Implementing functionals in HCL

Mark S. Gockenbach*

October 22, 1999

Contents

*Department of Mathematical Sciences, Michigan Technological University, 1400 Townsend Drive, Houghton, MI 49931-1295

1 Introduction

1.1 Functionals and HCL

The standard nonlinear program (NLP) takes the form

$$\begin{array}{ll} \min & J(x), \\ \text{s.t.} & G(x) \leq 0, \\ & H(x) = 0, \end{array}$$

where X and Y are Hilbert spaces, $J : X \to \mathrm{R}$, $H : X \to Y$, and $G : X \to \mathrm{R}^m$. We refer to J as a *functional* and to G, H as *operators*. The Hilbert Class Library (HCL) is a collection of C++ classes representing the fundamental mathematical objects arising in optimization problems defined on Hilbert space. These objects include vector spaces and vectors, linear and nonlinear operators, and (nonlinear) functionals. In this report, we explain how to implement a functional J in an HCL class. To make the discussion as useful as possible, we illustrate the complete process with two realistic examples. The companion report [2] explains how to implement operators in HCL classes.

The functional classes in HCL have been designed with two goals in mind. First of all, it should be possible to implement a functional and its derivatives efficiently. In particular, when the calculation of $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$ involve common, intermediate computations, these computations should be done only once and saved (when memory requirements are not prohibitive). Providing for this efficiency requires a mechanism with a certain degree of complexity. On the other hand, the second goal of the design is to make the implementation of functionals as straightforward as possible. This goal requires that the implementor be able to ignore the complexities if (i) there is no gain in efficiency, or (ii) the gain in efficiency is not regarded as worth the extra work in programming. To achieve both goals, we have designed a collection of classes that can be used in several ways.

The outline of this report is as follows. After a brief overview of HCL and a description of the illustrative example, we explain the most straightforward implementation of an functional in HCL (Section 2). To illustrate this approach, we use the Lennard-Jones potential energy, which models the potential energy of n mutually attracting particles, as an example. This is a good example of an objective function (from a practical application) that is defined algebraically. We then describe an important alternative implementation: saving intermediate computations with an "evaluation object" (Section 3). Finally, Section 4 summarizes the discussion and Appendix A gives a brief guide to choosing an implementation model.

Many functionals arising in practice are defined on product spaces and take the form $J : X \times Y \to \mathrm{R}$ (or $J : X \times Y \times Z \to \mathrm{R}$, etc.). Moreover, algorithms can and in some cases should explicitly recognize this structure. We have designed special classes to represent such functionals. These classes will be described in a forthcoming companion report.

1.2 A brief overview of HCL

It is not our intention to give a detailed introduction to the Hilbert Class Library; for such an introduction we refer the reader to the papers [4, 3, 1]. Here we just briefly mention the aim and basic structure of HCL.

HCL was designed to address the following problem: state-of-the-art optimization software has traditionally been written in procedural programming languages, such as Fortran or C. Accordingly, the optimization software requires a user to implement his functions and operators using specific data structures (usually simple arrays) and fixed interfaces. For many complex applications, this causes difficulties, sometimes insurmountable, because the data structures and interfaces natural for the application are incompatible with those of the optimization software. A simple example of this is a very large scale problem, in which the problem variables form a data set so large that it must be stored out of core.

This difficulty is addressed in HCL through object-oriented design. Base classes are defined to represent the common mathematical objects; classes derived from these can then be manipulated by optimization code without regard to the particular data structures involved in their implementation. The fundamental HCL classes include:

- `HCL_VectorSpace`: This class represents a space of vectors. A vector space can be compared to another vector space (for error checking), and can produce a vector from the space (usually to store intermediate results). Some member functions are listed in Table 1.

`operator!=`	`X != Y`	true if $X \neq Y$, false if $X = Y$
`Member`	`x = X.Member();`	returns a pointer to a dynamically allocated vector

Table 1: Some `HCL_VectorSpace` member functions (name, usage, effect).

- `HCL_Vector`: This class represents a vector. Its member functions include algebraic operations (addition, scalar multiplication), calculation of norm and inner product, and identification of the corresponding vector space. Some member functions are listed in Table 2.

- `HCL_LinearOp`: This class represents a linear operator $L : X \to Y$ mapping one Hilbert space into another. An object of this type can identify the domain and range, and compute the image $y = Lx$, as well as (possibly) $x = L^*y$, $x = L^{-1}y$, and $y = L^{-*}x$. Some member functions are listed in Table 3.

Other HCL classes, most significantly `HCL_Functional`, will be introduced below.

For readers who are unfamiliar with object-oriented design, we emphasize the significance of the HCL design. An optimization algorithm knows that every vector class has an `Add` method, and knows how to invoke it, and the optimization code can do this without knowing the data structures representing the vector. The same holds for a linear operator and its Image

`Add`	`x.Add( y,z );`	$x \leftarrow y + z$
`Norm`	`n = x.Norm();`	$n \leftarrow \|x\|$
`Space`	`x.Space() == X`	true if $x \in X$, false otherwise

Table 2: Some `HCL_Vector` member functions (name, usage, effect).

