Stanford Exploration Project, Report 93, November 11, 1997, pages 253–262

Stanford Exploration Project, Report 93, November 11, 1997, pages 253–262

Short Note

Hilbert Class Library: Ideas behind the design

Matthias Schwab and Hector Urdaneta[1]

keywords: *linear, inversion, modeling, NMO, algorithm*

INTRODUCTION

The Hilbert Class Library (HCL) is a C++ library for applied, large-scale, numerical optimization. The Hilbert Class Library essentially defines a set of abstract classes which programmers use to derive their application specific data structures, operators, and solvers. The common set of base classes ensures the cooperation of these various software pieces. Consequently, experts write code exclusively in their field of expertise and can be confident that it will integrate with the software written by other programmers. The HCL library is accompanied by a growing number of examples and solvers.

HISTORY

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 in a more traditional environment. Simultaneously, Dave Nichols and Lester Dye (Petroleum Engineering at Stanford) 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 HCL library (Schwab, 1994). Mark Gockenbach is HCL's primary author (Gockenbach and

[1]**email:** matt@sep.Stanford.EDU, hector@sep.Stanford.EDU

Symes, 1996) and maintains a detailed web-page (1996).

Currently, the HCL user community is small but nevertheless difficult to list precisely. At the present, Bill Symes's Rice Inversion Program and SEP experiment with geophysical problems. Martin Karrenbach has promised to continue HCL and related GOON work at Karlsruhe University. Dave Nichols shows continued interest, although his direct participation may be limited by his professional obligations. Finally, Mark Gockenbach continues to develop and maintain HCL. He also applies HCL to non-geophysical problems, such as molecular energy minimization.

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. A solver hands a set of model parameters to an application's objective function for evaluation . The objective function returns a single cost value to the solver . While this simple data transfer succeeds in the separation of the solver and application code, it does not support solvers that take advantage of additional application specific information, such as the derivative of the objective function.

PROGRAMMING ALGEBRA

Object-oriented programmers usually have to struggle to design a flexible and general base class hierarchy. In contrast, HCL developers were handed a concise and abstract hierarchy on a silver platter: vector algebra. HCL implements abstract mathematical concepts, such as vectors, vector spaces, and operators. Every HCL class corresponds to a standard item in an algebra or numerical analysis textbook.

The designer of a new software package needs to accommodate present requirements and to anticipate the problems of the future. In object-oriented programming particularly, the design of your base classes can severely limit one's software's abilities later. By implementing abstract mathematical concepts such as abstract base classes, the HCL classes are guaranteed to be devoid of any application specific details, and are poised to embrace new developments in the field of numerical analysis. Because of its reliance on broad mathematical concepts, the class library supports flexibly even intricate interactions between applications and solvers in a consistent and expandable fashion (Schroeder and Schwab, 1996). Furthermore, HCL's underlying, well-known mathematical concepts should foster the communication and intuitive understanding among HCL users independent of their field of expertise. The ease of communication is especially important since HCL attempts to bridge the gap between applied mathematics and various applied sciences.

For practical reasons, HCL only includes mathematical entities that are needed by its current users to estimate solutions to applied inversion problems. As a design principle, HCL does not provide an all-encompassing framework, but is expected to grow as users require additional classes.

WHY OBJECT ORIENTATION?

More and more, researchers at SEP attempt to solve complex and large geophysical inversion problems. Consequently, to attack these problems efficiently, SEP needs sophisticated and reusable solvers. An object-oriented programming language enables us to hand an operator object (e.g. an NMO stacking routine) and a data object (e.g. a CMP gather) to a solver object (e.g. a conjugate gradient routine). The solver uses a defined set of abstract operations (methods) of the operator and the data to minimize the residual.

What are these abstract operations and why do we need object orientation to implement them? HCL 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 HCL 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 that the object-oriented implementation of the addition is carried by each object.[2] Object orientation allows us to wrap algorithms and data structures into functional units, such as vectors that add themselves. The solver does not need to know about the implementation details of the vector's addition method, but merely trusts that the standard `Add()` method invokes the correct computations resulting in an addition.

