) += x(xi) * y(yi)   % adjoint of tcaf
end do
end do
```

Another feature of our convolutions is that they are written recursively. This allows convolution in any dimension without having to write separate routines. Recursion, however, is usually a speed bottleneck. An alternative would be to write a single iterative (i.e. non-recursive) version in a sufficiently large dimension. That is, because of our indexing mechanism of RGFs in C++, one could use (iterative) convolution code written for ten dimensions on a filter, input, and output of only two dimensions.

Furthermore, we have envisioned a more general convolution operator in which the boundary conditions (transient or internal) are specified for each dimension individually. We believe that this would be easy to implement with a recursive algorithm, although we are not sure if there is a demand for such an operator. Even more general would be a convolution in which the boundary conditions are specified at each boundary; however, making the code elegant and efficient might be difficult. Nonetheless, we could write it in C++ whereas in Fortran, because of restrictions on recursion, it would be quite difficult.

In addition to convolutions, we wrote masking operators, which zero specified components of an RGF. These are important in some of Claerbout's missing data problems.

SOLVERS

One of the most promising aspects of HCL is that general solvers can be written and used on any vector. The same code can be used to solve problems on regular data, irregular data, or any other vector type.

We wrote a routine, `NormalSolve`, which solves the normal equations, $\mathbf{L}^T\mathbf{L}\mathbf{x} = \mathbf{L}^T\mathbf{b}$. Its arguments include an `HCL_LinearOpAdj`, $\mathbf{L}$, and an `HCL_LinearSolver`. An `HCL_LinearOpAdj` is a linear operator in which the adjoint as well as the forward have been implemented. A `HCL_LinearSolver` estimates $\mathbf{x}$ in $\mathbf{L}\mathbf{x} = \mathbf{b}$, when its member function, `Solve( L, b, x )` is invoked. `NormalSolve` is quite general, since it can estimate $\mathbf{L}\mathbf{x} = \mathbf{b}$ for *any* linear operator with adjoint, $\mathbf{L}$, by recasting it into normal form. Typically, we pass HCL's conjugate gradient solver as the solver used by `NormalSolve`. Finally, since the normal equations are so general, we probably should have written `NormalSolve` as a `HCL_LinearSolver` instead of a stand-alone function.

```
// Solve L'Lx = L'b using a given solver, Slvr (typically cojugate gradient)

void NormalSolve(       HCL_LinearOpAdj  &L,
                  const HCL_Vector       &b,
                        HCL_Vector       &x,
                        HCL_LinearSolver &Slvr )
{
    // Form the normal as a compound linear operator (with adjoint), L'L
    // The zeroes indicate that the caller will delete L and L.Adjoint()
```

```
  HCL_CompLinearOpAdj   LtL( (HCL_LinearOpAdj *) & L            , 0 ,
                             (HCL_LinearOpAdj *) &(L.Adjoint()), 0 );
    // Allocate space for L'b
  HCL_Vector          * Ltb = L.Adjoint().Range().Member();

  L.Adjoint().Image( b, *Ltb );        // Calculate L'b
  Slvr.Solve( LtL, *Ltb, x );          // Solve L'Lx = L'b
  delete Ltb;
}
```

Mark Gockenbach's nonlinear minimization class, `HCL_UMin_lbfgs`, is quite general and powerful (1996). The user specifies a (real-valued) functional and its gradient. The member function `Minimize( f, x )` implements a limited memory Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm to find an $\mathbf{x}$ which minimizes f. The algorithm is an unconstrained minimization and is especially suited to large-scale problems. We used it successfully to solve a nonlinear missing data problem.

MISSING DATA AND UNKNOWN FILTER PROBLEMS

We tested our operators by implementing one of Claerbout's missing data and unknown filter problems. The data is a superposition of two plane waves in two dimensions with a large elliptical region missing (see Figure 1).

Two-stage data infill

We used the two-stage data infill method (Claerbout, 1996). Our results are similar to Claerbout's; the interpolated data shows a loss of power near the edge of the rectangle (see Figure 1). Claerbout's Ratfor implementation uses two routines, `pef` and `mis`, which solve for the filter and the data, respectively. The abstraction of C++ and the HCL framework allows a single routine to be used for both the data and the filter:

