Implementing nonlinear operators in HCL

Mark S. Gockenbach*

September 21, 1999

1 Introduction

1.1 Operators and HCL

Many optimization problems involve (nonlinear) operators of the form

$$G : X \to Y,$$

where X and Y are Hilbert spaces. For example, the Output Least-Squares (OLS) formulation of the inverse problem for G takes the form

$$\min_{x \in X} \frac{1}{2} \|G(x) - y^{obs}\|^2,$$

where $y^{obs} \in Y$ is the observed data. Another common example is an optimization problem with equality constraints:

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

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, in some detail, how to implement a nonlinear operator G in an HCL class. To make the discussion as useful as possible, we illustrate the complete process with a realistic example, the standard "elliptic" inverse problem (see below).

The operator classes in HCL have been designed with two goals in mind. First of all, it should be possible to implement operators and their derivatives efficiently. In particular, when the calculation of $G(x)$, $DG(x)$, and $D^2G(x)$ 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 operators as straightforward as possible. This goal requires that the implementor be able

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

`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).

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 operator in HCL (Section 2). We then describe various alternatives: using pre-existing HCL classes to represent derivatives (Section 3), and saving intermediate computations (Section 4). Finally, Section 5 summarizes the discussion and present a brief guide to choosing an implementation model.

Many operators arising in practice are defined on product spaces and take the form $G : X \times Y \to Z$ (or $G : X \times Y \times Z \to W$, etc.). Moreover, algorithms can and in some cases should explicitly recognize this structure. We have designed special operator classes to represent such operators. 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 [3, 2, 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.

- `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.

`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).

- `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_Op`, 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. Similarly for a linear operator and its Image 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.

1.3 A model problem

Consider the elliptic boundary value problem (BVP)

$$\begin{aligned} -\nabla \cdot (a \nabla u) &= f \text{ in } \Omega, \\ u &= 0, \end{aligned} \tag{1}$$

where Ω is a connected open set in the plane, a, u and f all map Ω into R, and $a(x,y) \geq a_{min} > 0$ for all $(x,y) \in \Omega$. This BVP models, for example, the small vertical displacement

u of an elastic membrane clamped around its boundary and subjected to a transverse force described by f. The membrane occupies the set $\{(x,y,0)|(x,y)\in\Omega\}$ when unloaded and $\{(x,y,u(x,y))|(x,y)\in\Omega\}$ when the force is applied. The field a describes the elastic properties of the (inhomogeneous) membrane. In the forward (or direct) problem, one is given a and f and wishes to determine u. This amounts to solving the BVP in the usual sense, and can be done (approximately) using the finite element method. The inverse problem is to estimate a from f and an observation of u. One way to formulate the inverse problem is through Output Least-Square:

$$\min \frac{1}{2}\|F(a)-u^{obs}\|^2, \tag{2}$$

where $u=F(a)$ solve (1) (f will be regarded as fixed throughout this discussion). To seriously treat the inverse problem, we should impose the constraint $a\geq a_{min}$, and also apply some type of regularization (as the inverse problem is ill-posed). However, our purpose is not to discuss the resolution of the inverse problem, but rather to describe various implementations in HCL of the operator F.

Let $\mathcal{T}$ be a triangulation of Ω, let $\mathcal{A}$ be the space of continuous piecewise linear functions with respect to $\mathcal{T}$, and let $\mathcal{U}$ be the subspace of $\mathcal{A}$ consisting of those functions which are zero on the boundary of Ω (we can assume for the sake of this discussion that Ω is polygonal). We define $F:\mathcal{P}\to\mathcal{U}$ by $F(a)=u$, where u is the Galerkin approximation to the solution of (1),

$$u\in\mathcal{U},\ \int_\Omega a\nabla u\cdot\nabla v=\int_\Omega fv\ \forall v\in\mathcal{U}, \tag{3}$$

and $\mathcal{P}=\{a\in\mathcal{A}|a(x,y)>0\forall(x,y)\in\Omega\}$. Let $\{\psi_1,\psi_2,\ldots,\psi_M\}$ be the standard basis for $\mathcal{A}$ and $\{\phi_1,\phi_2,\ldots,\phi_N\}$ be the standard basis for $\mathcal{U}$. Then we can write

$$a=\sum_{i=1}^{M}\alpha_i\psi_i,\ \ u=\sum_{i=1}^{N}\mu_i\phi_i$$

and we see that $\mathcal{A}\simeq \mathrm{R}^M$, $\mathcal{U}\simeq\mathrm{R}^N$. Abusing notation, we will write both

$$u=\sum_{i=1}^{N}\mu_i\phi_i\in\mathcal{U}$$

and

$$u=\begin{bmatrix}\mu_1\\ \mu_2\\ \vdots\\ \mu_N\end{bmatrix}\in\mathrm{R}^N;$$

context will determine which is intended. Now, (3) is equivalent to

$$Ku=b,$$

where $K=K(a)\in\mathrm{R}^{N\times N}$, $b\in\mathrm{R}^N$,

$$K_{ij}=\int_\Omega a\nabla\phi_j\cdot\nabla\phi_i,$$

$$b_i = \int_\Omega f \phi_i.$$

Thus we compute $F(a)$ in two steps:

1. Assemble the "stiffness" matrix $K(a)$ and the load vector b;
2. Solve $K(a)u = b$ to get u.

To make this discussion as concrete as possible, we wrote a Fortran subroutine called `assemble` to construct $K(a)$ and b from a description of the mesh, a function (subprogram) implementing f, and the vector $a \in \mathrm{R}^M$. The sparse matrix $K(a)$ is stored in compressed row storage, and the Yale SparsePack code can be used to solve $K(a)u = b$. Here is the calling sequence for `assemble` and the meaning of the input and output variables:

- Calling sequence:

```
assemble(M,N,L,NodeList,NodePtrs,ElList,a,ik,jk,k,nzk,maxnzk,b)
```

- Input variables:
  - `M,N,L` (integers) number of nodes, interior nodes, and triangles, respectively, in the mesh;
  - `NodeList, NodePtrs, ElList` (arrays) description of mesh;
  - `a` (array of length `N`) coefficients $\alpha_1, \alpha_2, \ldots, \alpha_M$;
  - `maxnzk` (integer) maximum number of nonzeros in K.
- Output variables:
  - `ij,jk,k,nzk` storage for the matrix K in compressed row storage;
  - `b` (array of length `N`) components of the load vector b.

This code will form the basis of the `Image` method for the HCL operator implementing F.

Now, if we write

$$J(a) = \frac{1}{2}\|F(a) - u^{obs}\|^2$$

for the OLS function, we see that

$$\nabla J(a) = DF(a)^*(F(a) - u^{obs}). \tag{4}$$

It follows that we will need $DF(a)$, which is a linear operator mapping $\mathcal{A}$ into $\mathcal{U}$, and $DF(a)^*$, its adjoint. Writing $\delta u = DF(a)\delta a$, we have

$$\begin{aligned} K(a)u = b \quad &\Rightarrow \quad DK(a)\delta a u + K(a)\delta u = 0 \\ &\Rightarrow \quad \delta u = -K(a)^{-1} DK(a)\delta a u \\ &\Rightarrow \quad \delta u = -K(a)^{-1} DK(\delta a)u; \end{aligned}$$

the last step follows from the fact that $K(a)$ depends linearly on a. Thus

$$DF(a)\delta a = -K(a)^{-1}K(\delta a)u.$$

This is a satisfactory formula for computing $DF(a)\delta a$, but it does not show us how to compute $DF(a)^*\delta u$. To put the expression for $DF(a)\delta a$ in a more useful form, we note that, since $K(\delta a)u \in \mathrm{R}^N$ depends linearly on δa, there is a linear operator $B = B(u) : \mathrm{R}^M \to \mathrm{R}^N$ such that

$$B(u)\delta a = K(\delta a)u \;\forall u \in \mathcal{U}, \delta a \in \mathcal{A}.$$