`Domain`	`x.Space() == L.Domain()`	true if x lies in the domain of L
`Image`	`L.Image( x,y );`	$y \leftarrow Lx$
`AdjImage`	`L.AdjImage( y,x );`	$x \leftarrow L^* y$
`InvImage`	`L.InvImage( y,x );`	$x \leftarrow L^{-1} y$
`InvAdjImage`	`L.InvAdjImage( x,y );`	$y \leftarrow L^{-*} x$

Table 3: Some `HCL_LinearOp` member functions (name, usage, effect).

method. From this it follows that one can write, for example, a conjugate gradient code that can be applied to a wide variety of linear operators (implemented by sparse matrices or finite difference simulation or any other method) and vectors (stored in 1D arrays or disk files or any other data structure).

We close with 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 Implementation of a functional using the base class `HCL_Functional`

To apply a gradient-based HCL optimization algorithm to an optimization problem with objective function J, we must be able to compute $J(x)$, $\nabla J(x)$, and (perhaps) $\nabla^2 J(x)$. The functional J must be implemented in a concrete class derived from `HCL_Functional`, the base class for functionals. We begin by discussing the public interface of `HCL_Functional`.

2.1 The public interface of `HCL_Functional`

The class `HCL_Functional` has 10 public member functions; these comprise the properties of an object of this type, and are therefore closely related to the mathematical properties of a functional. For the sake of this discussion, we consider an operator $J : X \to \mathrm{R}$, where X is a Hilbert space. We assume that a class `XSpace`, derived from `HCL_VectorSpace`, has been chosen to represent X, and that the corresponding vector class is `XVector`. The functional J is implemented in a class derived from `HCL_Functional`. We now explain the 10 member functions of `HCL_Functional` in terms of the mathematical properties of J implemented by them. These 10 member functions can be logically divided into three groups.

2.1.1 Evaluation methods: `Value`, `Gradient`, `Hessian`, and `Evaluate`

The `Value` method implements the computation of $J(x)$ for a particular $x \in X$. Specifically, assuming that `x` is an instance of `XVector`, and that `x` has been given a meaningful value, the command

```
Jx = J.Value(x);
```

assigns to `Jx` the value of $J(x)$.

The `Gradient` method implements the computation of the vector $\nabla J(x)$ for a particular $x \in X$. Recall that $\nabla J(x)$, defined by

$$DJ(x)\delta x = (\nabla J(x), \delta x)_X \ \forall \delta x \in X,$$

is the usual representer of the derivative $DJ(x)$ of J at x, and is the quantity needed in optimization algorithms using first derivatives. The `Gradient` method is invoked as follows:

```
J.Gradient( x,y );
```

Assuming `x` is an instance of `XVector` with a meaningful value, and `y` is another instance of `XVector`, then this command assigns to `y` the value $\nabla J(x)$. Note that the vector `y` must be created to hold the output before the `Value` method is invoked.

The `Hessian` method allows the computation of $\nabla^2 J(x)$. Recall that the linear operator $\nabla^2 J(x)$ is the usual representer of $D^2 J(x)$, and is defined by

$$D^2 J(x)(\delta x, \delta x) = (\nabla^2 J(x)\delta x, \delta x)_X \ \forall \delta x \in X.$$

Since $\nabla^2 J(x)$ is a linear operator, it is represented in HCL by an instance of the base class `HCL_LinearOp`. Assuming `x` is an instance of `XVector` with a meaningful value, and that `H` is an `HCL_LinearOp` *pointer* variable, then the command

```
H = J.Hessian( x );
```

assigns to `H` the address of the object implementing $\nabla^2 J(x)$. Note that the program unit invoking `J.Hessian( x )` is responsible for releasing the dynamically allocated memory pointed to by `H`. The command

```
HCL_delete( H );
```

should be used for this purpose.

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 $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$, 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.

The last of the evaluation methods is called `Evaluate`. It creates an object of type `HCL_EvaluateFunctional`, which is referred to as an *evaluation object*. This class represents the triple $(J(x), \nabla J(x), \nabla^2 J(x))$; that is, it represents the realization of the functional at a particular $x \in X$. The use of evaluation objects is an advanced feature of the design and will be explained in detail below. For now we just point out that, using an evaluation object, one can always implement the computation of J and its derivatives efficiently, even when they share intermediate computations. The evaluation object is a place where such computations can be saved for re-use. We also point out that the `Evaluate` method gives an alternative method of computing $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$. For instance,

```
e = J.Evaluate( x );  // e is an HCL_EvaluateFunctional pointer
Jx = e->Value();
```

is equivalent to

```
Jx = J.Value( x );
```

(The difference is that, by using the evaluation object, intermediate calculations done while computing $J(x)$ can be stored in the evaluation object pointed to by `e`, and they are therefore available if, later in the code, $\nabla J(x)$ is requested.) Optimization algorithms implemented using HCL should use the evaluation object, in case the user has used the evaluation object mechanism to avoid recalculating intermediate quantities.

2.1.2 `Domain` and `MaxStep`

Recall that our generic example is a functional $J : X \to \mathrm{R}$, where the Hilbert space X is represented by an instance of the class `XSpace`. The command `J.Domain()` returns a reference to the instance of `XSpace` representing X, and is typically used for error checking. A typical code fragment is

```
if( x.Space() != J.Domain() )
   // error condition
```

In many cases, the functional J is not defined on all of X, but rather on a subset S of X. In this case, `J.Domain()` still refers to X (so this method is named improperly). HCL contains no classes to represent general subsets, and so it is not possible to refer to the actual domain of J (unless it happens to be all of X). A partial work-around to this problem is provided by the `MaxStep` method. The command

```
alpha = J.MaxStep( x,z );
```

computes the largest scalar α such that $x + \alpha z$ lies in the domain of J. A value of $\alpha = 0$ indicates that x lies in the domain but no $x + \alpha z$ does for $\alpha > 0$ (thus x lies on the boundary of the domain of G), while a value $\alpha < 0$ indicates that x does not lie in the domain of J.