```
// Solve the missing data problem LMx = -Lk, that is L(Mx + k) = 0
// k contains the known data (and is zero where unknown)
// M is a mask operator which zeroes the known components
// MissSolve can be used for finding missing data or an unknown filter

void MissSolve(
           HCL_LinearOpAdj  &L,
     const HCL_Vector       &k,  // known data with zeroes where unknown
           HCL_Vector       &x,  // initial guess (should be pre-masked!)
                                 // on exit x contains u, the unknown data
           HCL_LinearOpAdj  &M,  // mask which zeroes the known data
           HCL_LinearSolver &S ) // solver used with the normal equations
{
  HCL_Vector *b = L.Range().Member();    // Allocate space for b
  L.Image( k, *b );                      // Set     b =  Lk
  (*b).Neg();                            // Set     b = -Lk
```

```
  HCL_CompLinearOpAdj A( &M, 0, &L, 0 ); // Set     A =  LM
                                         // Solve LMx = -Lk
                                         // (using the normal equations)
  NormalSolve( A, *b, x, S );
  delete b;
}
```

The first step of two-stage data infill is to create two RGFs which consist of zeroes and ones. The first has the shape of the data region, but with ones where the data is known. The second has the shape of the prediction-error filter, but with ones where the filter values are modifiable. Internal convolution is then performed with these two RGFs to give an RGF which we call the influence mask. Then we use `MissSolve` to find the prediction-error filter using a linear operator which is the composition of two operators. Specifically, it consists of `icaf` with the known data, followed by masking with the influence mask. (Notice that `icaf` is an operator which takes a filter as the argument).

After finding the filter, we fill in the missing data by again using `MissSolve`. This time, the operator is `tcai` with the newly-calculated filter. We employed HCL's conjugate gradient solver to find both the filter and the missing data.

Nonlinear minimization

We solved the same problem using Gockenbach's `HCL_UMin_lbfgs`. The functional we minimized, was the norm-squared of internal convolution of the data-set with the prediction-error filter. To use the BFGS solver, we had to construct the functional to take a single vector argument. Thus, we combined the data-set and filter into a `HCL_ProductVector`. Specifically, we minimized the functional

$$g\left(\left[\begin{array}{c}\mathbf{u}_d\\ \mathbf{u}_f\end{array}\right]\right) = \|\mathbf{B}(\mathbf{M}_d\mathbf{u}_d + \mathbf{k}_d, \mathbf{M}_f\mathbf{u}_f + \mathbf{k}_f)\|^2$$

Here, $\mathbf{B}$ represents internal convolution, and the subscripts f and d represent the filter and the data, respectively. For both the filter and the data, there are known components, $\mathbf{k}$, which are given, and unknown components, $\mathbf{u}$, for which we solve. Finally, the $\mathbf{M}$'s represent masking operators which zero the known components of the data or filter. Thus, we find a filter, $\mathbf{u}_f + \mathbf{k}_f$, and data, $\mathbf{u}_d + \mathbf{k}_d$, which are constrained to contain known values. A prediction-error filter is most useful in interpolating missing data, when it nearly zeroes the known data under convolution. That explains why we minimize the norm of $\mathbf{B}$. To keep the resulting filter from simply being zero, we constrain it to contain a 1 by setting $\mathbf{k}_f$ as appropriate for a prediction-error filter.

Because the functional, g, is based on a bilinear operator, $\mathbf{B}$, using a general nonlinear minimizer is probably unwarranted. Claerbout has a nonlinear solver which takes advantage of the bilinear property of convolution. His results for this synthetic data problem appear similar to ours. Both appear to approximate the original data very well. So, although we probably applied a solver which was too general, we learned that it is relatively easy to use `HCL_UMin_lbfgs`. We envision applying it successfully to future problems involving more complicated functionals. Our results are shown in Figure 1.

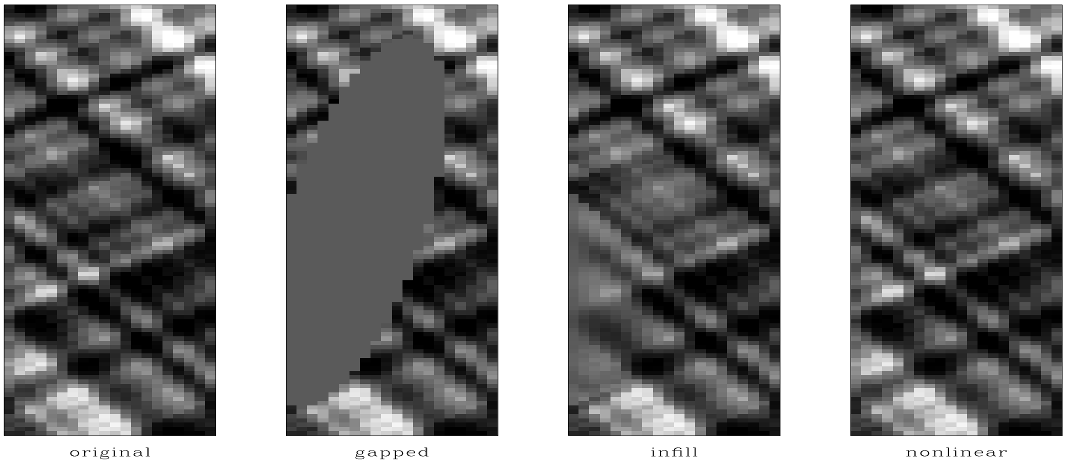

Figure 1: The missing data and unknown filter problem. Original shows the synthetic data. Gapped is the known data given to the solvers, $\mathbf{k}_d$. The two-stage data infill method shows a lack of power near the edge. The nonlinear functional minimization method shows results visually identical to the original data. joel1-interp [CR]

It should be noted that although our nonlinear solver's result is more accurate than the infill method's, it took an order of magnitude longer to calculate. In fact, had we let the infill solver run for more iterations, it may have converged to a better answer.

LINKING WITH FORTRAN77

We implemented the two-dimensional `tcai` operator in the HCL framework by calling Claerbout's Fortran77 routine `tcai2`. It was a simple exercise, and the relevant portions of code are shown below. (Notice that this example, being `tcai` alone and not combined with `tcaf`, does not emphasize the bilinearity of transient convolution. That is, the forward and only one of the two adjoints are coded together here.)