To compute $B(u)$, we note that

$$\begin{aligned}
(B(u)\delta a)_i &= (K(\delta a)u)_i \\
&= \sum_{j=1}^{N} K_{ij}\mu_j \\
&= \sum_{j=1}^{N} \mu_j \int_\Omega \delta a \nabla\phi_j \cdot \nabla\phi_i \\
&= \sum_{j=1}^{N}\sum_{k=1}^{M} \mu_j \delta\alpha \int_\Omega \psi_k \nabla\phi_j \cdot \phi_i \\
&= \sum_{k=1}^{M} \delta\alpha \sum_{j=1}^{N} \mu_j \int_\Omega \psi_k \nabla\phi_j \cdot \phi_i \\
&= \sum_{k=1}^{M} \delta\alpha \int_\Omega \psi_k \nabla u \cdot \phi_i \\
&= \sum_{k=1}^{M} B_{ik}\delta\alpha,
\end{aligned}$$

where

$$B_{ik} = \int_\Omega \psi_k \nabla u \cdot \nabla\phi_i.$$

Given the definition of B, we see that

$$DF(a)\delta a = -K(a)^{-1}B(u)\delta a \quad (u = F(a));$$

since $K(a)$ is symmetric, we then obtain

$$DF(a)^*\delta u = -B(u)^*K(a)^{-1}\delta u \tag{5}$$

(we assume that the adjoints are computed with respect to the Euclidean inner product).

To enable us to compute $DF(a)^*$, we wrote a Fortran subroutine called `assemble2` to compute B. The calling sequence is

```
assemble2(M,N,L,NodeList,NodePtrs,ElList,u,ib,jb,b,nzb)
```

and the variables are analogous to those of `assemble`.

Finally, writing $\delta u = DF(a)\delta a$, $\delta u_1 = DF(a)\delta b$, and $\delta^2 u = D^2F(a)(\delta a, \delta b)$, we have

$$\begin{aligned} K(a)\delta u = -B(u)\delta a \quad &\Rightarrow \quad DK(a)\delta b\delta u + K(a)\delta^2 u = DB(u)\delta u_1\delta a \\ &\Rightarrow \quad K(\delta b)\delta u + K(a)\delta^2 u = -B(\delta u_1)\delta a = -K(\delta a)\delta u_1 \\ &\Rightarrow \quad K(a)\delta^2 u = -K(\delta a)\delta u_1 - K(\delta b)\delta u \\ &\Rightarrow \quad \delta^2 u = -K(a)^{-1}(K(\delta a)\delta u_1 + K(\delta b)\delta u). \end{aligned}$$

Thus

$$D^2F(a)(\delta a, \delta b) = -K(a)^{-1}(K(\delta a)\delta u_1 + K(\delta b)\delta u). \tag{6}$$

We can thus compute $D^2F(a)(\delta a, \delta b)$ using the quantities we already have. Later we will use this formula to compute $\nabla^2 J(a)$.

2 Implementation of an operator using the base class `HCL_Op`

To apply a gradient-based HCL optimization algorithm to 2, we must be able to compute $J(a)$ and $\nabla J(a)$, which require $F(a)$ and $DF(a)$. (We defer the discussion of $\nabla^2 J(a)$ and $D^2F(a)$ until Section 2.3 below.) The operator F must be implemented in a concrete class derived from `HCL_Op`, the base class for (nonlinear) operators. We begin by discussing the public interface of `HCL_Op`.

2.1 The public interface of `HCL_Op`

The class `HCL_Op` has 12 public member functions; these comprise the properties of an object of this type, and are therefore closely related to the mathematical properties of an operator. For the sake of this discussion, we consider an operator $G : X \to Y$, where X and Y are Hilbert spaces. We assume that classes `XSpace` and `YSpace`, both derived from `HCL_VectorSpace`, have been chosen to represent X and Y, respectively, and that the corresponding vector classes are `XVector` and `YVector`, respectively. The operator G is implemented in a class derived from `HCL_Op`. We now explain the 12 member functions of `HCL_Op` in terms of the mathematical properties of G implemented by them. These 12 member functions can be logically divided into three groups.

2.1.1 Evaluation methods: `Image`, `Deriv`, `SecondDeriv`, and `Evaluate`

The `Image` method implements the computation of $G(x)$ for a particular $x \in X$. Specifically, assuming that `x` and `y` are instances of `XVector` and `YVector`, respectively, and that `x` has been given a meaningful value, the command

```
G.Image(x,y);
```

assigns to y the value of $G(x)$. Note that the vector y must be created to hold the output before the Image method is invoked.

The Deriv method implements the computation of the linear operator $DG(x)$ for a particular $x \in X$. This method is different from Image in one important aspect: the Deriv method dynamically creates (allocates) the output, which is an instance of HCL_LinearOp, and returns a pointer to the new object. Thus, if x is an instance of XVector with a meaningful value, and L is an HCL_LinearOp *pointer* variable, then the command

```
L = G.Deriv( x );
```

assigns to L the address of the object implementing $DG(x)$. Note that the program unit invoking G.Deriv(x) is responsible for releasing the dynamically allocated memory pointed to by L. The command

```
HCL_delete( L );
```

should be used for this purpose.

The SecondDeriv method allows the computation of $D^2G(x)$, and works in a manner analogous to that of Deriv. Mathematically $D^2G(x)$ is a bilinear operator, and so

```
B = G.SecondDeriv( x );
```

assigns to the pointer variable B the address of an instance of HCL_BiLinearOp. We describe bilinear operators and second derivatives below in Section 2.3.

It would be well at this point to explain a significant aspect of the design of HCL: classes tend to have "fat interfaces." For example, HCL_Op has methods to implement $G(x)$, $DG(x)$, and $D^2G(x)$, even though the class may be used to represent an operator 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_Op had (in pre-version 1.0 HCL) only the Image method. The Deriv method was present in the derived class HCL_OpDeriv, while the SecondDeriv method appeared in HCL_OpDeriv2. 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, with fat interfaces, means 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_EvaluateOp`, which is referred to as an *evaluation object*. This class represents the triple $(G(x), DG(x), D^2G(x))$; that is, it represents the realization of the operator 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 G 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 $G(x)$, $DG(x)$, and $D^2G(x)$. For instance,

```
e = G.Evaluate( x );  // e is an HCL_EvaluateOp pointer
e->Image( y );
```

is equivalent to

```
G.Image( x,y );
```

(The difference is that, by using the evaluation object, intermediate calculations done while computing $G(x)$ can be stored in the evaluation object pointed to by `e`, and they are therefore available if, later in the code, $DG(x)$ is requested.)