2.1.3 Debugging methods

Two methods allow the implementor of a functional class to test the consistency of the evaluation methods. The method `CheckGrad` allows the implementor to check whether

$$\frac{J(x + h\delta x) - J(x - h\delta x)}{2h} = (\nabla J(x), \delta x)_X + O(h^2),$$

as ought to hold for a sufficiently smooth functional J. It does so by simply computing

$$\frac{\left\| \frac{J(x+h\delta x)-J(x-h\delta x)}{2h} - (\nabla J(x), \delta x)_X \right\|}{\|(\nabla J(x), \delta x)_X\|}$$

for a given $x, \delta x \in X$ and a sequence of values of h. The implementor can then inspect the results to determine whether they are consistent with the expected $O(h^2)$ behavior.

In a similar way, `CheckHess` provides data for testing the correctness of $\nabla^2 J(x)$.

The `Scan` method computes values of J on a line segment, allowing a user to inspect a "slice" of the graph of J. The command is

```
J.Scan( x,dx,n,hmin,hmax,"file" );
```

computes the values

$$J(x + h\delta x), J(x + 2h\delta x), \ldots, J(x + nh\delta x),$$

where $h = (hmax - hmin)/(n - 1)$, and sends them to the screen and to the file name "file". The file name is an optional argument; if it is omitted, the results are only displayed on the screen.

Finally, all HCL classes, including `HCL_Functional`, have a `Write` method. The command

```
J.Write( str );
```

sends a description of the functional `J` to the output stream `str` (often `cout`, the standard output stream). The information provided is determined by the implementor of the class, but it should be useful for debugging, as the `Write` method is often invoked by HCL error-handling code.

2.2 Implementing a functional using the protected image methods

Having described the public interface of `HCL_Functional`, we now give an example of the implementation of a derived class. We first describe the Lennard-Jones potential energy function, a model objective function for the molecular conformation problem, and then show how to implement it in an HCL class.

2.2.1 The Lennard-Jones potential

Suppose $p_1, p_2, \ldots, p_n$ are mutually attractive particles. Then, given a configuration of the particles, there is an associated total potential energy, which is determined by the distances between the particles. If we denote by $d(p_i, p_j)$ the distance between particles i and j, then the Lennard-Jones model of the total potential energy is

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(\frac{1}{d(p_i, p_j)^{12}} - \frac{2}{d(p_i, p_j)^6} \right).$$

The molecular conformation problem is to determine the relative position of the particles in the configuration of minimal potential energy.

To define an objective function, we first describe a configuration by the coordinates in 3-space of the n particles. When the problem is parameterized in this fashion, there is an obvious equivalence between two configurations when one can be obtained from the other by a rigid motion—a combination of translations and rotations. To deal with this we remove the rigid motions from the space of configurations by fixing particle p_1 at the origin, particle p_2 on the x-axis, and particle p_3 in the xy-plane. Therefore, the coordinates of particles $p_1, p_2, p_3, p_4, \ldots, p_k, \ldots, p_n$ are

$$(0,0,0), (x_1,0,0), (x_2,x_3,0), (x_4,x_5,x_6), \ldots, (x_{3k-8}, x_{3k-7}, x_{3k-6}), \ldots, (x_{3n-8}, x_{3n-7}, x_{3n-6}),$$

respectively. We then define

$$s_{ij}(x) = d(p_i, p_j)^2, \;\; i = 1, 2, \ldots, n-1, j = i+1, \ldots, n.$$

The Lennard-Jones potential energy is then defined in terms of x as

$$J(x) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left(\frac{1}{s_{ij}(x)^6} - \frac{2}{s_{ij}(x)^3} \right).$$

Note that $J : \mathrm{R}^{3n-6} \to \mathrm{R}$. The simplest version of the molecular conformation problem is

$$\min_{x \in \mathrm{R}^{3n-6}} J(x). \tag{1}$$

In order to make this discussion as concrete as possible, we will use some legacy Fortran code implementing the calculations of $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$. (As explained in Section 1.2, one of our primary aims in designing HCL is to allow existing simulation code to be linked to high-quality optimization code without the need for the simulation code to be rewritten. Our present example is an illustration.) We have three Fortran subroutines, with the following calling sequences:

- `fnlj(N,x,v)`,
- `gradlj(N,x,g)`,
- `hesslj(N,x,H)`.

The meanings of the arguments is as follows:

- `N` (integer) the number of problem variables $(3n-6)$ (input);
- `x` (real array) the vector of particle coordinates (input);
- `v` (real) the value $J(x)$ (output);
- `g` (real array) the gradient vector (output);
- `H` (real 2D array) the Hessian matrix (output).

We now show how to turn this code into an HCL class.

2.2.2 Implementing the Lennard-Jones potential in an HCL class

In order to apply one of the HCL unconstrained minimization algorithms to solve (1), the functional J must be implemented in a class derived from `HCL_Functional`; we will call this class `LennardJonesEnergy`. The 10 methods discussed in Section 2.1 must be defined for this derived class.

