1

An Overview of HCL1.0

Mark S. Gockenbach[†] William W. Symes[‡]

October 26, 1999

Contents

1 Introduction

Many problems in the applied sciences involve optimization, either because a variational principle underlies the mathematical description, or because parameters in a mathematical model

*This work was partially supported by the National Science Foundation under grant DMS-9627355.
[†]Department of Mathematical Sciences, Michigan Technological University, 1400 Townsend Drive, Houghton, MI 49931-1295
[‡]Department of Computational and Applied Mathematics, Rice University, PO Box 1892, Houston, TX 77251-1892

are to be chosen in an optimal fashion. Among the many examples of such problems, we mention optimal design, minimal energy, optimal control, and parameter identification (inverse) problems.

Numerical solution of optimization problems is well-developed; several decades of work by numerical analysts have resulted in a collection of refined and effective algorithms. Many of these have been implemented in high-quality software packages, typically written in Fortran or sometimes C. Some of the better known examples are MINOS [17], MINPACK [15], and LANCELOT [2].

According to anecdotal evidence collected by the authors, these software packages do not have the impact on scientific computing that one might expect. In fact, we are aware of a significant number of applications projects (including our own) in which the investigators felt obliged to write their own optimization code, despite the availability of well-tested packages of demonstrably greater sophistication and effectiveness.

The principal reason for avoiding "off-the-shelf" software is the difficulty of adapting application-specific code (such as simulators) to the interface required by the optimization software. This difficulty is caused by a mismatch between data structures demanded by the simulator, on the one hand, and the optimization software, on the other. Because of the constraints of procedural computer languages, the optimization software is invariably coordinate-based; vectors are stored in one-dimensional arrays, inner products are written as explicit loops, and so on. On the other hand, one-dimensional (in-core) arrays may not offer an appropriate (or even feasible) data representation for the application.

For example, seismic data processing involves 3- or 4-dimensional data sets far too large to be stored in one-dimensional in-core arrays. These data sets are stored on disk; when it is necessary, for example, to compare two data sets by subtracting them, successive "slices" of the data are read from disk, subtracted, and written to disk. This example also illustrates another difficulty with procedural programming. The standard data structures developed for seismic data sets include various parameters describing the data (grid parameters, details of acquisition geometry, etc.). These parameters are not accommodated easily in a standard one-dimensional array; there is simply no natural place to put them.

While Fortran optimization packages are tied to specific data representations, the algorithms themselves are almost always intrinsically coordinate-free. Krylov space methods and least change secant ("quasi-Newton") algorithms use only the basic Hilbert space operations, without explicit reference to a particular basis. *Software packages for optimization, implemented in procedural programming languages such as Fortran, typically do not—and cannot easily—express this representation-free nature of the underlying algorithms.* In our opinion this accounts for the failure of many scientific computing projects to use packaged optimization software.

Motivated by our own experience with seismic inverse problems, we have designed a library of C++ classes that allow optimization and iterative linear algebra algorithms to be coded in such a way that they can be used to solved problems of arbitrary complexity. We do this by using the object-oriented programming paradigm; that is, the objects manipulated by our computer codes are defined by their properties, not by their representations. Because the natural mathematical setting for the algorithms we have in mind is Hilbert space, we have named the collection of these classes the *Hilbert Class Library (HCL)*.

This report is a revision of [11], updating that reference for version 1.0 of HCL.

1.1 Object-oriented programming and C++

The object-oriented programming paradigm includes two principles of software design: a computer program should be organized around the fundamental objects it manipulates, and these objects are defined by their properties, rather than by their implementation. Taken together, these principles allow for a degree of abstraction that is not easily obtained in a procedural programming style.

In optimization code, it is easy to identify the important objects; they are vectors, operators, real-valued functions, and so forth. Moreover, their properties are, for the most part, specified by the mathematical definitions. In C++, abstraction is accomplished in the following way: a *base class* is defined to represent an object, such as a vector. The base class defines the properties that are common to such objects, but it does not implement any particular instance. *Derived classes* are then defined for each desired instance of the base class; for the example of vectors, derived classes may include Euclidean vectors (implemented using one-dimensional arrays), functions sampled on a regular grid (implemented using multi-dimensional arrays), out-of-core vectors (implemented using disk files), and so forth.

Each derived class must implement the properties defined by the base class; for example, a derived vector class must provide code to add two vectors. A derived class may also implement additional properties; commonly, these include access functions. Any code that is written to apply to objects from the base class may use the properties defined in the base class, while only code written for the derived class can use the additional properties defined for it.

The effect of this is that optimization code does not need to know about the data structures used by the application code, because this information is hidden in the classes. As we show below, a minimization routine can be written that will apply equally well to vectors stored in one-dimensional arrays and to vectors stored on disk in some standard (say seismic) format.

The ability to express abstraction allows the union of generic and application-specific code. The generic code, such as optimization routines, is coded in a coordinate-free manner; this enables a very high-level, easily understood style of coding (such as calling a function named "Add" to add two vectors). The application-specific code uses derived classes designed for the specific application; this allows the user maximum flexibility.

1.2 The use of C++ in scientific computing

Use of C++ in scientific computing, while by no means widespread, has increased over the last several years. For example, the designers of LAPACK [1] have provided a C++ interface (see [6]). The interface is based on a collection of vector and matrix classes; these classes have evolved over the past few years through several related packages: Sparselib++ [4], IML++ [5], and MV++ [20] due to Pozo and his collaborators. Currently a package called Template Numerical Toolkit (TNT) [21] is being designed as a successor to these packages.

The work culminating in TNT is very closely related to ours in motivation. Quoting from [4]:

> ... code involving sparse matrices tends to be very complicated, and not easily portable, because the details of the underlying data formats are invariably entangled within the application code.
>
> To address these difficulties, it is essential to develop codes which are as "data format free" as possible ...