2.1.2 `Domain`, `Range`, and `MaxStep`

Recall that our generic example is an operator $G : X \to Y$, where the Hilbert spaces X and Y are represented by instances of the classes `XSpace` and `YSpace`, respectively. The command `G.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() != G.Domain() )
   // error condition
```

Similarly, `G.Range()` returns a reference to an instance of `YSpace`. We remark that the use of the word "range" is an abuse of terminology, since, by definition, the range of G is the set

$$\{G(x)|x \in X\},$$

and this equals Y only if the operator G is surjective. However, there is no common term for the space Y (some authors use the word "co-domain," but this does not appear to be standard), and we believe that the use of the word range will engender the least confusion.

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

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

computes the largest scalar α such that $x + \alpha z$ lies in the domain of G. 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 G.

2.1.3 Debugging methods

Four methods allow the implementor of an operator class to test the consistency of the evaluation methods. The method `CheckDeriv` allows the implementor to check whether

$$\frac{G(x + h\delta x) - G(x - h\delta x)}{2h} = DG(x)\delta x + O(h^2),$$

as ought to hold for a sufficiently smooth operator G. It does so by simply computing

$$\|\frac{G(x + h\delta x) - G(x - h\delta x)}{2h} - DG(x)\delta x\|/\|DG(x)\delta x\|$$

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.

Similarly, `CheckSecondDeriv` provides data for testing the correctness of $D^2G(x)$. There are also methods

`CheckDerivAdj`, `CheckSecondDerivAdj`.

These combine a call to `CheckDeriv` or `CheckSecondDeriv` with a call to the `HCL_LinearOp` or `HCL_BiLinearOp` method `CheckAdj`. For example, if `L` is an instance of `HCL_LinearOp`, then `L.CheckAdj()` compares

$$(Lx, y) \text{ and } (x, L^*y)$$

for random vectors x and y to determine whether the implemented code is consistent.

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

```
G.Write( str );
```

sends a description of the operator `G` 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 an operator using the protected image methods

Having described the public interface of HCL_Op, we now give an example of the implementation of a derived class. Our example will be the operator F described in Section 1.3.

We first note that the four testing methods, CheckDeriv, CheckSecondDeriv, etc., are all implemented in the base class, and need not be overridden. Three methods, Domain, Range, and Write, are pure virtual, meaning that they are not implemented in the base class, and *must* be overridden in a derived class. We begin our discussion with these methods.

Before defining the operator class, we must choose vector space classes for the spaces of the problem variables. Recall that $F : \mathcal{P} \to \mathcal{U}$, where $\mathcal{P} \subset \mathcal{A}$ and $\mathcal{A} \simeq \mathrm{R}^M, \mathcal{U} \simeq \mathrm{R}^N$. Accordingly, we will represent both $\mathcal{A}$ and $\mathcal{U}$ by instances of HCL_RnSpace, and $a \in \mathcal{A}$, $u \in \mathcal{U}$ by instances 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 can now define the private data section of the class, which must contain the domain and range (recall that the Domain and Range methods return *references* to the domain and range spaces, so the object must save these spaces), as well as the mesh information that will be required for various computations. Here is the beginning of the class definition, including the private data section. Note that we have named the operator class EllipticSolOp.

```
class EllipticSolOp_d: public HCL_Op_d
{
   private:
      HCL_RnSpace_d * dom;
      HCL_RnSpace_d * rng;
      int M,N,L;
      double * NodeList;
      int * NodePtrs;
      int * ElList;
      double amin;
```

Next we need a constructor and destructor for the class. For simplicity, we will assume that the mesh information is read from a file; therefore, the only input to the constructor will be a file name. The constructor reads a_{min}, the dimensions M, N, and L from the file, allocates the arrays NodeList, NodePtrs, and ElList, and then fills these arrays with information read from the file. It then creates instances of HCL_RnSpace to act as the domain and range spaces, and assigns their addresses to the private variables dom and rng. The destructor deletes all of the memory allocated by the constructor; note that HCL objects are deleted with the special function HCL_delete, because HCL implements reference counting. Here is the code for the constructor and destructor; we omit the code reading the mesh information from the file, as it adds nothing to this discussion.

```
EllipticSolOp_d::EllipticSolOp_d( char * fname )
{
   ifstream meshfile( fname );
   meshfile >> amin >> M >> N >> L;

   NodeList = new double[4*M];
   NodePtrs = new int[N];
   ElList = new int[3*L];

   // Input commands omitted here

   dom = new HCL_RnSpace_d( M );
   rng = new HCL_RnSpace_d( N );
}

EllipticSolOp_d::~EllipticSolOp_d()
{
   HCL_delete( rng );
   HCL_delete( dom );
   delete[] ElList;
   delete[] NodePtrs;
   delete[] NodeList;
}
```

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

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

HCL_VectorSpace_d & EllipticSolOp_d::Range() const
{
   return *rng;
}
```

The `Write` method is also simple to implement:

```
ostream & EllipticSolOp_d::Write( ostream & str ) const
{
   str << "EllipticSolOp_d with domain" << endl;
   Domain().Write( str );
   str << "and range" << endl;
```

```
   return Range().Write( str );
}
```

The last method, other than the evaluation methods, is **MaxStep**. Recall that we require $a(x,y) \geq a_{min}$ for all $(x,y) \in \Omega$; since the coefficient a is piecewise linear, we just require that the nodal values satisfy

$$\alpha_i \geq a_{min},\ i = 1, 2, \ldots, M.$$

The constant a_{min} is read from the mesh file by the constructor and stored in the private variable `amin`. Here is the code for the `MaxStep` method:

```
double EllipticSolOp_d::MaxStep( const HCL_Vector_d & x,
                                 const HCL_Vector_d &dir) const
{
   if( x.Space() != Domain() )
   {
      cerr << "Error in EllipticSolOp_d::MaxStep: input point "
              "is not in domain" << endl;
      cerr << "Space of input point" << endl;
      x.Space().Write( cerr );
      cerr << "Domain" << endl;
      Domain().Write( cerr );
      exit(1);
   }
   if( dir.Space() != Domain() )
   {
      cerr << "Error in EllipticSolOp_d::MaxStep: input direction "
              "is not in domain" << endl;
      cerr << "Space of input direction" << endl;
      dir.Space().Write( cerr );
      cerr << "Domain" << endl;
      Domain().Write( cerr );
      exit(1);
   }
   HCL_RnVector_d & a = (HCL_RnVector_d&)x;
   HCL_RnVector_d & da = (HCL_RnVector_d&)dir;

   HCL_FPArch_d arch;
   double alpha = arch.large;
   for( int i=1;i<=M;i++ )
      if( a(i) < amin )
         return -1.0;
      else if( da(i) < 0.0 )
         alpha = min( alpha,(amin-a(i))/da(i) );
```

```
    return alpha;
}
```

Now we turn to the evaluation methods:

`Image`, `Deriv`, `SecondDeriv`, `Evaluate`.

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

- the `HCL_EvaluateOp` "evaluation object";
- the `HCL_LinearOp` object representing the derivative $DF(a)$;
- the `HCL_BiLinearOp` object representing the second derivative $D^2F(a)$.

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_OpDefaultEval` to represent the evaluation object;
- `HCL_OpDefaultDeriv` to represent the first derivative;
- `HCL_OpDefaultSecondDeriv` to represent the second derivative.

These objects are automatically created by the `Evaluate`, `Deriv` and `SecondDeriv` methods (actually, to be precise, the `Deriv` and `SecondDeriv` methods work by creating an evaluation object by calling `Evaluate`, and then calling the corresponding derivative method from the evaluation object). However, in order for these default objects to work, the class implementor must implement (override) the following protected virtual functions:

- `Image1`,
- `DerivImage`,
- `DerivAdjImage`,
- `DerivInvImage`,
- `DerivInvAdjImage`,
- `SecondDerivImage`,
- `SecondDerivPartialAdjImage`.

