Stanford Exploration Project, Report 93, November 11, 1997, pages 273–292

An IGF90 tutorial

Hector Urdaneta and Martin Karrenbach[1]

keywords: *linear, interpolation, algorithm, optimization*

ABSTRACT

We encapsulate SEP's Fortran90 data structure for handling data with an irregular spatial sampling (SEPlib90) in a C++ class representation (IGF90) and provide it with an interface to a C++ library of inversion and optimization algorithms. The IGF90 class provides a mechanism for users to take advantage of the efficient Fortran90 data structures and the object-oriented programming paradigm of C++. We present different examples to illustrate the IGF90 class user interface. We use the IGF90 class and the C++ library of optimization algorithms to solve a missing data problem given a roughening filter. We discuss the design of the C++/Fortran90 interface, and illustrate it with the C++ interface to SEP's Fortran90 data structure and the interface to a Fortran90 operator.

INTRODUCTION

Many mathematical algorithms in inversion and optimization are based on the existence of vectors, which populate vector spaces. One particular and often occuring space is the "Hilbert Space". An abstract mathematical C++ Framework that implements "Hilbert Vectors and Vector Spaces" has been implemented by Mark Gockenbach and Bill Symes (http://www.math.lsa.umich.edu/∼gock/HCL.html). The Hilbert Class Library (HCL) (Schwab and Urdaneta, 1996) is a collection of (mostly) abstract base classes which are specifically designed to serve as an interface to classes derived from them. These abstract base classes define the common structure of (and include) the minimal set of objects necessary to perform optimization in Hilbert Space. The library can only be used by deriving classes from the base classes. To derive a class from a base class, one needs to implement (fill up) the interface of the base class.

There are three fundamental classes defined in the HCL class library, a vector class, a general operator class, and a solver class:

[1]**email:** hector@sep.stanford.edu, martin.karrenbach@physik.uni-karlsruhe.de

- The vector class obeys the mathematical definition of vector operations, such as vector addition and scaling. Examples of geophysical vectors are a complex 5-dimensional *(x, y, t, offset, azimuth)* irregularly-sampled 3-D prestack volume, and a simple well log.

- The operator class represents operator mapping from one vector space to another. The normal-moveout correction (NMO) is an example of a geophysical linear operator.

- A solver will take an operator and a vector and compute the minimization problem associated with the operator. Solvers are implemented by means of the mathematical properties of vectors and linear operators only.

An example of a vector class is the regularly-gridded function class (RGF). The RGF class is a pure C++ class for operating with traditional SEPlib (also referred as SEPlib77) seismic datasets (n-dimensional hypercubes), which are regularly gridded functions in a mathematical sense. In a companion paper in this report, Schroeder and Schwab (1996) describe the implementation and use of the RGF class. The RGF class is derived from the base class HCL_Vector. The HCL_Vector class defines the interface for all operations that can be expected from vectors, such as adding, scaling, subtracting, inner product, etc. The RGF class implements the details of the vector operations on SEPlib77 data structures. The presence of a common interface guarantees the interaction between solvers and any instance of the RGF class, or of any other class derived from the HCL_Vector class.

A second example of a vector class is the irregularly-gridded function (IGF90) class. Like the RGF class, the IGF90 class also derives from the HCL_Vector class. The IGF90 class encapsulates the vectorial behavior of irregularly-gridded SEPlib90 data structures. The IGF90 class is a hybrid C++/Fortran90 interface between the Fortran90 internals of the SEPlib90 data structures (Clapp and Crawley, 1996b) and the C++ abstractions of a HCL_Vector derived class; simply put, the IGF90 class is a C++ interface to the Fortran90 code of SEPlib90 data structures. The IGF90 class was written by the authors as part of the GOON project (Claerbout and Biondi, 1996) to bridge the advantages of the C++ language such as code reusability and readability, use of off-the-shelf mathematical optimization libraries, and the efficiency of the Fortran90 language.

This paper is a tutorial about the IGF90 class and its usage. It assumes that the reader is familiar with SEPlib90 data structures (Biondi et al., 1996), and has a basic knowledge of object-oriented programming. The examples in this tutorial are used for solving an inversion algorithm. Finally, we describe the design of the C++/Fortran90 interface, we illustrate it with the interface to the SEPlib90 data structures, and show an example of how to create the interface for a Fortran 90 operator.

THE IGF90 CLASS

The IGF90 class encapsulates the SEPlib90 data structure and its algebraic operations within a C++ framework. The C++ framework provides an interface to the HCL library, and thus allows the use of the HCL solvers on SEPlib90 data structures. The IGF90 class defines the vector operations of a SEPlib90 data set (its member functions), and has a vector space (its data member), an IGF90Space, which is an opaque handle (a pointer) to the Fortran90 structure that defines the header and grid information of the irregularly-sampled data set. The header information contains the irregular geometry and the grid information contains the mapping information between a specified grid and the geometry.