We first note that the testing methods `CheckGrad`,`CheckHess`, and `Scan` are all implemented in the base class, and need not be overridden. Two methods, `Domain` and `Write`, are pure virtual, meaning that they are not implemented in the base class, and *must* be overridden in a derived class. As we will see below, `MaxStep` and the four evaluation methods need not be overridden in this example, although we must implement certain *protected* methods in their place.

Before defining the functional class, we must choose a vector space class for the space of the problem variables. Since $J : \mathrm{R}^{3n-6} \to \mathrm{R}$, we will represent the domain of J by an instance of `HCL_RnSpace`, and $x \in \mathrm{R}^{3n-6}$ by an instance of `HCL_RnVector`. These classes implement the simplest vector space/vector concept; the components of the vector are stored in a 1D array and the vector space is defined by a single integer, the dimension.

We now present the class definition. Note that the private data section contains only the object representing the domain. The class must store the domain, since the `Domain` method returns a *reference* to the domain space. For this problem, this is the only private data required. Here is the class definition

```
class LennardJonesEnergy_d: public HCL_Functional_d
{
   private:
      HCL_RnSpace_d * dom;
```

```
   protected:

      virtual double Value1( const HCL_Vector_d & x ) const;
      virtual void Gradient1( const HCL_Vector_d & x,HCL_Vector_d & y ) const;
      virtual HCL_LinearOp_d * Hessian1( const HCL_Vector_d & x ) const;

   public:

      LennardJonesEnergy_d();
      LennardJonesEnergy_d( const LennardJonesEnergy_d & );
      LennardJonesEnergy_d( int n );
      virtual ~LennardJonesEnergy_d();

      virtual HCL_VectorSpace_d & Domain() const;

      virtual ostream & Write( ostream & str ) const;
};
```

We must define a constructor and destructor for the class. For this example, these are trivial: the constructor just creates the `HCL_RnSpace` object representing the domain, and the destructor deletes it. The function `HCL_delete` is used to delete the domain object; this function is part of the HCL mechanism for implementing reference counting. Here is the code for the constructor and destructor.

```
LennardJonesEnergy_d::LennardJonesEnergy_d( int n )
{
   dom = new HCL_RnSpace_d( 3*n-6 );
}

LennardJonesEnergy_d::~LennardJonesEnergy_d()
{
   HCL_delete( dom );
}
```

(The class definition also includes default and copy constructors; these are implemented to return error messages, since they are not intended to be used.)

The `Domain` and `Write` methods are trivial:

```
HCL_VectorSpace_d & LennardJonesEnergy_d::Domain() const
{
   return *dom;
}
```

```
ostream & LennardJonesEnergy_d::Write( ostream & str ) const
{
   str << "Lennard-Jones energy functional on " << (dom->Dim()+6)/3
       << " particles"
       << endl;
   str << "Domain is" << endl;
   return Domain().Write( str );
}
```

The last method, other than the evaluation methods, is **MaxStep**. For this example, since the domain of J is all of R^{3n-6}, we need not override this function, which is defined in the base class to return the largest floating point number (signifying infinity).

Now we turn to the evaluation methods:

`Value`, `Gradient`, `Hessian`, `Evaluate`.

It is important to understand that the functional class must be able to create two independent objects:

- the `HCL_EvaluateFunctional` "evaluation object";
- the `HCL_LinearOp` object representing the Hessian $\nabla^2 J(x)$.

For reasons of efficiency, it may be necessary to create new classes to represent these objects (particularly the evaluation object). However, we have also provided default classes:

- `HCL_FunctionalDefaultEval` to represent the evaluation object;
- `HCL_FunctionalDefaultHessian` to represent Hessian.

In our present example, it is natural to use the default evaluation object, since there is no efficiency gain in defining a new evaluation class. However, the Hessian is most conveniently represented using a matrix operator class provided by HCL, rather than by the default Hessian class.

We first discuss the evaluation class. An evaluation class must be able to compute $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$. The default evaluation class will refer requests for these quantities to the protected evaluation methods `Value1`, `Gradient1`, and `Hessian1` of `LennardJonesEnergy`. Therefore, to enable the use of the default evaluation object, we must implement these three methods.

The first two methods, `Value1` and `Gradient1`, are quite simple to implement. The code consists of checking the consistency of the arguments to the methods, and then calling the Fortran code described above. Here is the code for the `Value1` and `Gradient1` methods:

```
double LennardJonesEnergy_d::Value1( const HCL_Vector_d & x ) const
{
   if( x.Space() != Domain() )
   {
```

```
      cerr << "Error in LennardJonesEnergy::Value1: input "
              "vector is not in domain" << endl;
      exit(1);
   }
   HCL_RnVector_d & c = (HCL_RnVector_d&)x;
   int n = dom->Dim();
   double val;
   F77NAME(fnlj)( n,c.Data(),val );
   return val;
}

void LennardJonesEnergy_d::Gradient1( const HCL_Vector_d & x,
                                          HCL_Vector_d & y ) const
{
   if( x.Space() != Domain() )
   {
      cerr << "Error in LennardJonesEnergy::Gradient1: input "
              "vector is not in domain" << endl;
      exit(1);
   }
   HCL_RnVector_d & c = (HCL_RnVector_d&)x;
   if( y.Space() != Domain() )
   {
      cerr << "Error in LennardJonesEnergy::Gradient1: output "
              "vector is not in domain" << endl;
      exit(1);
   }
   HCL_RnVector_d & g = (HCL_RnVector_d&)y;
   int n = dom->Dim();
   F77NAME(gradlj)( n,c.Data(),g.Data() );
}
```