We are trying to address the same concerns, but at a higher level of abstraction. For example, rather than a matrix, our corresponding fundamental object is a linear operator.

Several groups of researchers have undertaken to provide a mathematical framework, each similar in limited respects to that proposed here. Our effort was motivated directly by the work of the Stanford Exploration Project in geophysical software design; see [18]. The initial design of HCL was worked out in discussions with this group of researchers and with Dr. Lester Dye. The CWP Object-Oriented Optimization Library (COOOL) due to Deng et al. [3] consists of classes similar to some found in HCL, along with a variety of optimization algorithms for linear and nonlinear problems. A similar package is OPT++, designed by Meza [14]; this provides classes useful for defining unconstrained and constrained optimization problems and related algorithms. The Numerical Analysis Objects (NAO) project [7] includes a set of abstract base classes defining the objects involved in simulations and the solution of partial differential equations—geometric regions, functions, operators, and so forth.

HCL differs from all of these efforts in that its base classes specify *only* those objects essential to optimization in Hilbert space, reserving all access and other functions not necessary for this purpose to the derived classes. Also, we attempt to define classes at the highest level of mathematical abstraction—as much as possible, the interfaces to our classes reflect the mathematical definitions and do not refer to any implementation details. We believe that a direct linkage between the Hilbert space environment and the computer code is highly advantageous for managing large and complex real-world applications. This environment is flexible enough to allow a wide variety of applications to be treated, and is still simple enough to allow a manageable design.

2 The Hilbert Class Library

For the purposes of describing the design of HCL and the style of programming it supports, we will use the following application. Consider a linearly elastic, isotropic, inhomogeneous membrane, which, when at rest, occupies a set Ω in the plane. The membrane is fixed around the boundary and a small pressure f is applied in the transverse direction, causing the membrane to be displaced vertically. Under these assumptions, the vertical displacement u of the membrane satisfies the following Dirichlet problem:

$$-\nabla \cdot (a \nabla u) = f \text{ in } \Omega \quad (1)$$

$$u = 0 \text{ on } \partial\Omega. \quad (2)$$

Here the field $a(x)$, $x \in \Omega$, describes the elastic properties of the membrane. We consider the inverse problem of identifying a from a measurement of the displacement u. This problem has been studied by several authors; see, for example, Falk [8], Ito and Kunisch [12].

We pose the problem as an Output Least-Squares (OLS) problem:

$$\min_a \quad \frac{1}{2}\|u[a]-u^m\|^2_{L^2(\Omega)} \tag{3}$$

$$s.t. \quad a \geq \epsilon > 0, \tag{4}$$

where u^m is the observed data and $u[a]$ is the simulated displacement for the estimated a; that is, $u[a]$ and a satisfy (1)–(2).

Of course, this formulation requires the choice of a discretization and a simulation; we discretize the domain Ω using triangular elements and represent both u and a by continuous piecewise linear functions. The simulated displacement $u[a]$ will be computed by the finite element method. (For simplicity, we use the same symbol for the function u and an approximation to it from the appropriate finite element space, and similarly for a and f.)

We now list the mathematical objects that appear in this formulation of the problem:

- The discretized functions u and a are **vectors** in appropriate **vector spaces**.
- The finite element simulation involves forming the stiffness matrix E for each choice of the parameter a and solving the equation $E[a]u = f$. The matrix $E[a]$ defines an **invertible linear operator**.
- The objective function,

  $$J(a) = \frac{1}{2}\|u[a]-u^m\|^2_{L^2(\Omega)}, \tag{5}$$

  is a real-valued function defined on a vector space, that is, a **functional**.
- The ill-posed nature of the problem leads to the use of Tikhonov regularization [24]. The objective function becomes

  $$J(a) = \frac{1}{2}\|u[a]-u^m\|^2_{L^2(\Omega)} + \frac{\rho}{2}\|\nabla a\|^2_{L^2(\Omega)} \tag{6}$$

  (where ρ is the regularization weight). This new objective function is a **linear combination of functionals**.

The design of HCL is intended to allow the various component parts of the project to be put together to form an objective function, which can then be minimized by a generic optimization algorithm (i.e. an algorithm written without knowledge of this or any specific application).

Below we describe the organization of HCL and some of the important classes in detail, using the above example as an illustration. For a complete description of the software, we refer the reader to the documentation (see Section 4 for details on obtaining the software and documentation).

Before proceeding, we mention a few technicalities concerning the design of HCL. At the time of this writing, many compilers do not support ANSI/ISO C++. Because of our concern for portability, we attempt to use only features of C++ likely to be supported by any reasonable compiler. In particular, we provide single and double precision versions of the classes, rather than a single templated version; this explains the significance of the `_s` and `_d` suffixes in the code fragments below. For example, `HCL_Vector_d` is the double precision class. We use C-style (unprotected) casts and primitive error-handling code. These comments refer to HCL 1.0. Later versions will incorporate templates, dynamic casts, exception handling, and other advanced C++ features.

2.1 Organization

The Hilbert Class Library consists of four categories of classes:

- **Core classes:** These classes define the properties (interfaces) for vector spaces, vectors, linear operators, bilinear operators, functionals, and (nonlinear) operators. There are also "evaluation" classes, which do not correspond directly to mathematical constructs, but are used with functionals and operators for efficient evaluation of derivatives.
- **Tool classes:** These classes define common mathematical constructs that can be built up, in a canonical fashion, from the simpler core classes. An example is the class representing a linear combination of functionals. Assuming that one has already defined several functionals, this class makes it trivial to define a linear combination of them.
- **Algorithm classes:** These classes define interfaces for common types of algorithms, such as those for solving linear equations and unconstrained minimization problems.
- **Concrete classes:** These classes are derived from the core classes and represent specific implementations of various objects. For instance, two types of vectors are defined: a simple in-core Euclidean vector and an out-of-core vector designed to represent discretized distributed parameters.