Nearly all the member functions of the IGF90 class operate on this opaque handle. Those member functions in turn call appropriate Fortran90 routines to do the actual work. The Fortran90 data structure is not accessed within C++; rather, the handle is merely passed around in the C++ world, while the actual computation occurs in the alternate Fortran90 universe.

AN IGF90 TUTORIAL

This section introduces the use of the IGF90 class by means of some simple examples. The tutorial consists of different blocks, which become increasingly complex as we go along. We start by showing how to construct IGF90 objects and operators, and end by solving a missing data problem:

- Constructing IGF90 objects
- Using basic member functions of the IGF90 class
- Applying a linear operator to IGF90 objects
- Applying a chain of operators to IGF90 objects
- Applying a block operator to composite IGF90 objects
- Solving a block linear problem using IGF90 objects and operators

Constructing IGF90 objects

The first thing we need to know about an IGF90 object is how to create one. A class object is initialized by the initialization of its data members. C++ supports an automatic mechanism for the initialization of class objects. A special class member function, called a *constructor*, is invoked implicitely by the compiler whenever a class object is defined. Constructor functions are characterized by having the same name as of its class. There are several syntactical rules by which an IGF90 constructor can be called:

- Given the number of dimensions and the number of samples along each dimension,

```
#include "IGF90.h"
...

const int N = 7;            // number of dimensions
int numSamples[N];

for (int i=0; i < N; i++)
    numSamples[i] = i+1;    // num. of samples along dim. i

// Construct by giving dimensions
IGF90 data1( N, numSamples);
```

- Given the number of dimensions and a corresponding set of axes,

```
const int N = 3 ;           // number of dimensions
AxisArray axes(N);

Axis firstAxis( 0.000 , 0.004,  "Time", "seconds" , 1024 );
Axis secondAxis( -2.0 , 0.050,  "Offset", "km" , 256 );
Axis thirdAxis( 1 , 1,  "CDP-X", "meter" , 1024 );

axes[0] = firstAxis; axes[1] = secondAxis; axes[2] = thirdAxis;

// construct by giving dimensions and a list of Axes
IGF90 data2( N , axes );
```

  Each Axis object is constructed by specifying its origin, increment, label, physical unit and its number of samples.

- Given a seplib tag (file name) to read SEPlib90 data structure into buffer; tag is "in", or an auxiliary input tag,

```
// look for the tag "name=" on the command line
IGF90 data3("name");
```

- Given another IGF90 object,

```
// Constructs a copy of the first object
IGF90 data4(data3);
```

- Given an IGF90Space object,

```
// Constructs a copy of the first object
IGF90 data5( data3.Space() );
```

Note that in the first two examples, `data1` and `data2` are SEPlib90 data structures with a regular geometry. In the last example, the sintax `data3.Space()` returns a reference to the vector space of `data3`. `Space()` is a member function of the IGF90 class. The IGF90 `data5` is then constructed with the same geometry as `data3`.

For many programs, all objects are allocated in the heap and referenced through pointers. The **new** operator allocates new dynamic memory. The **delete** operator takes a pointer to an object and reclaims its memory. Because of memory constraints, we will typically want to allocate IGF90 objects dynamically, and deallocate them as soon as we have finished using them.

```
IGF90 * data5;     // A pointer to an IGF90 object.

data5 = new IGF90 ("input");  // allocate dynamically and invoke
              // the constructor

...
delete data5;      // invoke the destructor and reclaim the mem
```

Using basic member functions of the IGF90 class

Once an IGF90 object has being instantiated, we are next interested in knowing how to manipulate it, and what its interface to the user is. The IGF90 class provides a set of linear algebra utilities that can be used on any IGF90 object, such as adding, multiplying, and scaling IGF90's, as well as I/O and on-screen inspection tools. These set of member functions are inherited from the abstract parent class HCL_Vector, but are implemented by the IGF90 class. They are written in Fortran 90.

```
#include "IGF90.h"
...

IGF90 data1("name");

// outputs some diagnostics about data1 to stderr
data1.Inspect(cerr);

IGF90 data2(data1), data3(data1);

data1.ScaleAdd(2., data2);        // data1 = 2. * data1 + data2

data1.Neg();                      // data1 = -data1

data1.Mul(2.);                    // data1 = 2. * data1
```

```
data1.Mul(2., data2);            // data1 = 2. * data2

data1.Write("result");           // writes to file with tag "result"

data1.Add(data2);                // data1 += data2
data1.Add(data3, data2);         // data1 = data3 + data2

data1.Sub(data2);                // data1 -= data2
data1.Sub(data3, data2);         // data1 = data3 - data2

data1.AddScale(2., data2);       // data1 += 2. * data2
data1.AddScale(2., data2, data3); // data1 = data2 + 2. * data3

data1.Fill(777.7);               // data1 = 777.7
data1.Zero();                    // data1 = 0.
data1.Random();                  // data1 = random numbers [0,1]

cerr << data1.Inner(data2) << endl;   // inner product of data1 by data2

cerr << data1.Norm()       << endl;   // norm of data1
cerr << data1.Norm2()      << endl;   // squared norm of data1
```