WHY C++?

A variety of languages (including C and Fortran90) support object-oriented programming in the sense that they allow to bundle variables and methods into objects. A full-blooded object-oriented language (such as C++ or Java) additionally supports the combination of encapsulation, inheritance, and abstraction.

Encapsulation refers to the separation of the internal implementation of an object and its publicly available methods, the object's interface. For example, independent of whether an array of numbers is internally stored in a binary tree or a standard array, a public `Add()` method of such a class will perform the necessary steps to compute the element-wise addition. In general, objects interact exclusively through their public interface.

Inheritance enables a class to share another class's member variables and methods. Normally. this leads to classes that are enhanced, specialized versions of the parent class. Two classes that are derived from the same parent class have identical names and invocations for all inherited methods and variables[3].

[2] Actually, all objects of a class share the class methods, while they have individual copies of the class's member variables.

[3] Programmers of object-oriented languages distinguish between an "is a" and a "has a" relationship: only if an object is a variation of the other object, should a programmer use inheritance. For

Abstraction complements encapsulation and inheritance. Abstraction permits the programmer to declare an object's public methods without actually implementing them.

Abstraction allows us to enforce at compile time that future concrete implementations of a class implement the abstract class's methods. The vector class's public `Add()` method, which I mentioned above, is guaranteed to exist in any class derived from vector. Consequently, any object can trust that vectors have a method `Add()`.

HCL essentially is a framework of such abstract classes. Programmers derive any application-specific class from a corresponding abstract HCL class and thereby inherit HCL's standard interface. The C++ compiler forces an application programmer to implement all the methods and variables that are inherited from an abstract base class. Finally, an object of the new class can interact with any other object of HCL through the inherited variables and methods. Only full-blooded object-oriented languages allow the inheritance of interfaces and enforce their consistency during compilation.

APPLICATIONS

Fundamentally, the HCL library was designed to find estimates for a vector `x` subject to

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

where $\mathbf{A}$ is a linear or nonlinear operator (a matrix is merely a special case), and $\mathbf{b}$ is a known vector. In general, HCL does not assume that $\mathbf{A}$ is square, a matrix, or full rank, that the data is consistent, over-determined, or under-determined. Furthermore, HCL does not assume that a solution is unique or that a solution even exists. HCL only requires the problem is of the form stated in equation (1). The specific HCL solver you choose may, however, impose further constraints (usually on your operator $\mathbf{A}$).

Inversion problems fundamentally revolve around three programmable objects: the vector that contains the known data $\mathbf{b}$, the unknown model $\mathbf{x}$, and the operator $\mathbf{A}$ that images a model vector to a data vector, and finally a solver that, given an operator, a data vector, (and possibly a starting model $\mathbf{x_0}$) will return an estimate for the best-fitting model $\mathbf{x}$.

Vector objects

A simple example of a vector is an array of real numbers. This example is especially interesting since most of the data we encounter in applied sciences can be represented

---

a "has a" relationship a programmer should make make an object of one class a member variable of the other. If in doubt, the authors tend to prefer a "has a" relationship in non-abstract classes.

as such an array with regard to a certain basis. However, to represent the vector completely, we need to store its array representation and a definition of its basis.

HCL requires two objects to represent a data element: a vector object and a vector space object. True to its mathematical definition, a HCL vector is able to add another (compatible) vector to itself, to scale itself by a given scalar, and, since it is in a Hilbert space, to compute the inner product between itself and another vector.

A vector also carries a link to a corresponding vector space object. In general, the vector object represents the data by storing it with respect to some basis. The corresponding vector space object carries a representation of this basis. The vector space object can compare two vector objects and decide if they are from the same space (and therefore can be added, for example). The vector space object can also spawn a new vector member.

Additionally, a programmer can combine simple vectors to build more complex ones. The ProductVector, $\mathbf{x} = (\mathbf{x_1}, \mathbf{x_2})$, object contains the Cartesian product of the vector $\mathbf{x_1}$ and $\mathbf{x_2}$. Since the ProductVector class is derived from the general vector class, ProductVectors can be used as vectors.