2.2 HCL Core Classes

We begin by listing the core classes and the mathematical objects they represent:

- `HCL_VectorSpace`: A vector space
- `HCL_Vector`: A vector
- `HCL_Functional`: A real-valued function defined on a vector space
- `HCL_LinearOp`: A linear operator mapping one vector space to another
- `HCL_BiLinearOp`: A bilinear operator
- `HCL_Op`: A (nonlinear) operator from one vector space to another

In addition to the classes listed here, the core classes include "evaluation" classes, which we describe below. These classes form the core of HCL because any attempt to formulate an optimization problem will necessarily involve these mathematical objects.

2.2.1 `HCL_VectorSpace`

This is an abstract base class, and therefore serves to define the properties that all derived classes must share.

In HCL, vector spaces are explicitly defined so that two operations can be performed:

- test for equality of two vectors spaces (usually for purposes of error checking);

- create a vector from a given vector space.

These are properties shared by all vector spaces; that is, they are properties of the base class `HCL_VectorSpace`. A particular vector space may have additional properties; for example, a description of an underlying grid.

The class `HCL_VectorSpace` has the following member functions: `operator==`, `operator!=` and `Member`. The `operator!=` member enables one to write

```
if ( U != V )
   // Error condition
```

assuming that `U` and `V` are objects of type `HCL_VectorSpace`.

The member function `Member` is useful for allocating temporary vectors used to store intermediate stages of a computation. It returns a pointer to a dynamically allocated vector. Such a function is called a *virtual constructor*; it allows construction of an object when the exact type of the object is not known at compilation but only at execution.

A virtual constructor is required because the standard mechanism for dynamically allocating objects in C++, the **new** operator, requires knowledge of the exact type at compilation. Our primary reason for developing HCL is to allow the creation of optimization programs that are not tied to specific data representations; this implies that exact types are only known at execution.

2.2.2 `HCL_Vector`

A Hilbert space associates several operations with its vectors: vector addition, scalar multiplication, and inner product. In HCL, these are properties of the class `HCL_Vector`; that is, every vector can be added to another vector (from the same space!), multiplied by a scalar, and paired with another vector in an inner product. One vector can also be copied to another. Also, a vector can identify the vector space to which it belongs. Table 1 lists these member functions. If a method produces a scalar, it is the return value of the method (for example, `a = x.Inner(y)`), while if the method modifies a vector, then the vector invoking the method is the one modified (for example, `x.Add(y)`).

`Add`	`x.Add(y);`	$x \leftarrow x + y$
`Mul`	`x.Mul(a);`	$x \leftarrow ax$
`Inner`	`a = x.Inner(y);`	$a \leftarrow \langle x, y\rangle$
`Copy`	`x.Copy(y);`	$x \leftarrow y$
`Space`	`x.Space();`	Reference to vector space

Table 1: `HCL_Vector` member functions (name, syntax, and effect)

Beyond these basic properties, the class `HCL_Vector` includes methods that combine two or more operations into a single function call. These are provided for convenience in coding and for efficiency, and are listed in Table 2.

Those familiar with the C++ language may wonder why these operations are not provided through overloaded operators. Any straightforward implementation of overloaded operators involves an unacceptable overhead in time and memory. This overhead arises in the creation of temporary objects to hold the results of intermediate computations.

Beyond these basic operations, the class `HCL_Vector` also defines a large number of "component-wise" operations. For example, the method `DiagScale` computes the component-wise product of two vectors. Other such methods include `DiagRecipScale` (component-wise division), several versions of `Max` and `Min` (e.g. to compute the maximum component of a vector, or to compute a component-wise maximum of two vectors), `Abs` (component-wise absolute value), and `Sign` (component-wise sign). For a complete list of `HCL_Vector` methods, we refer the reader to the HCL1.0 documentation (see Section 4).

The `HCL_Vector` class provides a `Component` method, which allows access to an arbitrary component of a vector. However, its use is discouraged for two reasons. First, when the vector components are stored in-core, the overhead of a virtual function is significant compared to a simple data access. Second, when the vector components are stored out-of-core, accessing a single component involves prohibitive overhead due to I/O. For this reason, component-wise operations should be performed using the methods described in the previous paragraph, which can be implemented efficiently for each derived vector class.

`Add`	`x.Add(y,z);`	$x \leftarrow y + z$
`Mul`	`x.Mul(a,y);`	$x \leftarrow ay$
`Neg`	`x.Neg();`	$x \leftarrow -x$
`Zero`	`x.Zero();`	$x \leftarrow 0$
`Norm`	`a = x.Norm();`	$a \leftarrow \|x\|$
`Norm2`	`a = x.Norm2();`	$a \leftarrow \|x\|^2$
`Sub`	`x.Sub(y);`	$x \leftarrow x - y$
`Sub`	`x.Sub(y,z);`	$x \leftarrow y - z$
`ScaleAdd`	`x.ScaleAdd(a,y);`	$x \leftarrow ax + y$
`ScaleAdd`	`x.ScaleAdd(a,y,z);`	$x \leftarrow ay + z$
`AddScale`	`x.AddScale(a,y);`	$x \leftarrow x + ay$

Table 2: Additional member functions of `HCL_Vector` (name, syntax, and effect)

The simplest derived vector class would represent vectors in real n-space by storing their components in a one-dimensional array. HCL defines such a class, `HCL_RnVector`, and a corresponding vector space, `HCL_RnSpace`. These are concrete classes.

A more instructive example is a class we have defined for use with seismic data processing problems. Standard seismic data formats have long been used to store field data and the relevant physical fields (density, acoustic velocity, and so forth). We have defined a vector class called `SGFVector` (for Sampled Grid Function) that provides an interface to a disk file in one of the standard formats. The constructor for this class automatically reads data from a disk file, and the class allows access to the grid description as well as to the data samples themselves. Nonetheless, HCL optimization code can operate on an `SGFVector` just as easily as on an `HCL_RnVector`. The ability of HCL to hide these implementation details from generic algorithms is a valuable simplifying mechanism (and our original motivation for developing this software).