The `Hessian1` method is slightly more complicated, because it must create an new object, the Hessian operator, to return to the calling routine. It is natural to use an ordinary (dense) matrix to represent the Hessian. For this purpose, we use the class `HCL_GeneralMatOp`, which is implemented using `GeneralMatrix`, a stand-alone matrix class (not part of HCL proper). Here is the code for `Hessian1`:

```
HCL_LinearOp_d * LennardJonesEnergy_d::Hessian1(
                    const HCL_Vector_d & x ) const
{
   if( x.Space() != Domain() )
   {
      cerr << "Error in LennardJonesEnergy::Hessian1: input "
```

```
                "vector is not in domain" << endl;
       exit(1);
     }
     HCL_RnVector_d & c = (HCL_RnVector_d&)x;
     int n = dom->Dim();
     GeneralMatrix_d * h = new GeneralMatrix_d( n,n );
     F77NAME(hesslj)( n,c.Data(),h->data );
     HCL_GeneralMatOp_d * H = new HCL_GeneralMatOp_d( h,1,&(c.Space()),
                                                       &(c.Space()) );
     return H;
   }
```

Note that `Hessian1` returns a *pointer* to a dynamically-allocated object representing $\nabla^2 J(x)$ (in this case, the object is a matrix operator). The routine invoking `Hessian1` is responsible for deleting this object using `HCL_delete` when it is no longer needed. Failure to do this, particularly in an iterative algorithm, can cause all available memory to be used to store unneeded objects.

When the method `Hessian1` is used to create the Hessian operator, as it is here, the companion methods `HessianImage` and `HessianInvImage`, should not be implemented in the derived class.

As described in Section 1.2, a linear operator such as a Hessian operator is defined mainly by its `Image` method (as well as the `AdjImage`, `InvImage`, and `InvAdjImage` methods). In some applications, it is not convenient or not feasible to compute the matrix representation of the Hessian operator, but it is still possible to compute its action on a vector, $\nabla^2 J(x)\delta x$ (and perhaps the action of its inverse). In this case, the default Hessian class can be used. To enable this, the protected virtual functions `HessianImage` and (possibly) `HessianInvImage` should be implemented, and the method `Hessian1` left unimplemented. (When one of these methods is left unimplemented in a derived class, the method will be defined by its implementation in the base class, which will be an error message or some other appropriate action.) The effect of the method `HessianImage` must be to compute the action of the Hessian; specifically,

```
J.HessianImage( x,dx,dy );
```

assigns to the vector `dy` the value of $\nabla^2 J(x)\delta x$, where `x` and `dx` have been previously given the values of x and δx, respectively. Similarly, if it is known how to compute $\nabla^2 J(x)^{-1}\delta y$, then

```
J.HessianInvImage( x,dy,dx );
```

should assign to `dx` the value of $\nabla^2 J(x)^{-1}\delta y$. If it is not convenient or not necessary to compute $\nabla^2 J(x)^{-1}\delta y$, then this method should be left unimplemented. (Any algorithm which depends on on the inverse image of $\nabla^2 J(x)$ will result in an error message.)

The above evaluation methods,

`Value1`, `Gradient1`, `Hessian1`, `HessianImage`, `HessianInvImage`,

`v = J.Value1( x );`	$v \leftarrow J(x)$
`J.Gradient( x,y );`	$y \leftarrow \nabla J(x)$
`H = J.Hessian1( x );`	$H \leftarrow \nabla^2 J(x)$
`J.HessianImage( x,dx,dy );`	$\delta y \leftarrow \nabla^2 J(x) \delta x$
`J.HessianInvImage( x,dy,dx );`	$\delta x \leftarrow \nabla^2 J(x)^{-1} \delta y$

Table 4: `HCL_Functional` protected image methods (usage, effect). Note that `Hessian1` returns a *pointer* to a dynamically-allocated object representing $\nabla^2 J(x)$.

are protected, meaning that they are not part of the public interface of `HCL_Functional` (or any derived class). They exist only so that a class implementor can take advantage of the default evaluation and Hessian classes, obviating the necessity of writing new classes for these objects. The protected evaluation methods are summarized in Table 4.

By way of summary, we now describe how each public evaluation method of the class `LennardJonesEnergy` functions.

- `Evaluate` (implemented in `HCL_Functional`) Passes the input vector x and the address of the functional J to the constructor for for `HCL_FunctionalDefaultEval`, and returns the address of this newly created evaluation object.

- `Value` (implemented in `HCL_Functional`) Creates an evaluation object using `Evaluate` and calls the `Value` method of the evaluation object. The `HCL_FunctionalDefaultEval` `Value` method then calls the `LennardJonesEnergy` protected `Value1` method, which performs the actual computation. (Although this sequence of events may sound more complicated than necessary, it is in fact needed to allow several alternative methods of implementing functionals in HCL. Also, note that all of this involves very little overhead—just the copying of some pointers and a few function calls—compared to the floating point computation that must be done by any implementation. Finally, note that these contortions have been coded into the base class; the implementor of the `LennardJonesEnergy` need only code the image methods, that is, the actual computations.)