Applying a linear operator to IGF90 objects

This section illustrates how to use the C++ interface to apply an operator to an IGF90 data object. Operators, like the IGF90 class, are also derived from a class in the HCL library. This ensures the correct interaction between operators and data objects. The implementation of the convolution operator is encapsulated in the IGF90_ConvOp class. The subscript IGF90_ reminds us that this operator is to be applied to IGF90 objects. The IGF90_ConvOp class is derived from the HCL_LinearOpAdj (Linear Operators with an Adjoint) class. The details of how to build the interface of an operator will be presented in a later section. The example below shows the invocation of the forward transform of a convolution operator:

```
#include "iostream.h"
#include "IGF90.h"
#include "IGF90_ConvOp.h"
  ...

// Create model vector and a pointer to its vector space
IGF90 x( "model" );                          // read from disk
IGF90Space *domain_space = &( x.Space() );  // pointer to header info
IGF90Space *range_space = domain_space;

// Create output vector
IGF90 y( *range_space );
y.Random();
```

```
// Create the convolution operator
float eps; if(!fetch("eps","f",&eps)) eps=1. ;
IGF90 filter("filt");
// scale filter
filter.Mul(eps);
IGF90_ConvOp C( domain_space , range_space, filter ) ;

// Try out operator
C.Image( x, y );                            // y = C x
y.Write();                                  // output result
```

The syntax `IGF90_ConvOp C(domain_space, range_space, filter)` constructs the convolution operator `C`, given a pointer to the domain and range vector spaces and the filter coefficients. The operator `C` is an instance (an object) of the class IGF90_ConvOp. It can only be applied to IGF90 objects that have the same vector space as domain_space, and the output can only be saved in an IGF90 object that has the same vector space as range_space. This ensures that the geometry and the grid information of the input and output data sets correspond to what the operator expects. In the special case of the convolution operator, the input and output data sets have regular geometry.

The syntax `C.Image( x, y )` applies the forward mapping of the convolution operator `C` to the input `x` and outputs the result in `y`. Before applying the mapping, the `Image()` member function checks that the input and output IGF90 objects being passed agree with the domain and range spaces being handed over to the `C` operator at construction time.

If the adjoint is to be performed, the `Adjoint()` member function should be called on the `C` operator before the `Image()` member function is called. Note that the `Image()` member function expects as first argument the input and as second argument the output, independent of the operator being the forward or adjoint mapping. The following lines of code added to the above example call the forward and adjoint operations of the operator. The example implements a **dot product test** on the convolution operator:

```
IGF90 xp( *domain_space );    IGF90 yp( *range_space );

y.Random();     x.Random();

// Applied the Forward mapping
C.Image( x, yp );                            // yp = C x
// Applied the Adjoint mapping
C.Adjoint().Image( y, xp );                  // xp = C' y

// Dot product for convolution operator
cerr << "  " << yp.Inner(y)<< " " << xp.Inner(x) << endl;
```

Trying out another operator consists of changing a few lines of code. Assuming we keep the IGF90 model vector used in the previous example, `x`, we can write a new routine using a linear interpolation operator:

```
#include "iostream.h"
#include "IGF90.h"
#include "IGF90_ConvOp.h"
  ...

IGF90 data( "data" );
IGF90Space *Dspace = &( (IGF90Space&) data.Space() );

// Create linear interpolation operator,
// interpolate along header keys "sx" and "sy".
IGF90_LinInterpOp L( domain_space, Dspace, "sx", "sy" );
L.Adjoint().Image( data, x );                // x = L' data
x.Write();
```

The implementation of the linear interpolation operator is embedded within the IGF90_LinInterpOp class, also derived from HCL_LinearOpAdj. The constructor requires passing the domain and range in which the object will operate. The other two arguments indicate along which headers the interpolation will take place.

We can use the above example to interpolate the Seabeam data set (Claerbout, 1994). The data set is a collection of depth measurements at miscellaneous locations in the (x, y) plane, which are represented in a SEPlib90 structure as a set of one-sample-long traces. We use the IGF90 `data` object to hold this irregular data set. The left panel of Figure 1 shows the result of calling the adjoint interpolation operator on the data set. The gridding size is determined by the model vector `x`. The right panel in Figure 1 shows the result of convolving the interpolated Seabeam data set with a Laplacian filter. In the next subsection, we will see how to use the HCL library to chain these two operators into one.