2.2.3 HCL_Functional

There are two fundamental meeting points between a generic algorithm and application-specific software: vectors and functionals. Just as the standard Fortran representation of a vector as a one-dimensional array is too restrictive, so is the use of subroutine calls to implement functionals.

A typical objective function is defined by various parameters and usually built up from simpler objects, such as operators and data. These objects should be packaged with the functional when it is passed to the optimization algorithm. The usual Fortran "work-arounds" to this problem are reverse communication and the use of parameter arrays. In our opinion, the first is inelegant and leads to code that is difficult to understand and maintain. The second requires the packing and unpacking of data to and from a primitive data structure, which, in our experience, is tedious and error-prone.

The C++ class mechanism provides a solution to the problem, because a class can contain both data and code. The `HCL_Functional` class is an abstract base class representing a real-valued function defined on a vector space. This class can identify the domain of the functional it represents and can evaluate the functional, as well (perhaps) its gradient and Hessian, at a point.

The fundamental methods of `HCL_Functional` are `Domain`, `Value`, `Gradient`, and `Hessian`; these are described in Table 3.

`Domain`	`f.Domain();`	Reference to vector space
`Value`	`fx = f.Value(x);`	$fx \leftarrow f(x)$
`Gradient`	`f.Gradient(x,g);`	$g \leftarrow \nabla f(x)$
`Hessian`	`H = f.Hessian(x);`	$H \leftarrow \nabla^2 f(x)$
`Evaluate`	`eval = f.Evaluate(x);`	Creates evaluation object

Table 3: Some `HCL_Functional` member functions (name, syntax, and effect). Note that the `Hessian` method returns a pointer to the (newly created) object.

In addition to these methods, `HCL_Functional` also has a method called `Evaluate`. This method takes as input a vector x in the domain of the functional f, and returns an "evaluation object"—an instance of `HCL_EvaluateFunctional`—that represents the triple

$$(f(x), \nabla f(x), \nabla^2 f(x)).$$

The reason for the evaluation object is efficiency. In many applications, the calculations of $f(x)$, $\nabla f(x)$, and $\nabla^2 f(x)$ involve the computation of intermediate quantities that contribute to each of $f(x)$, $\nabla f(x)$, and $\nabla^2 f(x)$. For example, computing $J(a)$ (see (6)) requires the formation of the finite element matrix $E[a]$, the computation of $u[a]$ by solving $E[a]u = f$, and the calculation of the residual $u[a] - u^m$. If $\nabla J(a)$ is computed by the adjoint state method, then the first step is the computation of $E[a]^{-1}(u[a] - u^m)$ (the "adjoint state"). A natural implementation, and the one we chose, is to represent the vectors using `HCL_RnVector` and form $E[a]$ as a sparse matrix. The equation $E[a]u = f$ is solved using a sparse LU factorization of $E[a]$. One then wishes to save the LU factors and the residual vector for computing the gradient in case it is later requested. Without the evaluation object, this is not possible, because there is no place to save these values; we need an object representing the realization of f at a specific point x as well as the object representing f itself.

The methods of `HCL_EvaluateFunctional` parallel those of `HCL_Functional`. The methods `Domain`, `Value`, `Gradient`, and `Hessian` have the same effect as the corresponding methods of the functional class, except that in the case of the latter three methods, the vector x need not be input—it is intrinsic to the evaluation object itself.

In addition, `HCL_EvaluateFunctional` has methods called `ValueRef`, `GradientRef`, and `HessianRef`; the purpose of these methods is to ensure that the value, gradient, and Hessian are computed only once at a given point, and also to manage the allocation and de-allocation of storage for the gradient vector. These methods are a convenience; they allow the programmer to use the evaluation object as a data structure as well as a mechanism for generating the needed quantities. Instead of allocating a vector `g`, calling `eval->Gradient( g )` to put the value of the gradient in `g` (`eval` is a pointer to the evaluation object), and later de-allocating `g`, the programmer can just use `eval->GradientRef()`, which returns a reference to the gradient vector. Table 4 summarize the methods in the class `HCL_EvaluateFunctional`.

Both the functional class and the corresponding evaluation class allow access to $f(x)$, $\nabla f(x)$, and $\nabla^2 f(x)$. The methods in the functional class work as follows: they create an evaluation object, extract the needed value, and delete the evaluation object. They may be used for simplicity when there is no need to save any intermediate computations.

Indeed, from the point of view on the person implementing a specific functional, the entire mechanism involving the evaluation object can be ignored if desired. The implementor has the option of creating a single class, derived from `HCL_Functional`, to represent a functional f. In this case, the corresponding evaluation object will be an instance of `HCL_FunctionalDefaultEval` (a default class already implemented), the implementor need only code the computation of $f(x)$, $\nabla f(x)$, and $\nabla^2 f(x)$, and there will be no re-use of intermediate results in these computations. On the other hand, if it is important for reasons of efficiency to store intermediate quantities, the implementor can create two classes, one derived from `HCL_Functional` and the other from `HCL_EvaluateFunctional`, and have complete freedom to re-use intermediate results. The mechanisms for choosing one option or the other are described in detail in [9]; this report implements the OLS functional J discussed above as a concrete example.