- `Gradient` (implemented in `HCL_Functional`) Analogous to `Value`.

- `Hessian` (implemented in `HCL_Functional`) Creates an evaluation object using `Evaluate` and calls the `Hessian` method of the evaluation object. The `HCL_FunctionalDefaultEval` `Hessian` method calls the `Hessian1` method of `LennardJonesEnergy` to get (the address of) the Hessian object, which becomes the return value of the original invocation of `Hessian`. (Note that if `Hessian1` is not implemented in `LennardJonesEnergy`, then the default implementation returns the NULL pointer, which tells the Hessian method of the evaluation object to create an instance of `HCL_FunctionalDefaultHessian` instead.

3 Defining an evaluation object for increased efficiency

The implementation model presented in Section 2 is inadequate for many applications, for the following reason: the computation of $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$ involve common, intermediate calculations. When this is the case, it is not possible to reuse these calculations if a single class represents J, because there is no place to store them. For such an application, we must create two classes, one to represent the functional J, and the other to represent the triple $(J(x), \nabla J(x), \nabla^2 J(x))$—the realization of J at a single point x. This second object, the *evaluation object*, can store intermediate quantities pertaining to the computation of J, ∇J, and $\nabla^2 J$ at the point x.

In order to illustrate this technique, we describe a version of one of the HCL "tool" classes, `HCL_LeastSquaresFcnl`. This operator takes an operator of the form $F : X \to Y$, where X and Y are Hilbert spaces, and a vector d in Y, and creates the least-squares functional $J : X \to \mathrm{R}$ defined by

$$J(x) = \frac{1}{2}\|F(x) - d\|_Y^2.$$

Operators are implemented in HCL as instances of classes derived from `HCL_Op`. The least-squares functional J can be created automatically, since $J(x)$, $\nabla J(x)$ and $\nabla^2 J(x)$ can be computed using $F(x)$, $DF(x)$, and $D^2F(x)$, all of which are provided in `HCL_Op`.

Indeed, we have

$$J(x) = \frac{1}{2}\left(F(x) - d, F(x) - d\right)_Y,$$

so

$$\begin{aligned} DJ(x)\delta x &= (DF(x)\delta x, F(x) - d)_Y \\ &= (\delta x, DF(x)^*(F(x) - d))_X . \end{aligned}$$

This shows that

$$\nabla J(x) = DF(x)^*(F(x) - d).$$

Also,

$$\begin{aligned} D^2J(x)(\delta x, \delta x) &= (DF(x)\delta x, DF(x)\delta x)_Y + \left(D^2F(x)(\delta x, \delta x), F(x) - d\right)_Y \\ &= (\delta x, DF(x)^*DF(x)\delta x)_X + \left(\delta x, (D^2F(x)(\delta x, \cdot))^*\,(F(x) - d)\right)_X . \end{aligned}$$

In this formula, $D^2F(x)$ is a bilinear operator, so $D^2F(\delta x, \cdot)$ is a linear operator mapping X into Y and $(D^2F(\delta x, \cdot))^*$ is its adjoint. If we define the operator $L_3 : X \to X$ by

$$\delta x \mapsto \left(D^2F(\delta x, \cdot)\right)^* (F(x) - d),$$

then the above formula becomes

$$D^2J(x)(\delta x, \delta x) = (\delta x, (DF(x)^*DF(x) + L_3)\,\delta x)_X ,$$

which shows that

$$\nabla^2 J(x) = DF(x)^*DF(x) + L_3.$$

Note that we label the above operator L_3 because $D^2F(x)$ is a bilinear operator, and every bilinear operator $B : X \times Y \to Z$ naturally defines three linear operators:

$$\begin{aligned} &L_1 : Y \to Z, \quad L_1 y = B(x, y) \quad (x \in X \text{ fixed}), \\ &L_2 : X \to Z, \quad L_2 x = B(x, y) \quad (y \in Y \text{ fixed}), \\ &L_3 : X \to Y, \quad L_3 x = B(x, \cdot)*^z \quad (z \in Z \text{ fixed}). \end{aligned}$$

The class HCL_BiLinearOp, which is the base class for bilinear operators like $D^2F(x)$, has methods for computing these three operators. For more details about both HCL_Op and HCL_BiLinearOp, see the companion report *Implementing nonlinear operators in HCL* ([2]).

Now, in certain applications, the computation of $F(x)$, $DF(x)$, and $D^2F(x)$ require common intermediate quantities. For example, if F implements a finite element solution operator, then to compute any of $F(x)$, $DF(x)$, or $D^2F(x)$ requires the same stiffness matrix, which is expensive to compute (see [2] for a detailed example of this). Therefore, we should not implement the corresponding least-squares functional using the mechanisms described in Section 2, because that implementation model "de-couples" the computation of $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$.

As mentioned above, the HCL mechanism for coupling the computation of $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$ is the evaluation object. When this mechanism is used, the protected evaluation methods

Value1, Gradient1, Hessian1, HessianImage, HessianInvImage

are not used, nor is the class HCL_FunctionalDefaultEval. Instead, the virtual method Evaluate of HCL_Functional is overridden to create an instance of an evaluation class written especially for the particular J in question. This evaluation class, as we have already indicated, represents the triple $(J(x), \nabla J(x), \nabla^2 J(x))$ and provides a place to store the intermediate computations common to the three quantities.