The dot product test for the linear interpolation operator is simply coded as

```
IGF90 dp( *Dspace );      IGF90 xp( *domain_space );

data.Random();    x.Random();

// Applied the Forward mapping
L.Image( x, dp );                            // dp = L x
// Applied the Adjoint mapping
L.Adjoint().Image( data, xp );               // xp = L' data

// Dot product for linear interpolation operator
cerr << "  " << dp.Inner(data)<< " " << xp.Inner(x) << endl;
```

Applying a chain of operators to IGF90 objects

Using the HCL library, we can implement a chain of operators. There is no preset maximum number of operators that we can chain into one. The chain operator knows the order in which it must apply the operators that compose it, by the order in which the operators are

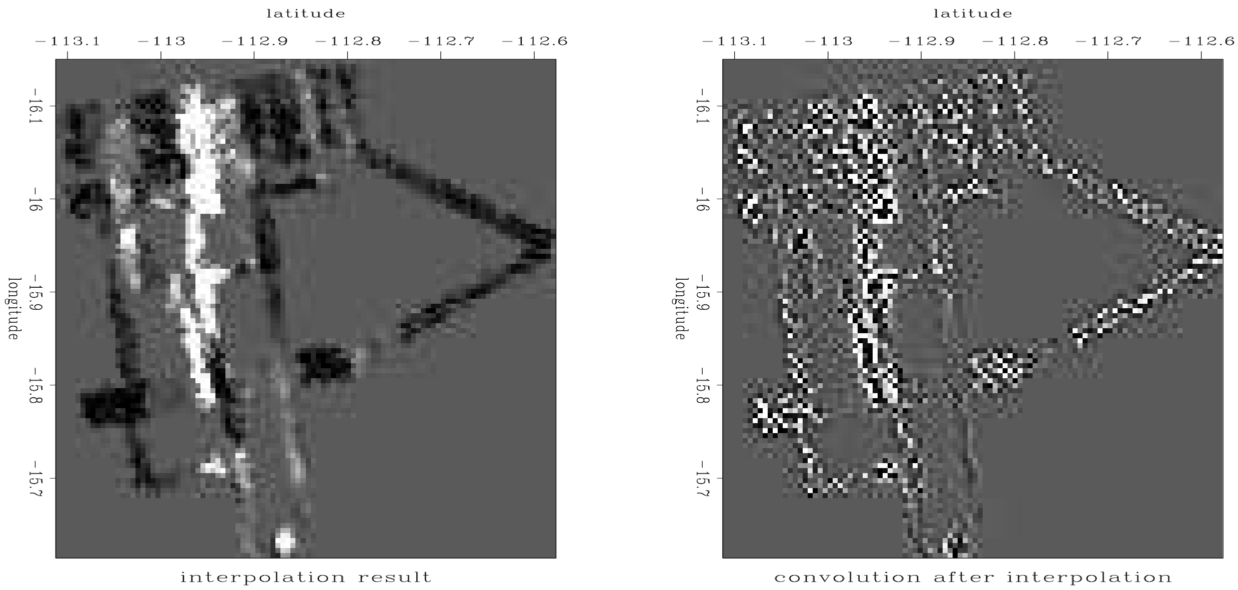

Figure 1: The left panel shows the Seabeam dataset after being interpolated into a 100×100 mesh. In the right panel, the interpolation result is convolved with a Laplacian filter. The Seabeam dataset has been decimated to 1/100th the original dataset. $\boxed{\text{hector1-anis}}$ [CR]

handed to the chain at construction time. The chain operator is encapsulated inside the HCL_CompLinearOpAdj class, or a class of Compound Linear Operators with Adjoint. This class handles the chain of operators by keeping an array of abstract HCL_LinearOpAdj's. HCL_LinearOpAdj is the *abstract base class* from which any operator with Adjoint in the HCL library is derived. It can be considered as an empty class that attempts to define the set of member functions that are common to all linear operators that have an Adjoint.

Objects from the IGF90_ConvOp and IGF90_LinInterpOp classes can then be handled abstractly in the chain operator as HCL_LinearOpAdj's. The chain operator itself can also be chained with another operator into a new chain operator. The following lines of code implement a basic chain operator that is applied to an IGF90 object:

```
#include <HCL_CompLinearOp.h>
  ...

// Chain Linear interpolation and convolution
// Make a chained operator out of C and L
HCL_CompLinearOpAdj chain(&C, 0, &L, 0) ;

// x = C'(L'(data))
chain.Adjoint().Image( data, x );
```

where `data` and `x` are IGF90 objects as defined in the previous examples. Figure 2 shows the result of applying the chain operator to the Seabeam data set. The result, as expected, is the same as that obtained in Figure 1.