Note, however, as explained earlier, those methods which are not relevant or not needed for a particular operator can be implemented as error messages; indeed, *all* of the above methods are implemented in the base class as errors, so those that are not needed for a particular concrete class can just be ignored. For example, often $DF(x)^{-1}$ either does not exist or is not needed; in that case, the methods `DerivInvImage` and `DerivInvAdjImage` can be ignored. Similarly,

`G.Image1( x,y );`	$y \leftarrow G(x)$
`G.DerivImage( x,dx,dy );`	$\delta y \leftarrow DG(x)\delta x$
`G.DerivAdjImage( x,dy,dx );`	$\delta x \leftarrow DG(x)^{*}\delta y$
`G.DerivInvImage( x,dy,dx );`	$\delta x \leftarrow DG(x)^{-1}\delta y$
`G.DerivInvAdjImage( x,dx,dy );`	$\delta y \leftarrow DG(x)^{-*}\delta x$
`G.SecondDerivImage( x,dx1,dx2,dy );`	$\delta y \leftarrow D^2G(x)(\delta x_1, \delta x_2)$
`G.SecondDerivPartialAdjImage( x,dx1,dy,dx2 );`	$\delta x_2 \leftarrow (D^2G(x)(\delta x_1, \cdot))^{*}dy$

Table 4: `HCL_Op` protected image methods (usage, effect).

if $D^2F(x)$ will not be used, then `SecondDerivImage` and `SecondDerivPartialAdjImage` can be ignored.

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

The protected image functions are summarized in Table 4. We now present the code for the methods corresponding to F and DF for our model problem; for a discussion of second derivatives and the code for the corresponding methods, see Section 2.3. Here is the code for the `Image1` method:

```
void EllipticSolOp_d::Image1( const HCL_Vector_d & x,
                                    HCL_Vector_d & y ) const
{
   if( x.Space() != Domain() )
   {
      cerr << "Error in EllipticSolOp_d::Image1: input vector is not "
              "in domain" << endl;
      cerr << "Space of input vector" << endl;
      x.Space().Write( cerr );
      cerr << "Domain" << endl;
      Domain().Write( cerr );
      exit(1);
   }
   if( MaxStep( x,x ) < 0.0 )
   {
      cerr << "Error in EllipticSolOp_d::Image1: input point "
              "is not in domain" << endl
           << "(though it belongs to the correct vector space)"
           << endl;
      exit(1);
   }
   if( y.Space() != Range() )
```

```
    {
       cerr << "Error in EllipticSolOp_d::Image1: output vector is not "
               "in range" << endl;
       cerr << "Space of output vector" << endl;
       y.Space().Write( cerr );
       cerr << "Range" << endl;
       Range().Write( cerr );
       exit(1);
    }
    HCL_RnVector_d & a = (HCL_RnVector_d&)x;
    HCL_RnVector_d & u = (HCL_RnVector_d&)y;

    SymSparseRCSMatrix_d K( N,N,5*N );
    HCL_RnVector_d b( N );
    F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                       a.Data(),K.rowstart,K.column,K.data,K.nNonZero,
                       K.nAlloc,b.Data() );
    HCL_SymSparseRCSMatOp_d Ka( &K,0,rng );
    Ka.InvImage( b,u );
}
```

In the `Image1` method, we use two classes:

- `SymSparseRCSMatrix`,
- `HCL_SymSparseRCSMatOp`.

The first implements the compressed row storage sparse matrix, while the second take such a sparse matrix and creates the linear operator represented by it. Note the use of the `InvImage` method from the `HCL_SymSparseRCSMatOp` class. This method results in a call to a Yale SparsePack routine, which is linked into the `SymSparseRCSMatrix` class. (The reason that there are two classes involved is that `SymSparseRCSMatrix` is not a part of HCL proper; it is part of a stand-alone collection of matrix classes. The class `HCL_SymSparseRCSMatOp` is an HCL "tool" class that is implemented using the sparse matrix class.)

Here is the code for the `DerivImage` and `DerivAdjImage` methods. For the sake of brevity, we omit the error-checking at the beginning of each method.

```
void EllipticSolOp_d::DerivImage( const HCL_Vector_d & x,
                                  const HCL_Vector_d & dx,
                                        HCL_Vector_d & dy ) const
{
   // Error-checking code omitted.

   HCL_RnVector_d & a = (HCL_RnVector_d&)x;
   HCL_RnVector_d & da = (HCL_RnVector_d&)dx;
```

```
   HCL_RnVector_d & du = (HCL_RnVector_d&)dy;

   SymSparseRCSMatrix_d K( N,N,5*N );
   HCL_RnVector_d b( N );
   F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      a.Data(),K.rowstart,K.column,K.data,K.nNonZero,
                      K.nAlloc,b.Data() );
   HCL_SymSparseRCSMatOp_d Ka( &K,0,rng );
   HCL_RnVector_d u( N );
   Ka.InvImage( b,u );

   SymSparseRCSMatrix_d K1( N,N,5*N );
   HCL_RnVector_d b1( N );
   F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      da.Data(),K1.rowstart,K1.column,K1.data,
                      K1.nNonZero,K1.nAlloc,b1.Data() );
   HCL_SymSparseRCSMatOp_d Kda( &K1,0,rng );

   HCL_RnVector_d tmp( N );
   Kda.Image( u,tmp );
   Ka.InvImage( tmp,du );
   du.Neg();
}

void EllipticSolOp_d::DerivAdjImage( const HCL_Vector_d & x,
                                     const HCL_Vector_d & dy,
                                           HCL_Vector_d & dx ) const
{
   // Error-checking code omitted.

   HCL_RnVector_d & a = (HCL_RnVector_d&)x;
   HCL_RnVector_d & da = (HCL_RnVector_d&)dx;
   HCL_RnVector_d & du = (HCL_RnVector_d&)dy;

   SymSparseRCSMatrix_d K( N,N,5*N );
   HCL_RnVector_d b( N );
   F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      a.Data(),K.rowstart,K.column,K.data,K.nNonZero,
                      K.nAlloc,b.Data() );
   HCL_SymSparseRCSMatOp_d Ka( &K,0,rng );
   HCL_RnVector_d u( N );
   Ka.InvImage( b,u );

   SparseRCSMatrix_d B( N,M,5*M );
```

```
    F77NAME(assemble2)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                        u.Data(),B.rowstart,B.column,B.data,B.nNonZero,
                        B.nAlloc );
    HCL_SparseRCSMatOp_d Bu( &B,0,dom,rng );

    HCL_RnVector_d tmp( N );
    Ka.InvImage( du,tmp );
    Bu.AdjImage( tmp,da );
    da.Neg();
}
```

Since $DF(a)$ is not invertible (and, in any case, would not be used), the methods

`DerivInvImage` and `DerivInvAdjImage`

are not overridden, so any call to these methods will result in an error.

Hopefully the above code for the image methods is understandable in terms of the formulas derived in Section 1.3. By way of summary, we now describe how each public evaluation method of the class `EllipticSolOp` functions.

- `Evaluate` (implemented in `HCL_Op`) Passes the input vector a and the address of the operator G to the constructor for for `HCL_OpDefaultEval`, and returns the address of this newly created evaluation object.
- `Image` (implemented in `HCL_Op`) Creates an evaluation object using `Evaluate` and calls the `Image` method of the evaluation object. The `HCL_OpDefaultEval Image` method then calls the `EllipticSolOp` protected `Image1` 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 operators 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 be coded into the base class; the implementor of the `EllipticSolOp` need only code the image methods, that is, the actual computations.)
- `Deriv` (implemented in `HCL_Op`) Creates an evaluation object using `Evaluate` and calls the `Deriv` method of the evaluation object. The `HCL_OpDefaultEval Deriv` method then calls the constructor of `HCL_OpDefaultDeriv` and returns the address of the result, which is in turn the return value of the original invocation of `Deriv`. Now consider the following code fragment:

  ```
  HCL_LinearOp_d * L = F.Deriv( a );
  L->Image( da,du );
  ```

  The pointer `L` points to the instance of `HCL_OpDefaultDeriv`, and `L->Image( da,du )` invokes the `HCL_OpDefaultDeriv Image` method, which calls the `EllipticSolOp`

DerivImage method. Once again, all of these contortions have already been coded in the HCL classes, and the implementor's only contribution is the code for DerivImage.