The following code fragment, taken from a line search algorithm, illustrates the use of the functional and evaluation classes. In this code, the function value is used to test the "sufficient decrease" condition common in line searches; then, if the condition holds, the gradient is used to check the slope condition. By the use of evaluation objects, we neither require that both the function and gradient be evaluated at the same time (when it is possible that the gradient will not be used), nor require that the gradient be computed "from scratch" (i.e. without access to the intermediate quantities already computed).

```
xnext.AddScale( mu,xcur,pdir ); // Compute x+ = x + μp
if( !First )                    // Delete old eval. object
   HCL_delete( eval );
eval = f.Evaluate( xnext );     // Compute new eval. object
First = 0;
f_xnext = eval->ValueRef();     // Get new value
```

```
NumFcnSampled++;
if( f_xnext <= f_x + alpha*mu*initslope )
{
   // Sufficient decrease in function value;
   // check for sufficient decrease in slope
   newslope = eval->GradientRef().Inner( pdir );
   if( newslope < beta*initslope )
   ⋮
```

Domain	`f.Domain();`	Reference to vector space
Value	`fx = eval->Value();`	$fx \leftarrow f(x)$
Gradient	`eval->Gradient(g);`	$g \leftarrow \nabla f(x)$
Hessian	`eval->Hessian();`	$H \leftarrow \nabla^2 f(x)$
ValueRef	`fx = eval->ValueRef();`	$fx \leftarrow f(x)$
GradientRef	`eval->GradientRef();`	Reference to $\nabla f(x)$
HessianRef	`eval->HessianRef();`	Reference to $\nabla^2 f(x)$

Table 4: Some `HCL_EvaluateFunctional` member functions (name, syntax, and effect)

It would be well at this point to explain a significant aspect of the design of HCL: class interfaces include methods that may not be useful or even defined for certain objects. For example, `HCL_Functional` has methods to implement $f(x)$, $\nabla f(x)$, and $\nabla^2 f(x)$, even though the class may be used to represent a functional that is not differentiable, or that is only once differentiable. Similarly, `HCL_LinearOp` has methods implementing Lx, L^*y, $L^{-1}y$, and $L^{-*}x$, even though it may be used to represent an operator which is not invertible. Methods that are not appropriate for a particular derived class should be implemented as errors.

We originally designed a hierarchy of base classes to carefully reflect the available methods for a given object. For instance, the class `HCL_Functional` had (in pre-version 1.0 HCL) only the `Value` method. The `Gradient` method was present in the derived class `HCL_FunctionalGrad`, while the `Hessian` method appeared in `HCL_FunctionalHess`. However, this approach resulted in an excessive number of classes, particularly among "tool" classes, an important part of HCL. (A tool class is used to combine fundamental objects to represent more complicated objects. For example, there is a tool class combining several linear operators in a linear combination. Under the old, hierarchical design, there were two such classes, depending on whether the underlying linear operators were of the type `HCL_LinearOpAdj` (adjoint implemented) or `HCL_LinearOp` (adjoint not implemented).) The current design implies that some error checking has been deferred to run time, but it results in a much cleaner collection of classes.

2.2.4 Other core classes

The other core classes are the operator classes, representing linear, bilinear, and nonlinear operators. For example, `HCL_LinearOp` is the base class for linear operators. It has methods

- `Domain`, `Range`,

- `Image`, `AdjImage`, `InvImage`, `InvAdjImage`.

The `Image` method computes the action of the operator on a vector:

`L.Image(x,y); //` $y \leftarrow Lx$

Similarly, `AdjImage` computes the action of L^*, `InvImage` computes the action of L^{-1}, and `InvAdjImage` computes the action of L^{-*}. Note that these latter two operations may not be meaningful (or may not be needed) for a particular operator; in that case, the corresponding methods can be implemented as errors.

Using the linear operator and vector classes, one can define a variety of algorithms for the iterative solution of linear equations. Below is most of the `Solve` method found in the `HCL_PCG` class, which implements the preconditioned conjugate gradient algorithm. It is worth emphasizing the truly generic nature of this code. There is no assumption about the representation of the vectors—they can be stored in core or on disk, in any desirable data structure. There is no assumption about the nature of the operator or the preconditioner—the operator could be a sparse matrix or a finite difference simulation, and the preconditioner could be a fast Poisson solver or an incomplete factorization.

```
int HCL_PCG::Solve( const HCL_LinearOp & A,const HCL_Vector & b,
                          HCL_Vector & x ) const
{
   if( A.Domain() != MInv->Range() )
   {
      cerr << "Error in HCL_PCG::Solve: domains of A and M "
              "do not agree" << endl;
      exit(1);
   }
   .
   .  (More error checking)
   .
   // Get algorithmic parameters

   int ItnMax;
   if( ParamTable->GetValue( "MaxItn",ItnMax ) )
      ItnMax = 100;
   .
   .  (More parameters)
   .
   // Allocate needed objects

   HCL_Vector *r = MInv->Domain().Member();
   HCL_Vector *u = MInv->Range().Member();
   HCL_Vector *p = MInv->Range().Member();
   HCL_Vector *v = MInv->Domain().Member();
```

```
// compute initial residual

A.Image( x,*v );
r->Sub( b,*v );

// compute u = Minv*r

MInv->Image(*r, *u);
p->Copy( *u );

// Main iteration

double res0 = b.Norm();
double res = r->Norm();
double ratio = res/res0;
int itn = 0;
double rtu = r->Inner( *u );
while( ratio > Tol && itn < ItnMax )
{
   // compute v = Ap

   A.Image( *p,*v );

   // compute step length alpha

   double ptv = p->Inner( *v );
   double alpha = rtu/ptv;

   // update x and r

   x.AddScale( alpha,*p );
   r->AddScale( -alpha,*v );

   // compute u = Minv*r

   MInv->Image(*r, *u);

   // compute the new relative residual

   res = r->Norm();
   ratio = res/res0;
   itn++;
```

```
        // compute beta and the new search direction p

        double rtu1 = r->Inner( *u );
        double beta = rtu1/rtu;
        p->ScaleAdd( beta,*u );
        rtu = rtu1;
    }

    HCL_delete( v );
    HCL_delete( p );
    HCL_delete( u );
    HCL_delete( r );

    .
    .  (Display messages if desired and return)
    .
}
```

HCL_BiLinearOp is the base class for bilinear operators of the form $B : X \times Y \to Z$, where X, Y, and Z are Hilbert spaces. Bilinear operators are important in HCL because they arise naturally as the second derivatives of nonlinear operators. This class has methods for computing the image, $B(x, y)$, as well as for creating the linear operators $x \mapsto B(x, y)$ and $y \mapsto B(x, y)$.