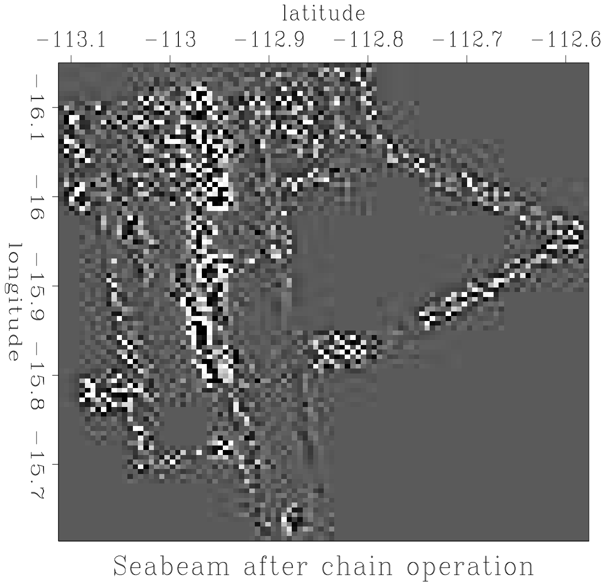

Figure 2: Chained interpolation and convolution operations on the Seabeam dataset. As expected, the result is the same as in the previous example, after application of the convolution to the interpolated data. hector1-chain [CR]

More operators can be added to the bottom of the chain operator (they will be applied last) using the `SetNext()` member function:

```
  HCL_CompLinearOp chain2(&C, 0, &L, 0) ;
  chain.SetNext(&chain2.Adjoint(), 0);     // chain = L' C' L C
```

Applying a block operator to composite IGF90 objects

Blocks of IGF90 operators can also be easily implemented within the HCL mainframe. The following example implements a basic block operator that gets applied to a composite IGF90 object:

```
#include "HCL_BlockOp.h"
#include "HCL_ProductVector.h"
char HCL_ProductVectorSpace::Type;
  ...

  // Create data vector and its space
  IGF90 data( "data" );
  IGF90Space *Dspace = &( (IGF90Space&) data.Space() );

  // Build axes
  Axis a0( 0. , 1.,  "trace", "number" , 1 );
  Axis a1( -112.85 , 0.001,  "sx", "meters" , 50 );
  Axis a2( -16.012501 , .00025,  "sy", "meters" , 150 );

  const int NN=3; AxisArray aa(NN);
  aa[0] = a0; aa[1] = a1; aa[2] = a2;
```

```
IGF90 y( NN, aa );
y.Random();   // model some noise

// Create model vector and its space
IGF90 x( NN, aa );
IGF90Space *Sspace = &( (IGF90Space&) x.Space() );
x.Zero();

// Create a 2x1 left-hand-side (lhs) block vector
HCL_ProductVector lhs(&data,0,  &y,0);
HCL_ProductVectorSpace *Lhsspace = &((HCL_ProductVectorSpace &) lhs.Space());

// Create the 2x1 block operator
HCL_BlockLinearOpAdj N( Sspace,  Lhsspace );
N.SetNext( &L, 0 );                  //      N = | L |
N.SetNext( &C, 0 );                  //          | C |

//                                             |data|
// Try the adjoint block operator  x =  |L'  C'| |    |
//                                             | y  |
N.Adjoint().Image( lhs, x );
x.Write();
```

The `lhs` vector of the HCL_ProductVector class implements a 2×1 matrix of HCL_vectors. The `N` object of the HCL_BlockLinearOpAdj class implements a 2×1 block operator. The shape of the domain and the range vector spaces determines the shape of `N` operator. The `SetNext()` member function fills the element of the block operator by rows, from top to bottom.

Adding the next couple of lines to the previous example implements the dot product test for the block operator:

```
IGF90 xp( *Sspace );  IGF90 dp ( *Dspace );
IGF90 yp( *Sspace );

HCL_ProductVector lhsp(&dp,0,  &yp,0);

data.Random(); y.Random(); x.Random();
lhsp.Zero();   xp.Zero();

// Applied the Forward mapping
N.Image( x, lhsp );
// Applied the Adjoint mapping
N.Adjoint().Image( lhs, xp );

// Dot product for block operator
cerr << "  " << lhsp.Inner(lhs)<< " " << xp.Inner(x) << endl;
```

Figure 3 shows the result of the block operation. The block operator interpolates the Seabeam data and adds to it some background filter noise. The filter used is a Laplacian filter.

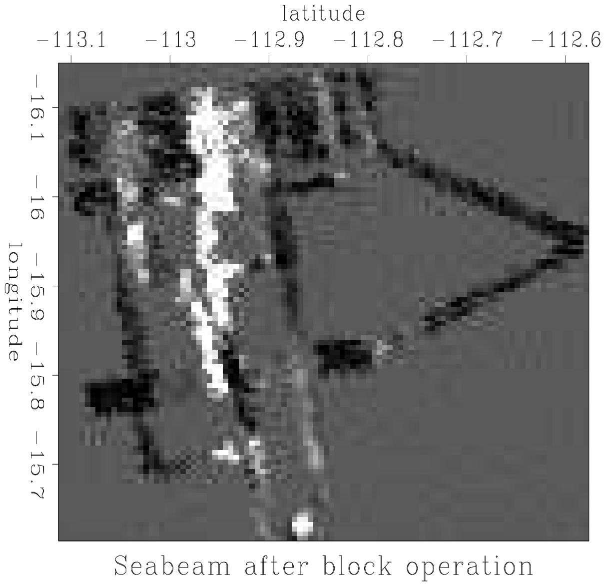

Figure 3: Row operation of interpolation of Seabeam dataset and convolution of random noise. hector1-block [CR]