- SecondDeriv (implemented in HCL_Op) This method is exactly analogous to Deriv; it results in a pointer to a dynamically allocated instance of HCL_OpDefaultSecondDeriv. A call to the second derivative's Image method will result in a call to the EllipticSolOp SecondDerivImage method.

The careful reader will notice gross inefficiencies in the above image methods. An optimization algorithm will typically compute $J(a)$ for the current estimate a of the solution, and then may or may not evaluate $\nabla J(a)$. The computation of $J(a)$ requires that the finite element matrix $K(a)$ and the load vector b be assembled and that $u = K(a)^{-1}b$ be computed. The computation of the gradient also needs u. If the memory requirements are not excessive (as they certainly are not for a 2D problem), it would be much more efficient to save u in case it is needed again. Gaining this efficiency requires that we implement a special evaluation class for this problem, rather than relying on HCL_OpDefaultEval. This will be addressed in Section 4.

For some optimization algorithms, there will also be an efficiency gain in avoiding

HCL_OpDefaultDeriv.

For example, a Gauss-Newton method using conjugate gradients to approximate the Gauss-Newton step will need to repeatedly evaluate $DF(a)\delta a$ and $DF(a)^*\delta u$. Since, for this problem, the derivative $DF(a)$ is the composition of several simpler linear operators, it is possible to use HCL_CompLinearOp, an HCL tool class that implements a composition of linear operators, in place of HCL_OpDefaultDeriv. This variation on the implementation of F is explained in Section 3.

Before we discuss the more complicated mechanisms that allow more efficient code, we explain how second derivatives are represented and implemented in HCL.

2.3 Second derivatives and bilinear operators

Let us return to the consideration of a generic operator $G : X \to Y$. The second derivative of G (assuming G is sufficiently smooth) is a symmetric, bilinear operator operator $DG(x) : X \times X \to Y$. When an operator G appears in an optimization algorithm, it is often in the form

$$(G(x), \lambda)$$

(in a Lagrangian) or

$$(G(x), G(x))$$

(in a penalty term) or

$$\|G(x) - y^{obs}\|^2$$

(in an objective function). Note that if $f(x) = (G(x), \lambda)$, then

$$Df(x)\delta x = (DG(x)\delta x, \lambda),$$

and

$$D^2 f(x)(\delta x, \delta x) = (D^2 G(x)(\delta x, \delta x), \lambda).$$

It follows that $\nabla^2 f(x)$, which is the quantity needed in an optimization algorithm, is given by

$$\nabla^2 f(x)\delta x = (D^2 G(x)(\delta x, \cdot))^* \lambda,$$

where $D^2 f(\delta x, \cdot)$ represents the linear operator defined by

$$\delta y \mapsto D^2 f(x)(\delta x, \delta y)$$

(δx fixed). An optimization algorithm using second derivatives will need this linear operator and its adjoint. (The same linear operator arises if G appears in a penalty term or a misfit function.) These considerations serve as motivation for our design of the class `HCL_BiLinearOp`.

2.3.1 The class `HCL_BiLinearOp`

Let $B : X \times Y \to Z$ be a bilinear operator, where X, Y, and Z are Hilbert spaces. Such an operator can be implemented in a class derived from `HCL_BiLinearOp`. We now describe the public methods of this class.

- `Domain1`, `Domain2`, `Range` These methods return references to the `HCL_VectorSpace`s representing X, Y, and Z, respectively.
- `Image` The command

```
B.Image( x,y,z );
```

  assigns to `z` the value of $B(x, y)$.

- `Op1`, $\widehat{\mathrm{O}}$p2, `Op3` These methods create the linear operators

$$L_1 : y \mapsto B(x, y) \ (x \text{ fixed}),$$

$$L_2 : x \mapsto B(x, y) \ (y \text{ fixed}),$$

  and

$$L_3 : x \mapsto B(x, \cdot)^* z \ (z \text{ fixed}),$$

  respectively. Note that the operator

$$y \mapsto B(\cdot, y)^* z \ (z \text{ fixed}),$$

  is the adjoint of L_3:

$$\begin{aligned} (L_3 x, y)_Y &= (B(x, \cdot)^* z, y) \\ &= (z, B(x, y))_Z \\ &= (B(x, y), z)_Z \\ &= (x, B(\cdot, y)^* z)_X. \end{aligned}$$

  The three methods `Op1`, `Op2`, and `Op3` each creates a new instance of `HCL_LinearOp` and returns its address. Therefore, the typical command is

```
HCL_LinearOp * L = B.Op1( x );

// The operator pointed to by L is used here.

HCL_delete( L );
```

- `CheckAdj` This is a debugging method; it checks that the "adjointness" relationship is satisfied for the operators L_1, L_2, and L_3. It does so by creating the operators in turn (the fixed vector in each case is given a random value) and then calling the `HCL_LinearOp` `CheckAdj` method.

- `Write` Like all HCL classes, this class has a `Write` method, which outputs a description of the object.

Note that since `Op1`, `Op2`, `Op3` each return (a pointer to) a newly created object, there must be classes defined to represent each object. The objects in question represent the linear operators L_1, L_2, and L_3, respectively, and therefore the classes must be derived from `HCL_LinearOp`. In certain cases the implementor of the bilinear operator class may wish to define one or more new classes, and in other cases an existing HCL class, such as a matrix operator class, may be most appropriate. In the generic case, however, the appropriate image and adjoint image methods are naturally coded in subroutines (methods), and the only purpose of the classes implementing L_1, L_2, and L_3 is to allow these methods to be called. Moreover, only three calculations are required to implement L_1, L_2, L_3 and their adjoints:

$$B(x,y),\ B(x,\cdot)^*z,\ B(\cdot,y)^*z.$$

Therefore, we created a default class to represent any of L_1, L_2, and L_3; the class is called

`HCL_RestrictedBiLinearOp`.

The methods `Op1`, `Op2`, and `Op3` are implemented in `HCL_BiLinearOp` to create instances of the restricted bilinear operator class; in order for this mechanism to work, the implementor of the bilinear operator class representing B must override the protected virtual function `PartialAdjImage`, as well as the public virtual function `Image`. The calling sequence for `PartialAdjImage` is

```
B.PartialAdjImage( i,u,z,w );
```

This command must implement $w \leftarrow B(u,\cdot)^*z$ if `i` is 1 and $w \leftarrow B(\cdot,u)^*z$ if `i` is 2.

The alternative to this default mechanism is to override the virtual functions `Op1`, `Op2`, `Op3` in the derived class, and to not override `PartialAdjImage` (which is implemented as an error in the base class). In this case, the implementor of the bilinear operator class must choose or define classes to represent L_1, L_2, and L_3. In any case, the implementor must implement the virtual functions `Domain1`, `Domain2`, `Range`, `Image`, and `Write`.

2.3.2 Second derivatives

The `SecondDeriv` method of the `HCL_Op` class must create a bilinear operator to represent $D^G(x)$. As with the first derivative, we have created a default class,

`HCL_OpDefaultSecondDeriv`,

to represent this object. Using the default class requires that the implementor of the operator class override the protected virtual functions

`SecondDerivImage` and `SecondDerivPartialAdjImage`.