HCL_Op is an abstract base class for representing nonlinear operators and their derivatives. Note that if F is a nonlinear operator, then $DF(x)$ is a linear operator and $D^2F(x)$ is a bilinear operator. In order to handle the derivatives efficiently, we define evaluation objects for operators as we did for functionals. The implementation of operators and bilinear operators is described in detail in [10], which includes a detailed concrete example involving the application described in Section 2.

2.3 Tool classes

In addition to the core classes, HCL defines a number of classes that combine the core classes to represent standard mathematical constructs. For instance, here is a code fragment that forms the OLS functional J (without regularization—see (5)) using the tool class HCL_LeastSquaresFcnl:

```
EllipticSolOp3_d F( "mesh" );
HCL_RnVector_d u( "data" );
HCL_LeastSquaresFcnl_d J( &F,&u );
```

Note that the operator mapping a to $u[a]$, the solution of the BVP (2), is implemented in an operator class called EllipticSolOp3; its constructor reads the necessary mesh information from the file named "mesh". The data for the OLS functional is read from a file named "data" and used to create an instance of `HCL_RnVector`. Then the OLS functional itself is create as an instance of `HCL_LeastSquaresFcnl`, which stores the operator and the data.

The advantage of using a tool class such as `HCL_LeastSquaresFcnl` is that it eliminates unnecessary code. Given the operator and the data, the OLS functional and its derivatives can be computed in a purely mechanical (and efficient) fashion. The code to do this must be written only once (in the tool class).

At the time of this writing, HCL defines more than a half-dozen such classes; these include:

- `HCL_GenericProductVector`: a vector from a product space,
- `HCL_LinCombLinearOp`: a linear combination of linear operators,
- `HCL_CompLinearOp`: a composition of linear operators,
- `HCL_LinCombLinearFcnl`: a linear combination of functionals,
- `HCL_BlockOp`: a linear operator defined by "blocks,"

and others.

2.4 Algorithm classes

The purpose of HCL is to define an environment in which high-quality algorithms can be written and used to solve complex problems. It is hoped that numerical analysts developing algorithms will consider implementing them using HCL classes.

To demonstrate the feasibility of doing this, we have taken several popular algorithms for large-scale problems and translated them into HCL. Here we discuss the general framework for algorithms, and two specific examples: Nocedal's limited memory BFGS algorithm for unconstrained minimization [19] and Sorensen's implicitly-restarted Arnoldi method for large-scale eigenvalue problems [23].

We have defined HCL classes to represent the algorithms themselves. The reason for doing so is that complicated algorithms are often built up out of simpler algorithms; if these algorithms are objects, then it is easier to put the building blocks together and also to experiment with various choices.

For instance, consider a minimization algorithm, such as limited memory BFGS, that is based on a line search. Several popular line search algorithms might be suitable. Because we define a line search base class, the choice of line search algorithm can be deferred to the user.

At this time our algorithm base classes are tentative and rather simple. We discuss three:

- `HCL_LinearSolver`: Solves a linear operator equation (mainly intended for iterative algorithms such as PCG);
- `HCL_LineSearch`: Searches for a minimizer of a functional along a line segment;
- `HCL_UMin`: Minimizes a nonlinear functional subject to no constraints.

Each of these classes contains just two methods: `Parameters` and `Solve` or `Search` or `Minimize`, respectively. The `Parameters` method provides a way to access or change the scalar parameters (such as stopping tolerances) needed by each algorithm (details are found in the documentation).

Here is a fragment of a main program that invokes the limited memory BFGS algorithm; note how the line search is chosen and passed to the BFGS constructor:

```
HCL_LineSearch * line;     // Choose the line search
if( flag == 1 )            // Dennis & Schnabel line search
   line = new HCL_LineSearch_DS( "lsearch.dat" );
else if( flag == 0 )       // Fletcher line search
   line = new HCL_LineSearch_Fl( "lsearch.dat" );
else if( flag == 2 )       // More & Thuente line search
   line = new HCL_LineSearch_MT( "lsearch.dat" );
HCL_UMin_lbfgs umin( line,"umin.dat" );  // Create the minimization
                                         // algorithm
umin.Minimize( f,a );      // Minimize f using starting point a
```

In the above code, the algorithmic parameters are read from files by the constructors.

3 Performance

The use of C++ for scientific computation often raises concerns about run-time efficiency. Because the use of pointers introduces the aliasing problem, it has traditionally been more difficult to design optimizing compilers for C than for Fortran. C++ inherits this characteristic from C, and, in addition, some of its most important constructs, such as virtual functions, incur run-time overhead.

In this section we argue that HCL, if used properly, can achieve efficiency comparable to code written entirely in Fortran. There are two basic keys to achieving high-efficiency with C++:

- Avoid defining virtual functions that perform very little computation. Because C++ supports in-lined functions (that is, function calls that are replaced by the compiler with equivalent executable code), there is no reason to avoid the clarity provided by function calls because of concerns about the function call overhead. However, virtual functions cannot be in-lined, so care must be exercised.

- Take advantage of the fact that C++ allows mixed-language programming. Simulators and other computationally-intensive parts of the code can be written in Fortran or some other highly efficient language.

The second point is the key. HCL is designed to ease the high-level organization of the code, while allowing the application scientist as much flexibility as possible. This flexibility includes the ability to choose the most efficient programming language.

HCL was designed specifically for large-scale optimization problems in which the target functional or simulation requires a significant amount of computation. In an application that is dominated by the generic optimization or linear algebra costs, HCL might be noticeably slower than a similar Fortran code (though not by as much as one might expect). We give examples below. We also point out that the performance gap between C++ and Fortran, while it might never be eliminated, is decreasing (see [22], for example).