```
// if TCAIF77 is not defined, the C++ version is compiled
// if TCAIF77 is     defined, the F77 version is compiled

#ifdef TCAIF77
extern "C" void tcai2_(int &adj, int &add,
                       float *aa, int &na1, int &na2,
                       float *xx, int &nx1, int &nx2,
                       float *yy, int &ny1, int &ny2
                       );
```

```
#endif

void tcai2D::apply(int adj, RGF &xx, RGF &yy) const {
  int m1 = filter.GetSpace().GetNSample(0);
  int m2 = filter.GetSpace().GetNSample(1);
  int n1 =     xx.GetSpace().GetNSample(0);
  int n2 =     xx.GetSpace().GetNSample(1);

#ifdef TCAIF77

  int l1 =     yy.GetSpace().GetNSample(0);
  int l2 =     yy.GetSpace().GetNSample(1);
  float *a = (float *) filter;
  float *x = (float *) xx;
  float *y = (float *) yy;
  int zeroAdd = 0;

  tcai2_( adj, zeroAdd,
          a, m1, m2   ,
          x, n1, n2   ,
          y, l1, l2   );
#else

  if (adj==0) { yy.Zero(); }
  else        { xx.Zero(); }

  for (int a2=0; a2<m2; a2++){ for (int x2=0; x2<n2; x2++){ int y2 = x2+a2;
  for (int a1=0; a1<m1; a1++){ for (int x1=0; x1<n1; x1++){ int y1 = x1+a1;
    if (adj==0)
      yy(y1,y2) = yy(y1,y2) + xx(x1,x2) * filter(a1,a2);
    else
      xx(x1,x2) = xx(x1,x2) + yy(y1,y2) * filter(a1,a2);
  }}}}
#endif
}
```

DOT PRODUCT TEST OF ADJOINT OPERATORS

The adjoint, A, of an operator, F, is generally defined as satisfying $\mathbf{x} \cdot A\mathbf{y} = \mathbf{y} \cdot F\mathbf{x}$ for all vectors $\mathbf{x}$ and $\mathbf{y}$ (in the appropriate vector spaces). (Here, the dot denotes the dot product.) SEP has traditionally used this definition to test if its operators are coded correctly. Thus, if the implementation of an operator's forward and adjoint satisfy the equality when $\mathbf{x}$ and $\mathbf{y}$ are random, we trust that they are indeed adjoints.

We wrote the dot product test in HCL. It can test any `HCL_LinearOpAdj`. Unfortunately, the user must supply the random vectors, since there is no provision for a random vector in HCL. There is no general procedure mathematically to choose a random vector from a vector space *in general.* However, we provided RGF with the method `Random()`, which fills the RGF with random numbers. Therefore, we could write a dot product test which requires only the operator, as long as it acts on and returns RGFs.