The first method implements $\delta y \leftarrow D^2G(x)(\delta x_1, \delta x_2)$; the calling sequence is

```
G.SecondDerivImage( x,dx1,dx2,dy );
```

The second method implements

$$dx_2 \leftarrow (D^2G(x)(\delta x_1, \cdot))^* \delta y,$$

and is invoked as follows:

```
G.SecondDerivPartialAdjImage( x,dx1,dy,dx2 );
```

Note that $D^2G(x)$ is a symmetric bilinear operator, and so there is no need to implement

$$dx_1 \leftarrow (D^2G(x)(\cdot, \delta x_2))^* \delta y.$$

2.3.3 Second derivatives for the model problem

We now return to our concrete example and show how we implement second derivatives. Recall from our discussion in Section 1.3 that

$$D^2F(a)(\delta a, \delta b) = K(a)^{-1}(K(\delta a)K(a)^{-1}K(\delta b)u + K(\delta b)K(a)^{-1}K(\delta a)u).$$

We also need a formula for $(D^2F(a)(\delta a, \cdot))^* dv$. We have

$$\begin{aligned}
(D^2F(a)(\delta a, \delta b), \delta v) &= (K(a)^{-1}(K(\delta a)K(a)^{-1}K(\delta b)u + K(\delta b)K(a)^{-1}K(\delta a)u), \delta v) \\
&= (K(\delta a)K(a)^{-1}K(\delta b)u + K(\delta b)K(a)^{-1}K(\delta a)u, K(a)^{-1}\delta v) \\
&= (K(\delta b)u, K(a)^{-1}K(\delta a)K(a)^{-1}\delta v) + (B(\delta u)\delta b, K(a)^{-1}\delta v) \\
&= (B(u)\delta b, K(a)^{-1}K(\delta a)K(a)^{-1}\delta v) + (B(\delta u)\delta b, K(a)^{-1}\delta v) \\
&= (\delta b, B(u)^*K(a)^{-1}K(\delta a)K(a)^{-1}\delta v) + (\delta b, B(\delta u)^*K(a)^{-1}\delta v),
\end{aligned}$$

where we write δu for $DF(a)\delta a$. We therefore obtain

$$(D^2F(a)(\delta a, \cdot))^*\delta v = B(u)^*K(a)^{-1}K(\delta b)K(a)^{-1}\delta v + B(\delta u)^*K(a)^{-1}\delta v,\ \delta u = K(a)^{-1}K(\delta a)u.$$

Here is the code for the `SecondDerivImage` and `SecondDerivPartialAdjImage` methods. Once again we omit the error-checking at the beginning of the methods.

```
void EllipticSolOp_d::SecondDerivImage( const HCL_Vector_d & x,
                                        const HCL_Vector_d & dx1,
                                        const HCL_Vector_d & dx2,
                                              HCL_Vector_d & dy ) const
{
   // Error-checking omitted

   HCL_RnVector_d & a = (HCL_RnVector_d&)x;
   HCL_RnVector_d & da = (HCL_RnVector_d&)dx1;
   HCL_RnVector_d & db = (HCL_RnVector_d&)dx2;
   HCL_RnVector_d & du = (HCL_RnVector_d&)dy;

   SymSparseRCSMatrix_d * K1 = new SymSparseRCSMatrix_d( N,N,5*N );
   HCL_RnVector_d b1( N );
   F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      da.Data(),K1->rowstart,K1->column,
                      K1->data,K1->nNonZero,K1->nAlloc,b1.Data() );
   HCL_SymSparseRCSMatOp_d * Kda =
          new HCL_SymSparseRCSMatOp_d( K1,0,rng );

   HCL_RnVector_d du1( N ),tmp1( N );
   DerivImage( a,db,du1 );
   Kda->Image( du1,tmp1 );
   HCL_delete( Kda );
   delete K1;

   K1 = new SymSparseRCSMatrix_d( N,N,5*N );
   F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      db.Data(),K1->rowstart,K1->column,
                      K1->data,K1->nNonZero,K1->nAlloc,b1.Data() );
   HCL_SymSparseRCSMatOp_d * Kdb =
          new HCL_SymSparseRCSMatOp_d( K1,0,rng );

   HCL_RnVector_d du2( N ),tmp2( N );
   DerivImage( a,da,du2 );
   Kda->Image( du2,tmp2 );
   HCL_delete( Kda );
   delete K1;

   tmp1.Add( tmp2 );

   SymSparseRCSMatrix_d * K = new SymSparseRCSMatrix_d( N,N,5*N );
   HCL_RnVector_d b( N );
```

```
   F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      a.Data(),K->rowstart,K->column,
                      K->data,K->nNonZero,K->nAlloc,b.Data() );
   HCL_SymSparseRCSMatOp_d * Ka =
          new HCL_SymSparseRCSMatOp_d( K,0,rng );
   Ka->InvImage( tmp1,du );
   HCL_delete( Ka );
   delete K;

   du.Neg();
}

void EllipticSolOp_d::SecondDerivPartialAdjImage(const HCL_Vector_d & x,
                                               const HCL_Vector_d & dx1,
                                               const HCL_Vector_d & dy,
                                               HCL_Vector_d & dx2 ) const
{
   // Error-checking omitted

   HCL_RnVector_d & a = (HCL_RnVector_d&)x;
   HCL_RnVector_d & da = (HCL_RnVector_d&)dx1;
   HCL_RnVector_d & db = (HCL_RnVector_d&)dx2;
   HCL_RnVector_d & dv = (HCL_RnVector_d&)dy;

   HCL_RnVector_d du( N ),tmp1( N );

   SymSparseRCSMatrix_d * K = new SymSparseRCSMatrix_d( N,N,5*N );
   HCL_RnVector_d b( N );
   F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      a.Data(),K->rowstart,K->column,
                      K->data,K->nNonZero,K->nAlloc,b.Data() );
   HCL_SymSparseRCSMatOp_d * Ka =
          new HCL_SymSparseRCSMatOp_d( K,0,rng );
   Ka->InvImage( dv,tmp1 );
   HCL_delete( Ka );
   delete K;

   DerivImage( a,da,du );

   SymSparseRCSMatrix_d * K1 = new SymSparseRCSMatrix_d( N,N,5*N );
   F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      da.Data(),K1->rowstart,K1->column,K1->data,
                      K1->nNonZero,K1->nAlloc,b.Data() );
   HCL_SymSparseRCSMatOp_d * Kda =
```

```
            new HCL_SymSparseRCSMatOp_d( K1,0,rng );
    HCL_RnVector_d tmp2( N );
    Kda->Image( tmp1,tmp2 );
    HCL_delete( Kda );
    delete K1;

    DerivAdjImage( a,tmp2,db );

    SparseRCSMatrix_d B( N,M,5*M );
    F77NAME(assemble2)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                        du.Data(),B.rowstart,B.column,B.data,B.nNonZero,
                        B.nAlloc );
    HCL_SparseRCSMatOp_d Bdu( &B,0,dom,rng );
    HCL_RnVector_d tmp4( M );
    Bdu.AdjImage( tmp1,tmp4 );
    db.Add( tmp4 );
    db.Neg();
```

Note, again, that the above code has various inefficiencies that are due to the fact that intermediate quantities are not saved for re-use.

3 Using existing HCL classes to represent a derivative

We now begin to address the issue of efficiency in the coding of operators. The implementation of our model operator F, given in the previous section, involved more-or-less the minimum amount of code imaginable, so it was efficient as far a programming time is concerned. For many problems, the runtime efficiency of code implemented in this fashion may be perfectly acceptable (for instance, finite-difference simulation may leave little opportunity for re-using intermediate computations). However, for our model problem, the runtime efficiency of the above code is very poor.