3.1 The limited memory BFGS algorithm

The limited memory BFGS algorithm, due to Nocedal [19], is a variant on the popular BFGS algorithm for unconstrained minimization. These are both quasi-Newton methods that build increasingly good Hessian approximations as the iteration proceeds. Since the limited memory version defines the approximation to the inverse Hessian in terms of outer products of vectors, it is easily implemented in HCL.

Below we give part of the main loop of the `Minimize` method from the class `HCL_UMin_lbfgs`. Note that the inverse Hessian approximation has been implemented as a class derived from `HCL_LinearOp`.

```
// Compute the lmBFGS search direction (the inverse Hessian
// approximation is a linear operator pointed to by H)

H->Image( feval->GradientRef(),*dir );
dir->Neg();

// Perform line search

HCL_EvaluateFunctional *tmp_eval = feval;
feval = LineSearch->Search( f,x,*xnext,*dir,tmp_eval );
if( feval == NULL ) feval = tmp_eval;
LineSearch->Parameters().GetValue("TermCode", ls_result);
LineSearch->Parameters().GetValue("MaxTkn", MaxTkn);
Itn++;
TermCode = StoppingTest(x,*feval,ls_result, MaxTkn);
if( TermCode )
{
   if( ls_result >= 0 )
   {
      HCL_delete( tmp_eval );
      x.Copy( *xnext );
      f_x = feval->ValueRef();
   }
   Display( f,x );
   Clean();
   return TermCode;
}

// Update lmBFGS inverse Hessian approximation
```

```
if( ls_result )    // LineSearch failed; reset to steepest descent
{                  // direction
   H->Reset();
   StDescDir = 1;
}
else
{
   H->Update( x,*xnext,tmp_eval->GradientRef(),
              feval->GradientRef() );
   StDescDir = 0;
}
```

We now compare the performance of this algorithm with the Fortran version (LBFGS) implemented by Nocedal and available from the Netlib software repository. We apply both codes to two problems:

1. the extended Rosenbrock function [16], a simple test problem for unconstrained minimization with varying dimension n;

2. the OLS functional (6).

Both algorithms call the same code to compute the function value and gradient. Results are presented in Tables 5 and 6, where "f calls" is the number of function and gradient evaluations and the times are in seconds on a Sun SPARC 10 workstation. (There are slight differences between Nocedal's and our implementation, so the numbers of iterations and function evaluations are slightly different.) The results suggest that there is no significant difference in performance between the two algorithms.

n	f calls (LBFGS)	f calls (HCL)	time (LBFGS)	time (HCL)
1,000	49	58	0.13	0.12
10,000	53	58	1.59	1.59
100,000	52	58	33.04	34.29

Table 5: Comparison of Nocedal's LBFGS code with HCL_UMin_lbfgs on the extended Rosenbrock function.

n	f calls (LBFGS)	f calls (HCL)	time (LBFGS)	time (HCL)
121	69	77	5.51	5.51
441	75	78	29.67	31.93
1681	77	75	181.96	183.21

Table 6: Comparison of Nocedal's LBFGS code with HCL_UMin_lbfgs on the elliptic inverse problem (6).

3.2 Implicitly-restarted Arnoldi

The implicitly-restarted, k-step Arnoldi algorithm (see [23]) is an algorithm for computing a few eigenvalues and eigenvectors of a linear operator. It has been implemented in a Fortran package called ARPACK [13]. We implemented ARPACK within the HCL framework, creating an algorithm class called `HCL_IRArnoldi`, and present here a comparison of the performance of the two codes.

The comparison is a little more complicated than it was for the limited memory BFGS algorithm. In the latter case, the linear algebra performed by the optimization code consists of level-1 operations; since both LBFGS and HCL use explicit loops when the vectors are in-core, the two codes cannot differ greatly in performance. The most significant linear algebra performed in the Arnoldi method is the Gram-Schmidt algorithm; ARPACK uses the classical Gram-Schmidt algorithm instead of the more stable modified Gram-Schmidt technique in order to allow the use of level-2 BLAS (the use of re-orthogonalization alleviates concerns about stability in this context).

As currently structured, HCL does not allow the use of level-2 BLAS in the same context—the fundamental object is a basis for a subspace, conveniently represented as a "matrix" of column vectors, which does not admit the use of the BLAS when the vectors are abstract (the vectors are not stored as columns in an in-core array). Therefore, when the computational cost is dominated by the generic linear algebra calculations, ARPACK can be significantly faster than `HCL_IRArnoldi` on a platform with optimized level-2 BLAS. (Countering this is HCL's ability to deal with essentially any vector storage scheme, and its avoidance of reverse communication.) On the other hand, when calculation of the action of the operator on a vector is expensive, the two codes show similar performance.

Moreover, if the performance penalty described above were considered a serious drawback, it could be eliminated in the following manner. Define methods to create and manipulate bases for subspaces in the vector space base class, and require them to be implemented for each concrete vector space. Then the implementor of the vector class would have the option of using level-2 BLAS or similarly efficient algorithms appropriate for the storage type. The resulting gain in efficiency would have to be balanced against the added burden imposed on those programmers who need to implement their own vector classes.

To illustrate the above discussion, we compared ARPACK and `HCL_IRArnoldi` on the following two problems:

1. a finite-difference operator representing the convection-diffusion operator

$$\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \rho \frac{\partial}{\partial x}$$

   on the unit square, subject to Dirichlet boundary conditions. This is a test problem shipped with ARPACK.

2. The Hessian of the OLS functional (6).

Note that in the first problem, the action of the operator is quite inexpensive, while in the second it is relatively expensive.

The results are presented in Tables 7 and 8, in which we report the problem size and the time to perform 10 iterations of the Arnoldi method. The results are as expected: ARPACK is noticeably faster on the first problem, and the two codes show the same performance on the second.