In the case of HCL_LeastSquaresFcnl, it is particularly easy to implement this evaluation class, which is called HCL_EvalLSFcnl, because the quantities needed for $(J(x), \nabla J(x), \nabla^2 J(x))$ are found in the triple $(F(x), DF(x), D^2F(x))$. Thus HCL_EvalLSFcnl just obtains and stores the HCL_EvaluateOp object associated with F (and the particular x in question).

The HCL class HCL_LeastSquaresFcnl also implements regularization in a convenient way. For the purposes of this discussion, the regularization feature obscures the main points under discussion. Therefore, we present a "stripped-down" version of HCL_LeastSquaresFcnl, which does not include regularization. We call this class LeastSquaresFcnl, and the corresponding evaluation class EvalLSFcnl.

Here is the definition of LeastSquaresFcnl.

```
class LeastSquaresFcnl_d: public HCL_Functional_d
{
   private:
      HCL_Op_d * N;
      HCL_Vector_d * d;

   public:

      LeastSquaresFcnl_d();
      LeastSquaresFcnl_d( const LeastSquaresFcnl_d & );
      LeastSquaresFcnl_d( HCL_Op_d * Operator,
                          HCL_Vector_d * Data );
```

```
        virtual ~LeastSquaresFcnl_d();

        virtual HCL_VectorSpace_d & Domain() const;

        virtual HCL_EvaluateFunctional_d *
                  Evaluate( const HCL_Vector_d & x ) const;

        virtual ostream & Write( ostream & str ) const;
    };
```

As we mentioned earlier, that the `Evaluate` method is overridden in this method, while the protected evaluation methods are not. The only data needed to define a least-squares functional are the operator and the data vector; these are stored in the private data section of `LeastSquaresFcnl`.

The principle method of `LeastSquaresFcnl` is `Evaluate`, which is one line!

```
    HCL_EvaluateFunctional_d * LeastSquaresFcnl_d::Evaluate(
                               const HCL_Vector_d & x ) const
    {
         return new EvalLSFcnl_d( x, N, d );
    }
```

This method merely passes the input vector `x`, along with the operator `N` and the data vector `d`, to the constructor of `EvalLSFcnl`.

Here is the definition of the class `EvalLSFcnl`.

```
    class EvalLSFcnl_d: public HCL_EvaluateFunctional_d
    {
       private:
          HCL_Op_d * N;
          HCL_EvaluateOp_d * Neval;
          HCL_Vector_d * d;
          HCL_Vector_d * x;
          HCL_Vector_d * r;
       public:

          EvalLSFcnl_d();
          EvalLSFcnl_d( EvalLSFcnl_d & F);
          EvalLSFcnl_d( const HCL_Vector_d & Input,
                        HCL_Op_d * Operator,
                        HCL_Vector_d * Data );
          virtual ~EvalLSFcnl_d();
```

```
        virtual HCL_VectorSpace_d & Domain() const;

        virtual double Value() const;

        virtual void Gradient( HCL_Vector_d & g ) const;

        virtual HCL_LinearOp_d * Hessian() const;

        virtual ostream & Write( ostream & str ) const;
};
```

The intermediate quantities need to compute $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$ are $(F(x), DF(x), D^2F(x))$, as well as the residual $F(x) - d$. These are computed in the constructor of `EvalLSFcnl`.

```
EvalLSFcnl_d::EvalLSFcnl_d( const HCL_Vector_d & Input,HCL_Op_d * Operator,
                            HCL_Vector_d * Data )
{
   N=Operator;N->IncCount();
   d =Data;d->IncCount();
   x = Input.Space().Member();
   x->Copy( Input );

   if (x->Space() != N->Domain())
   {
      cerr<<"Error: EvalLSFcnl_d"<<endl;
      cerr<<"input vector not in domain"<<endl;
      exit(1);
   }

   if (d->Space() != N->Range())
   {
      cerr<<"Error: EvalLSFcnl_d"<<endl;
      cerr<<"data vector not in range"<<endl;
      exit(1);
   }

   Neval = N->Evaluate(*x);

   // compute residual

   r=N->Range().Member();
   r->Sub( Neval->ImageRef(),*d );
```

```
}
```

Given these values, it is now easy to produce $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$. Here is the code for the methods `Value`, `Gradient`, and `Hessian` of `EvalLSFcnl`. Notice that

- the method `Op3` of `HCL_BiLinearOp` is used to obtain the linear operator L_3 discussed above;
- the Hessian of the least-squares functional is implemented as the sum of the operator $DF(x)^* DF(x)$ and L_3 using the HCL tool class `HCL_LinCombLinearOp`, which implements a general linear combination of linear operators.

```
double EvalLSFcnl_d::Value() const
{
   return 0.5*(r->Norm2());
}

void EvalLSFcnl_d::Gradient( HCL_Vector_d & g ) const
{
   if (g.Space() != Domain())
   {
      cerr<<"Error: EvalLSFcnl_d::Gradient"<<endl;
      cerr<<"gradient vector not in domain"<<endl;
      exit(1);
   }

   (Neval->DerivRef()).AdjImage(*r,g);
}

HCL_LinearOp_d * EvalLSFcnl_d::Hessian() const
{
   HCL_LinearOp_d * HH = Neval->Deriv();
   HCL_LinearOp_d * n1 = new HCL_Normal_d( HH );
   HCL_delete( HH );
   HCL_LinCombLinearOp_d * t = new HCL_LinCombLinearOp_d(
                                    &(Domain()),&(Domain()),2 );
   t->SetNext( n1,1.0 );
   HCL_BiLinearOp_d * D2Nx = Neval->SecondDeriv();
   HCL_LinearOp_d * L3 = D2Nx->Op3( *r );
   t->SetNext( L3,1.0 );
   HCL_delete( L3 );
   return t;
}
```