Solving a block linear problem using IGF90 objects and operators

In this example, we solve the 2-D inverse linear interpolation problem: given some data values sprinkled on the (x, y) plane, we need to find a function on a uniform mesh from which we can extract the data by linear interpolation. The solution is constrained to minimize the power of a given filter applied to it. The regression

$$\begin{bmatrix} \mathbf{d} \\ \mathbf{0} \end{bmatrix} \approx \begin{bmatrix} \mathbf{L} \\ \mathbf{C} \end{bmatrix} \mathbf{x} \tag{1}$$

expresses the problem at hand. $\mathbf{d}$ is the data we want to extract by linear interpolation $\mathbf{L}$ from a model $\mathbf{x}$, constrained so that the application of a filter with an operator $\mathbf{C}$ has minimum power.

The following lines of code show the creation and invocation of a conjugate gradient solver `CGstep solver`, which iterates the regression expressed in equation (1):

```
#include "CGstepSolver.h"
...

// Create the CG solver
float tol;  if(!fetch("Tol","f",&tol))  tol = 1.0e-2;
int niter; if(!fetch("niter","d",&niter))  niter = 50;

CGstep * solver = new CGstep( tol, niter );
```

```
// set debug parameter for solver
solver->Parameters().PutValue("Trace", 1);

// Run the CG solver
solver->Solve( N , lhs, x );

// Output result
x.Write();
```

The code consists of three concise steps: first, the block operator `N`, the input `x` and the output `lhs` are constructed (not shown); secondly, the `solver` is constructed; finally, the input data is inverted using the solver. The syntax `solver->Parameters() .PutValue("Trace", 1)` sets a flag for the solver to output the convergence rate to stdout. Figure 4 displays the result obtained after 70 iterations using a Laplacian filter.

The CGstep class is derived from the HCL_LinearSolver abstract base class. The HCL_LinearSolver class has only one member function, `Solve()`. This function takes as its arguments a linear operator, a vector that is the left hand side of the regression in equation (1), and a vector that is the solution to the regression. The CGstep class implements the `Solve()` member function as a conjugate gradient algorithm.

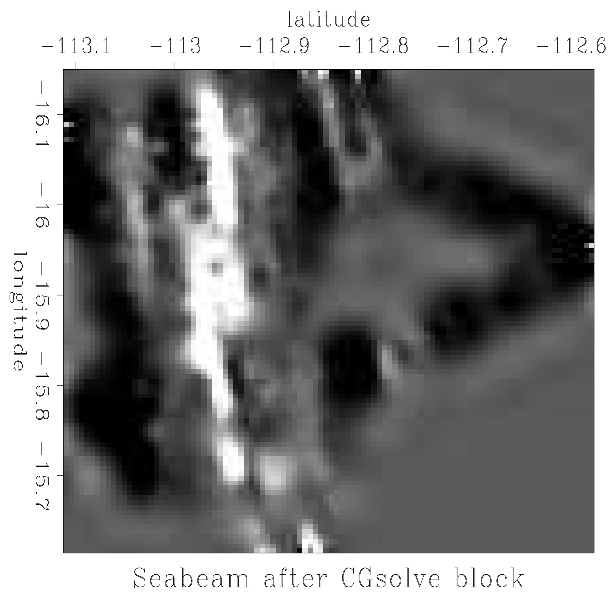

Figure 4: Solution obtained to the seabeam missing data problem after 70 iterations, using a Laplacian filter. hector1-cgslvbkop [CR]

BINDING C++ WITH FORTRAN90

C++ and Fortran90 are both modern standard languages that allow us to solve numerical problems in a flexible way. However, depending on the type of problem that is being solved, neither language is the perfect choice. If we are not contending with the deficiencies of either language, we have the option of combining both languages and living with the best of both worlds. Currently, there is no standard procedure for linking both languages together, which is a potential pitfall, but there is a simple way of creating Fortran90 objects within a C++ frame work.

Why

Object-oriented programming offers many benefits to researchers constantly trying new ideas and implementations. However, the mainstay of scientific and engineering programming is Fortran. The reasons for this are more than cultural. The restrictions in the Fortran type-system prohibit the use of pointers beyond the simple alias mechanisms allowed by Fortran90. Consequently, Fortran compilers can perform data-dependence-based optimizations which are not possible in C++. In addition, code generation technology and optimized intrinsic library functions present in Fortran are much better than their C++ counterparts. With the present generation of Fortran90 compilers, vector and array operations are easy to compile into efficient code.