A classic seismic example of a vector is a SEP cube, a set of real values on a regular grid. In HCL, SEP implemented such a SEP cube as a Regular Grid Function(Schroeder and Schwab, 1996) which is derived from the abstract vector class. The function values are stored as a one-dimensional array of real numbers and are accessible through a simple indexing operation. The vector adds another vector by first testing the two corresponding vector spaces for compatibility, and then by adding the corresponding array elements. The vector space object contains a set of axis objects. Each axis contains its offset, its increment, its number of samples, and its unit. The vector space checks its compatibility with another vector space, by comparing their axes.

Operator objects

An operator object is a function that accepts an input vector and returns an output vector when its `Image()` function is invoked. An operator knows about its domain and range, which are vector space objects. Similar to vectors, complex operators can be built from simple ones. The *Compound Operator* combines two operators so that the output of the first operator is the input to the second. Furthermore, the *Block Operator* arranges operators into operator matrices. For example, a Bock Operator organizes four operators $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$ into an operator matrix

$$\begin{bmatrix} \mathbf{A} & \mathbf{B} \\ \mathbf{C} & \mathbf{D} \end{bmatrix}$$

In many cases, the programmer may be able to enhance the operator to compute additional information, such as the adjoint or the derivative of the operator. Since some solvers require such additional information, HCL standardizes the invocations for these additional functions by including additional derived operator classes, such as a *linear operator* class, or a *linear operator with adjoint* class.

A classic seismic example of an operator is the NMO process and its adjoint: The NMO operator is linear, and it is fairly easy to implement its adjoint operator. An NMO input vector is a CMP gather. A CMP gather's two axes are offset, h, and two-way traveltime, t. The midpoint coordinates of the CMP gather, and the description of the axes are part of the CMP vector space object. The operator's output vector is a stacked trace. Its axis samples time, τ. The NMO operator is best defined by its adjoint. The adjoint of the NMO operator spreads an impulse in the stacked trace along a hyperbola in the CMP gather.

Solver objects

A solver's `Solve()` method returns an estimate of the model $\mathbf{x}$ for a given operator $\mathbf{A}$ of compatible type[4], a data input vector $\mathbf{b}$, and possibly a starting model vector $\mathbf{x_0}$. Usually, a programmer sets the solver's convergence criteria. For example, geophysicists usually combine a maximal number of iterations (for iterative solvers) or an acceptable residual norm. Currently, HCL includes about three solver classes. We hope that as more solvers are implemented, a solver class hierarchy will develop.

CODE FORMAT

C++ code written in the HCL framework can be as clear and clean as traditional Ratfor code in the SEPlib framework. However, the two code formats are very distinct. The code of an operator's `Image()` method is comparable in both frameworks. Particularly, since HCL's vector accessor functions imitate the Fortran indexing. On the other hand, `for` loops prevail over `do` loops in C++ code. HCL programmers are tempted to implement recursive looping to accommodate data inputs of various dimensions in a single operator. Such recursive looping can clutter the operator's code and possibly slow down its execution. Because of its object orientation, HCL hides ugly I/O details in the various vector classes. Furthermore, the solver objects encapsulate the entire optimization code. Consequently, the main routines are very readable and expose the overall approach. Since the solvers exclusively use a limited set of standard vector and operator methods such as `myvector.Add()` or `myop.Image()`, the solvers are almost as readable as pseudo-code and expose, without clutter, interesting abstract concepts.

HCL's operator classes deal with constructing and checking of vector spaces (the operator's domain and range), which can be tedious. The traditional SEPlib code does not check vector space information.

---

[4]as I pointed out above some solvers accept only certain operators

Unfortunately, HCL's abstract base class concept requires a considerable amount of explicit type casts which inevitably clutter the HCL code. The casts are necessary to specify at compile time what objects are returned by a class that returns an abstract object. Without the cast the compiler will not know about any additional methods or members the concrete class carries.

C++ DIFFICULTIES

Unfortunately, HCL's suffers from some intrinsic C++ shortcomings. The language is difficult to learn and error-prone because of the lack of automatic garbage collection, and the addressing of objects through pointers. Potentially, the lack of multiple inheritance could limit our mathematical elegance. We feel, however, that these disadvantages are at least partially offset by C++'s powerful, object-oriented programming paradigm.