At the risk of repeating ourselves, we wish to emphasize the reason for the evaluation object. By providing a place to save $(F(x), DF(x), D^2F(x))$ and $F(x) - d$, we ensure that the least-squares functional is implemented as efficiently as possible (assuming, of course, that the same care was taken in the implementation of the class representing F).

Many functionals arising in practical applications can be built up from operators and HCL tool classes without the need for a special functional class. However, when this is not true for a particular application, the implementor can use the mechanism used to implement `LeastSquaresFcnl` to obtain optimal runtime efficiency.

4 Summary

As detailed in this paper, there are several methods for implementing functionals (objective functions) in HCL. These various methods exist to enable the most efficient runtime performance while requiring the least programming effort. In particular, as we have described, the advanced mechanisms that are necessary in some examples for efficiency, need not be used when there is no advantage in doing so.

The brief guide presented in Appendix A summarizes four basic models for implementing a functional in HCL. An implementor should be able to choose one of these models so as to obtain good runtime performance with minimal programming effort.

A Guide to implementation models in HCL

Models for implementing a functional J in HCL

1. (a) **Indications:**

   i. The computation of $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$ do not share common (nontrivial) intermediate quantities.

   ii. There is no convenient (existing) class for implementing the Hessian.

   (b) **Method:** Create a single class, `JFcn`, derived from `HCL_Functional`. Override the protected member functions

   - `Value1`,
   - `Gradient1`,
   - `HessianImage`, and
   - `HessianInvImage` (if desired).

   Do *not* override the member functions `Evaluate` and `Hessian1`.

   (c) **Comments:** The evaluation object produced by `Evaluate` will be an instance of `HCL_FunctionalDefaultEval`, and the Hessian will be an instance of

   `HCL_FunctionalDefaultHessian`.

2. (a) **Indications:**

   i. The computation of $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$ do not share common (nontrivial) intermediate quantities.

   ii. The Hessian operator can be implemented using an existing class.

   (b) **Method:** Create a single class, `JFcn`, derived from `HCL_Functional`. Override the protected member functions `Value1`, `Gradient1`, `Hessian1`. Do *not* override the protected member functions

   - `HessianImage`,
   - `HessianInvImage`,

   Do *not* override the public member function `Evaluate`.

   (c) **Comments:** The evaluation object produced by `Evaluate` will be an instance of `HCL_FunctionalDefaultEval`, and the Hessian will be an instance of the class chosen by the implementor.

3. (a) **Indications:**

   i. The computation of $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$ share common (nontrivial) intermediate quantities.

   ii. The Hessian operator can be implemented using an existing class.

   (b) **Method:** Create two classes, `JFcn` (derived from `HCL_Functional`) and `EvalJFcn` (derived from `HCL_EvaluateFunctional`). Override the public member function `Evaluate` of `HCL_Functional` to call the constructor of `EvalJFcn`. Do not override any of the `HCL_Functional` functions

- `Value1`,
- `Gradient1`,
- `Hessian1`,
- `HessianImage`,
- `HessianInvImage`,

Override the public member functions

- `Value`,
- `Gradient`,
- `Hessian`

of `HCL_EvaluateFunctional`. The `Hessian` method creates an instance of the class chosen by the implementor.

(c) **Comments:** The evaluation object produced by `Evaluate` will be an instance of `EvalJFcn`, and the Hessian will be an instance of the class chosen by the implementor.

4. (a) **Indications:**

    i. The computation of $J(x)$, $\nabla J(x)$, and $\nabla^2 J(x)$ share common (nontrivial) intermediate quantities.

    ii. There is no convenient (existing) class for implementing the Hessian.

(b) **Method:** Create two classes, `JFcn` (derived from `HCL_Functional`) and `EvalJFcn` (derived from `HCL_EvaluateFunctional`). Override the public member function `Evaluate` of `HCL_Functional` to call the constructor of `EvalJFcn`. Do not override any of the `HCL_Functional` functions

- `Value1`,
- `Gradient1`,
- `Hessian1`,
- `HessianImage`,
- `HessianInvImage`,

Override the public member functions `Image` and `Gradient` and the protected image methods

- `HessianImage`,
- `HessianInvImage` (if desired),

of `HCL_EvaluateFunctional`. Do not override the public member function `Hessian` of `HCL_EvaluateFunctional`.

(c) **Comments:** The evaluation object produced by `Evaluate` will be an instance of `EvalJFcn`, and the Hessian will be an instance of `HCL_FunctionalDefaultHessian`.

References

[1] Mark S. Gockenbach. An overview of HCL1.0. (In preparation).

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

[3] 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.

[4] Mark S. Gockenbach and William W. Symes. The Hilbert Class Library: a library of abstract C++ classes for optimization and inversion. *Computers and Mathematics with Applications*, 32, 1996.