When testing concepts that act on large data sets, one is confronted with the problem of needing results in a limited time to test one's ideas. Thus efficiency becomes another important criterion. Parallel computers greatly reduce turn-around time for many computational problems. However, there is no standard portable programming language which allows such an object-oriented parallel portable implementation. We believe that it is essential to support a portable programming environment and an object-oriented framework that mixes Fortran90 and C++ across the entire range of supercomputer platforms, if we are to tackle large numerical problems.

Depending on what field of activity one is associated with, there are different points that vary in importance. For teaching purposes, *readability and exposure of ideas* are most important, while for research purposes, easy *translation of mathematical concepts* to numerical implementation dominates. For real work, the pressing issue is *efficiency* and speed to solve a real world problem. In maintaining a large program base it is necessary to have the flexibility to add new concepts with ease. In situations where one is involved in cooperative research projects, reusable portable code libraries become the important asset. The ability to assimilate and distribute them transparently is of great importance.

None of the current computer languages fulfills these goals per se. Some of the computer programming languages, however, are pretty close to what one would like: C++ and Fortran90. Let us quickly characterize a programming world by using the following fragments together:

- C++: abstract programming style, strong typing, multiple instances, flexibility.
- Fortran90: easy to use/learn for Fortran programmers, data parallel language, most compilers parallelize.

However, such a world is not a "standard" world: there is no programming standard (given by compiler writers or standard committees) for linking both languages together. So that one could potentially be at the mercy of the compiler writers.

How

This calamity can be overcome, and a usable, flexible set of research tools can be obtained, if we

- use C++ as the main framework and infrastructure
- create and operate on Fortran90 objects within Fortran90 subroutines/functions
- pass within C++ abstract pointers to Fortran90 objects (data structures and methods).

This is the short recipe for combining both languages. So far it has been successfully tested (in parallel) on an SGI PowerChallenge and on an IBM SP-2 without any change to the source code. We will now present two C++ interfaces, the first for a Fortran90 data structure and the second for a Fortran90 operator.

C++ interface to a Fortran90 data structure

Let us consider a C++/Fortran90 interface for the SEPlib90 data structure. The IGF90Space class provides an encapsulation of the SEPlib90 header information. The following lines of code show part of the IGF90Space class definition:

```
typedef void* HANDLE;
#define OPAQUE_SIZE 1024;

extern "C" sep90_reed_(const char * , int *, HANDLE);
class IGF90Space: public HCL_VectorSpace {
   protected:
      HANDLE opaque_f90_handle;   // pointer to Fortran90 data structure;

   public:
      IGF90Space();
      IGF90Space(char *tag);
      ~IGF90Space();

      ...    // some other constructors and virtual member functions

      HANDLE GetOpaque();
};

IGF90Space::IGF90Space(){
   opaque_f90_handle = new char[OPAQUE_SIZE];
}

IGF90Space::IGF90Space(char *tag){
   opaque_f90_handle = new char[OPAQUE_SIZE];
   int length = strlen(tag);
   sep90_reed_( tag, &length , opaque_f90_handle );
}

IGF90Space::~IGF90Space(){
```

```
   delete opaque_f90_handle;
}

HANDLE IGF90Space::GetOpaque() const {
   return(opaque_f90_handle);
}
```

The IGF90Space class is composed of an `opaque_f90_handle` pointer to the Fortran90 data structure. The `opaque_f90_handle` pointer is used to pass the Fortran90 data structure in the C++ world. The constructor `IGFSpace90(char *tag)` initializes an IGF90Space object by reading the header information of the SEPlib90 data structure from disk, which is done by the `sep90_reed` Fortran90 subroutine. Because of class encapsulation, access to the `opaque_f90_handle` pointer is obtained only through the `GetOpaque()` member function.

The Fortran90 subroutine `sep90_reed` provides a wrapping layer to the `sep_reed _mod` module. The Fortran90 `sep_reed_mod` module is described in this report by Clapp and Crawley (1996). Some Fortran90 compilers use the "$" sign to name subroutines defined within modules. The direct call of Fortran90 subroutines that are within a module will require the inclusion of the "$" sign in the C++ extern call. Some compilers will only allow the use of the "$" sign if one uses special compiler flags. We opted not to use these flags, but used another layer of Fortran90 wrapping in order to be completely portable. The next lines of code show the wrapper file for the Fortran90 `sep_reed_mod` module:

```
subroutine sep90_reed(tag,length,sepdata)
use sep_reed_mod

integer length
character*(length) tag
type (sep_90), pointer     ::sepdata

character(len=length)      :: newtag
newtag = tag

call init_vars(sepdata)
!read in only the headers
call sep_reed(newtag,sepdata,header_only=.true.)

return
end subroutine
```