A primary opportunity for improving the runtime efficiency of our sample code lies in the evaluation of $DF(a)\delta a$. Recall that

$$DF(a)\delta a = -K(a)^{-1}K(\delta a)u = -K(a)^{-1}B(u)\delta a,$$

where $u = F(a)$. As our code is currently written, each time we evaluate $DF(a)\delta a$, we must:

1. Assemble the stiffness matrix $K(a)$ and the load vector b.
2. Compute $u = K((a)^{-1}b$.
3. Assemble the stiffness matrix $K(\delta a)$.
4. Multiply $K(\delta a)u$.
5. Compute $\delta u = -K(a)^{-1}(K(\delta a)u)$.

A better implementation would compute $K(a)$, u, and $B(u)$ once and for all, and then use them until $DF(a)$ is no longer needed for that particular value of a.

Using the current model, this is not possible, since we have a single class `EllipticSolOp`, which must represent the operator F rather than its realization at a particular a. We cannot store the intermediate quantities in the operator class since there is no guarantee that two successive calls to, say, `DerivImage` refer to the same value of a. Moreover, although the `HCL_OpDefaultDeriv` is a separate class, it just refers back to `EllipticSolOp` for all of the image methods.

We can avoid recomputing the intermediate quantities only by representing the derivative by a completely independent class. In this case, there is an obvious choice. The operator $DF(a)$ is a composition of three linear operators ($B(u)$, $K(a)^{-1}$, and the negative identity), and HCL has a tool class `HCL_CompLinearOp` representing a composition of an arbitrary number of linear operators. It is not the purpose of this report to explain how `HCL_CompLinearOp` works in detail; rather, we wish to show how this or some other independent class (including one written by the implementor for a particular application) can be used for the derivative.

The `Deriv` method in class `HCL_OpDefaultEval` is defined as follows:

```
HCL_LinearOp_d * HCL_OpDefaultEval_d::Deriv() const
{
   HCL_LinearOp_d * DFx = op->Deriv1( *x );
   if( DFx )
      return DFx;
   else
      return new HCL_OpDefaultDeriv_d( this );
}
```

Note that this method first calls the (protected) method `Deriv1` from the operator class. This method is implemented in `HCL_Op` to return the NULL pointer, so unless `Deriv1` is overridden, an instance of `HCL_OpDefaultDeriv` will be created. However, the implementor of the derived operator class can override `Deriv1` (instead of overriding `DerivImage`, `DerivAdjImage`, and then the linear operator created by `Deriv1` will be return by the `Deriv` method of `HCL_OpDefaultEval`.

Here is the code for the `Deriv1` method of class `EllipticSolOp1`, the new version of `EllipticSolOp`. We should emphasize that this corresponds to a different implementation of F than the one presented in Section 2. In this new implementation, although much of the code is the same, the following code is included, and the code for the methods `DerivImage` and `DerivAdjImage`, presented in the preceding section, is omitted.

```
HCL_LinearOp_d * EllipticSolOp1_d::Deriv1(
                          const HCL_Vector_d & x ) const
{
   if( x.Space() != Domain() )
   {
```

```
   cerr << "Error in EllipticSolOp1_d::Deriv1: input vector "
           "is not in domain" << endl;
   cerr << "Space of input vector" << endl;
   x.Space().Write( cerr );
   cerr << "Domain" << endl;
   Domain().Write( cerr );
   exit(1);
}
if( MaxStep( x,x ) < 0.0 )
{
   cerr << "Error in EllipticSolOp1_d::Deriv1: input point is not "
           "in domain " << endl
        << "(though it belongs to the correct vector space)"
        << endl;
   exit(1);
}
HCL_RnVector_d & a = (HCL_RnVector_d&)x;

SymSparseRCSMatrix_d * K = new SymSparseRCSMatrix_d( N,N,5*N );
HCL_RnVector_d b( N );
F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                   a.Data(),K->rowstart,K->column,
                   K->data,K->nNonZero,K->nAlloc,b.Data() );
HCL_SymSparseRCSMatOp_d * Ka =
       new HCL_SymSparseRCSMatOp_d( K,1,rng );
HCL_RnVector_d u( N );
Ka->InvImage( b,u );

SparseRCSMatrix_d * B = new SparseRCSMatrix_d( N,M,5*M );
F77NAME(assemble2)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                   u.Data(),B->rowstart,B->column,
                   B->data,B->nNonZero,B->nAlloc );
HCL_SparseRCSMatOp_d * Bu =
          new HCL_SparseRCSMatOp_d( B,1,dom,rng );

HCL_InvLinearOp_d * Kinv = new HCL_InvLinearOp_d( Ka );
HCL_delete( Ka );
HCL_ScaleLinearOp_d * Neg =
               new HCL_ScaleLinearOp_d( &(Kinv->Range()),-1.0 );

HCL_CompLinearOp_d * D =
        new HCL_CompLinearOp_d( &(Bu->Domain()),
                                &(Kinv->Range()),3 );
D->SetNext( Bu );
```

```
      HCL_delete( Bu );
      D->SetNext( Kinv );
      HCL_delete( Kinv );
      D->SetNext( Neg );
      HCL_delete( Neg );

      return D;
   }
```

We note that, since `DerivImage` and `DerivAdjImage` are not implemented in `EllipticSolOp1`, we made a few changes to the `SecondDerivImage` method to eliminate references to these methods (using the `SecondDeriv` method instead). However, we do not give the new code here since it is so similar to the old code.

We also note that, as expected, the second version of the code results in much faster (repeated) evaluations of $DF(a)\delta a$ (as might be needed in an iterative Gauss-Newton code, for instance). On a Sun SparcStation 10, the second version is faster than the first by a factor of approximately 36 (for a 10×10 mesh).

The same mechanism can be used to increase the efficiency of the computations of

$$D^2F(a)(\delta a, \delta b) \text{ and } (D^2F(a)(\delta a, \cdot))^*\delta u.$$

By overriding the protected `SecondDeriv1` method in a derived operator class (instead of overriding

`SecondDerivImage` and `SecondDerivPartialAdjImage`),

the implementor can provide a more efficient implementation of the second derivative. Since, at least in version 1.0, HCL does not contain any tool classes for building up bilinear operators, the implementor would have to define a new class derived from `HCL_BiLinearOp`. We present such a class,

`EllipticSolOpSecondDeriv`,

below. Defining this class is not difficult. The constructor of the class creates the various objects involved (the objects representing u, $K(a)$, and $B(u)$) and saves them as private data. The destructor deletes these objects. The image methods are then essentially the same as before, except that the objects just listed have already been computed. Finally, there is the "boilerplate" code: the methods returning the domains and range, as well as the `Write` method.