These results are obtained using the optimized BLAS provided on the SGI Power Challenge, using one processor. For the first problem, on which ARPACK is significantly faster, the performance gap depends on the size of the problem (HCL takes anywhere from 31% longer to 106% longer). This is presumably due to the dependence of the efficiency of the optimized BLAS on the size of the fast cache memory on the SGI machine. Also, although we don't report the results here, ARPACK was no faster than HCL if the generic BLAS were used.

For the convection-diffusion problem, we used an $N \times N$ grid, a Krylov subspace of length 25, and $\rho = 20$. We asked for 4 eigenvalues and the corresponding eigenvectors. For the OLS problem, the same parameters were used, except the length of the Krylov subspace was 20.

N	time per 10 iterations (s) (ARPACK)	time per 10 iterations (s) (HCL)
20	0.49	0.71
40	0.95	1.70
60	1.75	3.44
80	2.92	5.87
100	4.44	9.04
120	6.32	13.00
140	9.57	18.27
160	15.84	25.64
180	25.30	35.12
200	35.39	46.45

Table 7: Comparison of ARPACK with HCL_IRArnoldi on the convection-diffusion example.

N	time per 10 iterations (s) (ARPACK)	time per 10 iterations (s) (HCL)
10	0.51	0.55
20	1.85	1.94
30	4.37	4.69
40	8.50	8.99
50	14.66	15.24
60	23.79	24.98
70	40.05	40.90

Table 8: Comparison of ARPACK with HCL_IRArnoldi on the OLS example.

4 Availability of HCL software

The Hilbert Class Library is available under the GNU Library Public License. Source code, together with makefiles and scripts for installation, can be downloaded from the following URL:

www.trip.caam.rice.edu/txt/hcldoc/html/index.html

Queries can also be made by electronic mail to `hcl@caam.rice.edu`.

References

[1] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[2] A.R. Conn, N.I.M. Gould, and Ph.L. Toint. *LANCELOT: a Fortran package for large-scale nonlinear optimization*. Springer-Verlag, New York, 1992.

[3] L. Deng, W. Gouveia, and J. Scales. The CWP object-oriented optimization library. *The Leading Edge*, 15(5):365–369, 1996.

[4] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozo, and K. REMINGTON. A sparse matrix library in C++ for high performance architectures. In *Proceedings of the Second Annual Object-Oriented Numerics Conference*, pages 214–218, 1994.

[5] J. Dongarra, A. Lumsdaine, R. Pozo, and K. REMINGTON. Updated users' guide, IML++ v 1.2. `http://math.nist.gov/iml++/`.

[6] J. J. Dongarra, R. Pozo, and D. W. Walker. Lapack++: A design overview of object-oriented extensions for high performance linear algebra. In *Proceedings of Supercomputing '93*, pages 162–171, 1993.

[7] C.C. Douglas, D.A. George, and M.E. Henderson. Object classes for numerical analysis. In *OON-SKI '94*, pages 32–49, 1994. Proceedings of the Second Annual Object-Oriented Numerics Conference.

[8] R. Falk. Error estimates for the numerical identification of a variable coefficient. *Mathematics of Computation*, 40:537–546, 1983.

[9] Mark S. Gockenbach. Implementing functionals in HCL. Technical Report 99-24, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, USA, 1999.

[10] Mark S. Gockenbach. Implementing nonlinear operators in HCL. Technical Report 99-22, Department of Computational and Applied Mathematics, Rice University, Houston, Texas, USA, 1999.

[11] Mark S. Gockenbach, Matthew J. Petro, and W. W. Symes. C++ classes for linking optimization with complex simulations. *ACM Transactions on Mathematical Software*, 25(2), 1999.

[12] K. Ito and K. Kunisch. The augmented Lagrangian method for parameter estimation in elliptic systems. *SIAM J. Control and Optimization*, 28:113–136, 1990.

[13] R. Lehoucq, D. Sorensen, and P. Vu. *ARPACK User's Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods.* Society for Industrial and Applied Mathematics, Philadelphia, 1998.

[14] J.C. Meza. OPT++: An object-oriented class library for nonlinear optimization. Technical Report 94–8225, Sandia National Laboratories, Sandia National Laboratories, Livermore, CA, 1994.

[15] J.J. Moré, B.S. Garbow, and K.E. Hillstrom. User guide for MINPACK–I. Technical Report ANL-80-74, Argonne National Laboratory, Argonne, IL, 1980.

[16] J.J. Moré, B.S. Garbow, and K.E. Hillstrom. Testing unconstrained optimization software. *ACM Trans. Math. Software*, 7:17–41, 1981.

[17] B.A. Murtagh and M.A. Saunders. MINOS 5.1 user's guide. Technical Report SOL 83–20R, Systems Optimization Laboratory, Stanford University, 1983.

[18] D. Nichols, G. Dunbar, and J. Claerbout. The C++ language in physical science. In *OON-SKI '93*, pages 339–353, 1993. Proceedings of the First Annual Object-Oriented Numerics Conference.

[19] J. Nocedal. Updating Quasi-Newton matrices with limited storage. *Mathematics of Computation*, 35:339–353, 1980.

[20] R. Pozo. MV++: Matrix / Vector user's guide and class reference manual. `http://math.nist.gov/mv++/`.

[21] R. Pozo. Template Numerical Toolkit for linear algebra: High performance programming with C++ and the Standard Template Library. presented at Environments and Tools For Parallel Scientific Computing III, Faverges de la Tour - France, 1996.

[22] A. Robison. C++ get faster for scientific computing. *Computers in Physics*, 10:458–462, 1996.

[23] D. Sorensen. Implicit application of polynomial filters in a k-step arnoldi method. *SIAM J. Matrix Anal. Appl.*, 13:357–385, 1992.

[24] A.N. Tikhonov and V.Y. Arsenin. *Solution of ill-posed problems.* Winston, New York, 1977.