Garbage collection

HCL objects, such as vectors, consume computer resources, particularly memory. During the execution of a program, C++ burdens the programmer with freeing these resources. If a programmer does not free a certain resource, after it outlived its usefulness, the resource will not be available to him until the program exits. Since in an object-oriented language, the allocation of resources and the dependencies between objects can be hidden, garbage collection by the programmer is error prone and potentially wasteful.

The lack of garbage collection is a tremendous complication in the HCL framework. For example, several vectors may be members of the same vector space. If the vector space object consumes a significant amount of resources (for example the header values of an irregular data set), we may want all vectors to refer to a single vector space copy. The programmer has to ensure that no vector is referring to that vector space anymore before she can deallocate the vector space.

Multiple Inheritance

Were HCL to follow strictly mathematical relationships, some of HCL's abstract base classes would need to be derived simultaneously from several parent classes. However, multiple inheritance can pose serious compiler problems, especially when the parent classes have common ancestors themselves.

For example, in Figure 1 a *compound linear operator with adjoint* should be derived from the *compound linear operator* and from the *linear operator with adjoint*. Both parent classes are derived from the *linear operator*. However, the compound linear

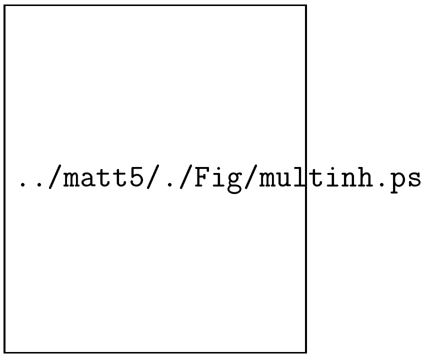

Figure 1: Multiple inheritance is mathematically desirable, but difficult to implement in C++.

operator with adjoint would inherit all linear operator members twice; a fact some compilers do not seem to appreciate.

In general, C++ practitioners advise against such multiple inheritance and HCL avoids them entirely. When does multiple inheritance matter? If we had a solver that inverts linear operators with adjoints and another one that inverts compound linear operators, a compound linear operator with adjoint should be an acceptable argument to both solvers. However, without multiple inheritance, the operator will be acceptable to only one of the solvers.

SUMMARY AND CONCLUSIONS

HCL translates the abstract mathematics of inversion into a set of expandable and efficient C++ classes. During the SEP conference on object-oriented programming (Claerbout and Biondi, 1996), Mark Gockenbach and Bill Symes convincingly demonstrated that the HCL library can be used in small to medium seized seismic research problems. Over the summer, SEP wrote concrete C++ classes that implement HCL's abstract base classes and that read and write traditional SEPlib data files. Martin Karrenbach (1996) found how to link Fortran90 with the HCL framework.

However, C++, and consequently HCL, are not easy to learn for researchers who traditionally write Fortran 77 programs. It is yet unclear if it is desirable or possible to hide the C++ framework from SEP researchers by having them write Fortran subroutines. Additionally, C++ compilers have not reached the maturity of C or Fortran compilers. We encountered several C++ compiler problems during this summer, and currently our code only executes on SEP's SGI Power Challenge[5]. Reliance on a combination of Fortran and C++ may severely limit the software's portability.

In conclusion, HCL seems to facilitate the cooperation SEP is seeking. Joel Schroeder (1996) demonstrated HCL's ability to reuse successfully a sophisticated solver written by a numerical analysis expert, Mark Gockenbach. The HCL class hierarchy proved elegant and flexible.

[5]We have not yet *seriously* attempted to port the HCL package.

We are, however, less fascinated by C++, in which HCL is implemented. C++ is at best a compromise between efficiency and object-oriented elegance. Finally, we are not convinced yet that a HCL framework that combines several computer languages, particularly C++ and Fortran 90, would be sufficiently portable. Portability is especially important if remote HCL users are to collaborate.

REFERENCES

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

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

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.

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

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

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

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

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