```
/*
  DotProductTest checks if a HCL_LinearOpAdj is coded correctly in the
  sense that the implementations of the forward and adjoint are actually
  adjoints of each other.

  Since there is no general random vector concept in HCL_Vector,
  the user should provide random vectors w and v
  in the domain and range of FWD, respectively.
*/

void DotProductTest(       ostream         & ostr, // stream to send output
                           HCL_LinearOpAdj & FWD , // operator to test
                     const HCL_Vector      & w   , // random vector in domain
                     const HCL_Vector      & v   , // random vector in range
                     int   verbose               ) // flags output verbosity
{
  if (verbose) {                           // if verbose: display FWD, w, v
    ostr << "DOT PRODUCT TEST\n"
         << "----------------\n"
         << "Operator (A):" << endl;
    FWD.Inspect(ostr);
    ostr << "Vector 1 (w):" << endl;
    w.Inspect(ostr);
    ostr << "Vector 2 (v):" << endl;
    v.Inspect(ostr); }

  HCL_Vector * FWDw = v.Space().Member();  // allocate space for (FWD )w
  HCL_Vector * ADJv = w.Space().Member();  // allocate space for (FWD')v

  FWD.          Image(w, *FWDw);           // calculate (FWD )w
  FWD.Adjoint().Image(v, *ADJv);           // calculate (FWD')v

  float vFWDw = v.Inner(*FWDw);            // calculate v'(FWD )w
  float wADJv = w.Inner(*ADJv);            // calculate w'(FWD')v

                                           // display results
                                           // ideally, v'(FWD)w = w'(FWD')v

  ostr << "RESULTS:"                       << endl;
  ostr << "<v, A (w)> = " << vFWDw         << endl;
  ostr << "<w, A'(v)> = " << wADJv         << endl;
  ostr << "Difference = " << vFWDw - wADJv << endl;  // should be small
}
```

CONCLUSIONS

We believe that all (or most) classes in HCL should be derived from a single generic class, `HCL_Object`. This could help enforce a similar format for the `Inspect()` method, which display the attributes of an object. Additionally, we might in the future want some other general functionality. Since no such base class exists in HCL, we created `SEP_Object`. Unfortunately, this required the use of multiple inheritance. We would not need multiple inheritance if HCL provided a general base class. HCL rejects multiple inheritance, since it can be unreliable and difficult to use in C++. Each `SEP_Object` includes a name and a description and the method `SEP_Inspect()`, which outputs descriptive information about the object. In this way, all objects derived from `SEP_Object` are guaranteed to have the `SEP_Inspect()` method. This helps to enforce a system with diagnostic, and therefore, debugging, tools.

In addition to a generic base class, we hope a future version of HCL will include `Gaxpy()` and `Residual()` methods for linear operators. The gaxpy (general alpha x plus y) operation applies an operator, $\mathbf{A}$, in the fashion $\mathbf{x} = \mathbf{A}\mathbf{y} + \mathbf{x}$. Likewise, the residual is calculated as $\mathbf{r} = \mathbf{A}\mathbf{x} - \mathbf{b}$. Both of these calculations are often used in solvers and are usually capable of being coded to run faster than simply applying the operator and adding (or subtracting) a vector. In the worst case, the trivial implementation of applying the operator and adding could be inherited from a base operator class.

From our experience, HCL is powerful in its simplicity and well designed. A few aspects of C++, however, make it more difficult to use. As much as we have used pointers through the years, they are often a little tricky and they usually make code less readable. Furthermore, although the `const` specification for parameters in C++ allows the compiler to make certain optimizations, we find that worrying about exactly when to use it can be a nuisance. In summary, we believe that the HCL framework, because of its abstraction, allows us to write modular code which functions as "interchangeable parts" in a way which previous systems do not.

ACKNOWLEDGMENTS

Dave Nichols and the SEPlib crowd developed and maintain the I/O routines which underlie the RGF file access. Mark Gockenbach and Bill Symes shared their HCL solvers and HCL/Fortran77 examples with us.

REFERENCES

Claerbout, J. F., 1996, Three-Dimensional Filtering: Environmental soundings image enhancement: http://sepwww.stanford.edu/public/docs/tdf/.

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.