C++ interface to a Fortran90 operator

To operate on the Fortran90 data structures we must write our operators in Fortran90 and provide them with a C++ encapsulation. As an example of a C++/Fortran90 interface, consider the public definitions of the IGF90_LinInterpOp class header:

```
#include "HCL_LinearOp.h"
```

```
class IGF90_LinInterpOp: public HCL_LinearOp {
   public:

      IGF90_LinInterpOp( IGF90Space *dom, IGF90Space *ran,
                                          int ky1=1, int ky2=2 );
      ~IGF90_LinInterpOp();

      HCL_VectorSpace & Domain() const;
      HCL_VectorSpace & Range()  const;
      virtual void Image( const HCL_Vector & xx, HCL_Vector & yy ) const;

      ...          // I/O
};
```

An instance of this class is constructed from a domain and a range vector spaces, and from two header keys, along which the linear interpolation will take place. The member functions `Domain()` and `Range()` return a pointer to the operator's domain and range vector spaces respectively. The `Image()` member function is a virtual function inherited from the base class HCL_LinearOp. It implements the forward mapping of the operator, by calling a Fortran90 subroutine, as shown below:

```
#include "IGF90_LinInterpOp.h"

extern "C" f90_linear_interp2d_(int &adjf, HANDLE, HANDLE,
                                const int &key1, const int &key2) ;
extern "C" int sep90_simple_check_(HANDLE, HANDLE);

void IGF90_LinInterpOp::Image( const HCL_Vector & xx,HCL_Vector & yy ) const
{
   if (sep90_simple_check_( ((IGF90 &) xx).GetOpaque(),
                            ((IGF90Space &) Domain()).GetOpaque() ) != 1 ) {
      cerr << "ERROR: LinInterpOp: IGF90 vector is not in domain" <<endl;
      exit(1);
   }
   IGF90 & x = (IGF90&) xx;

   if (sep90_simple_check_( ((IGF90 &) yy).GetOpaque(),
                            ((IGF90Space &) Range()).GetOpaque() ) != 1 ) {
      cerr << "ERROR: LinInterpOp: IGF90 vector is not in range" <<endl;
      exit(1);
   }
   IGF90 & y = (IGF90&) yy;

   int adjf =0;

   f90_linear_interp2d_( adjf, x.GetOpaque(), y.GetOpaque(), key1, key2 );
}
```

The `Image()` member function checks the vector spaces of the input and output vectors being passed versus the domain and range vector spaces given at construction time. The member function `GetOpaque()` returns the `opaque_f90_handle` pointers to the Fortran90 data structures. The Fortran90 subroutine `sep90_simple_check` does the space comparisons. Finally, we provide a Fortran90 wrapper, `f90_linear_interp2d`, that makes the appropiate calls to the the Fortran90 `linear_interp2d_mod` module. (Clapp and Crawley, 1996b):

```
subroutine f90_linear_interp2d(adj,model,data,key1_in,key2_in)
use linear_interp2d_mod
integer                  :: adj
type(sep_90),pointer     :: model, data
integer                  :: key1_in, key2_in

call interp2dinitn(model,data,key1_in, key2_in)
call linear_interp2d(adj==1, model, data)
return
end subroutine
```

IMPLEMENTATION PROBLEMS: CAVEATS

Dynamic allocation in Fortran90 of a size larger than the core memory can cause the machine to go berserk. It seems that the runtime system is confused about who is allocating memory.

Some compilers allow the use of the "$" sign in variable names if you use special compiler flags. That is useful if one wants to access Fortran90 routines inside modules. We opted not to use this flag, but used another layer of Fortran90 wrapping, in order to be completely portable.

At a given time we had 7 people working on the C++/Fortran90 building blocks—too many cooks stirring the pot. The whole system was in a continuing state of flux, which made it hard to pinpoint problems. Specially, as some problems got fixed, others were introduced. Our recommendation for the future: do not develop Fortran90 code while writing C++ wrappers for it. Many steps had to be done over and over again. Moreover, always require the Fortran90 operators to pass the dot product test.

CONCLUSIONS

The basic IGF90 building blocks work fine: constructors, member functions, linear operators, chained operators, block operators, and linear solvers. We have shown working examples of the IGF90 class and its interface to the HCL library of optimization algorithms. We have developed an interface to encapsulate Fortran90 data structures and operators within C++. We have discussed here the design of the C++/Fortran90 interface, as well as showed examples of the C++ interface to SEPlib90 data structure and the interface to a Fortran90 linear interpolation operator.

ACKNOWLEDGMENTS

We would like to thank Bob Clapp & Sean Crawley for providing the Fortran90 linear interpolation and convolution operators and Dave Nichols for testing the opaque_f90_handle.

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.

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

Schwab, M., and Urdaneta, H., 1996, Hilbert Class Library: Ideas behind the design: SEP–**93**, 253–262.