Here are the declaration and implementation for `EllipticSolOpSecondDeriv`. The `Write` method, unimplemented constructors, and all error-checking code is omitted in interests of brevity.

```
class EllipticSolOpSecondDeriv_d: public HCL_BiLinearOp_d
{
   private:
```

```
      HCL_RnSpace_d * dom;
      HCL_RnSpace_d * rng;
      int M,N,L;
      double * NodeList;
      int * NodePtrs;
      int * ElList;
      HCL_SymSparseRCSMatOp_d * Ka;
      HCL_SparseRCSMatOp_d * Bu;
      HCL_RnVector_d * u;
   protected:

      virtual void PartialAdjImage( int i,const HCL_Vector_d & xy,
                                    const HCL_Vector_d & z,
                                          HCL_Vector_d & yx ) const;
   public:

      EllipticSolOpSecondDeriv_d();
      EllipticSolOpSecondDeriv_d(
              const EllipticSolOpSecondDeriv_d & L );
      EllipticSolOpSecondDeriv_d( HCL_RnSpace_d * d,HCL_RnSpace_d * r,
                                  int MM,int NN,int LL,double * NL,
                                  int * NP,int * EL,
                                  const HCL_RnVector_d & a );
      virtual ~EllipticSolOpSecondDeriv_d();

      virtual HCL_VectorSpace_d & Domain1() const { return *dom; }
      virtual HCL_VectorSpace_d & Domain2() const { return *dom; }
      virtual HCL_VectorSpace_d & Range() const { return *rng; }

      virtual void Image( const HCL_Vector_d & x,const HCL_Vector_d & y,
                                HCL_Vector_d & z ) const;
      virtual ostream & Write( ostream & str ) const;
};

EllipticSolOpSecondDeriv_d::EllipticSolOpSecondDeriv_d(
                            HCL_RnSpace_d * d,HCL_RnSpace_d * r,
                            int MM,int NN,int LL,
                            double * NL,int * NP,int * EL,
                            const HCL_RnVector_d & aa )
{
   dom = d;d->IncCount();
   rng = r;r->IncCount();
   M = MM;
   N = NN;
```

```
   L = LL;
   NodeList = new double[4*M];
   int i;
   for( i=0;i<4*M;i++ ) NodeList[i] = NL[i];
   NodePtrs = new int[N];
   for( i=0;i<N;i++ ) NodePtrs[i] = NP[i];
   ElList = new int[3*L];
   for( i=0;i<3*L;i++ ) ElList[i] = EL[i];

   HCL_RnVector_d & a = (HCL_RnVector_d&)aa;
   SymSparseRCSMatrix_d * K = new SymSparseRCSMatrix_d( N,N,5*N );
   HCL_RnVector_d b( N );
   F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      a.Data(),K->rowstart,K->column,
                      K->data,K->nNonZero,K->nAlloc,b.Data() );
   Ka = new HCL_SymSparseRCSMatOp_d( K,1,rng );
   u = new HCL_RnVector_d( N );
   Ka->InvImage( b,*u );

   SparseRCSMatrix_d * B = new SparseRCSMatrix_d( N,M,5*M );
   F77NAME(assemble2)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      u->Data(),B->rowstart,B->column,
                      B->data,B->nNonZero,B->nAlloc );
   Bu = new HCL_SparseRCSMatOp_d( B,1,dom,rng );
}

EllipticSolOpSecondDeriv_d::~EllipticSolOpSecondDeriv_d()
{
   HCL_delete( Bu );
   HCL_delete( u );
   HCL_delete( Ka );
   delete[] ElList;
   delete[] NodePtrs;
   delete[] NodeList;
   HCL_delete( rng );
   HCL_delete( dom );
}

void EllipticSolOpSecondDeriv_d::Image( const HCL_Vector_d & x,
                                        const HCL_Vector_d & y,
                                              HCL_Vector_d & z ) const
{
   // Error-checking code omitted
   HCL_RnVector_d & da = (HCL_RnVector_d&)x;
```

```
   HCL_RnVector_d & db = (HCL_RnVector_d&)y;
   HCL_RnVector_d & du = (HCL_RnVector_d&)z;

   SymSparseRCSMatrix_d * K1 = new SymSparseRCSMatrix_d( N,N,5*N );
   HCL_RnVector_d b1( N );
   F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      da.Data(),K1->rowstart,K1->column,
                      K1->data,K1->nNonZero,K1->nAlloc,b1.Data() );
   HCL_SymSparseRCSMatOp_d * Kda =
          new HCL_SymSparseRCSMatOp_d( K1,1,rng );

   HCL_RnVector_d du1( N ),du2( N ),tmp1( N ),tmp2( N );
   Kda->Image( *u,tmp1 );
   Ka->InvImage( tmp1,du1 );

   K1 = new SymSparseRCSMatrix_d( N,N,5*N );
   F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      db.Data(),K1->rowstart,K1->column,
                      K1->data,K1->nNonZero,K1->nAlloc,b1.Data() );
   HCL_SymSparseRCSMatOp_d * Kdb =
          new HCL_SymSparseRCSMatOp_d( K1,1,rng );

   Kdb->Image( *u,tmp2 );
   Ka->InvImage( tmp2,du2 );

   Kda->Image( du2,tmp1 );
   Kdb->Image( du1,tmp2 );
   HCL_delete( Kdb );
   HCL_delete( Kda );

   tmp1.Add( tmp2 );

   Ka->InvImage( tmp1,du );
}

void EllipticSolOpSecondDeriv_d::PartialAdjImage( int i,
                                  const HCL_Vector_d & xy,
                                  const HCL_Vector_d & z,
                                        HCL_Vector_d & yx ) const
{
   // Error-checking code omitted.
   HCL_RnVector_d & da = (HCL_RnVector_d&)xy;
   HCL_RnVector_d & db = (HCL_RnVector_d&)yx;
   HCL_RnVector_d & dv = (HCL_RnVector_d&)z;
```

```
    HCL_RnVector_d du( N ),tmp1( N ),tmp2( N ),tmp3( M ),tmp4( N );

    Ka->InvImage( dv,tmp1 );

    SymSparseRCSMatrix_d * K1 = new SymSparseRCSMatrix_d( N,N,5*N );
    HCL_RnVector_d b( N );
    F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                       da.Data(),K1->rowstart,K1->column,
                       K1->data,K1->nNonZero,K1->nAlloc,b.Data() );
    HCL_SymSparseRCSMatOp_d * Kda =
           new HCL_SymSparseRCSMatOp_d( K1,1,rng );
    Kda->Image( tmp1,tmp2 );
    Kda->Image( *u,tmp4 );
    Ka->InvImage( tmp4,du );
    HCL_delete( Kda );

    Ka->InvImage( tmp2,tmp4 );
    Bu->AdjImage( tmp4,db );

    SparseRCSMatrix_d B( N,M,5*M );
    F77NAME(assemble2)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                       du.Data(),B.rowstart,B.column,B.data,B.nNonZero,
                       B.nAlloc );
    HCL_SparseRCSMatOp_d Bdu( &B,0,dom,rng );
    Bdu.AdjImage( tmp1,tmp3 );
    db.Add( tmp3 );
}
```

4 Defining an evaluation object for increased efficiency

The implementation of F presented in the preceding section represented a large increase in the performance of the code as it would typically be used in an optimization algorithm. However, these is one significant inefficiency remaining in `EllipticSolOp1`: the computations of $F(a)$, $DF(a)\delta a$, and $D^2F(a)(\delta a, \delta b)$ (and the various adjoints) repeat the computation of certain intermediate quantities, namely, the stiffness matrix $K(a)$, the solution $u = F(a)$, and the "partial adjoint" matrix $B(u)$. (Note that in the previous section, we eliminated the repeated computations of these quantities on successive evaluations of $DF(a)\delta a$ or $D^2F(a)(\delta a, \delta b)$, but we did not enable these quantities to be shared by, say, $DF(a)$ and $D^2F(a)$.)

To address this concern, we must define an evaluation object (that is, a class derived from `HCL_EvaluateOp`) specifically for this operator, rather than using `HCL_OpDefaultEval`. This method of implementing an operator puts all of the computations (the calculation of $F(a)$, $DF(a)$, and $D^2F(a)$) in the evaluation class, which will also hold, in its private data section, the intermediate computations mentioned above. Therefore, we do *not* override any of the protected methods of `HCL_Op`. Below is the definition of the new derived operator class, which we call `EllipticSolOp2`, along with the implementation of the `Evaluate` method, which is the only novel code in the implementation of the class. Note that this class does little more than store the domain, range, and mesh, and create evaluation objects.

```
class EllipticSolOp2_d: public HCL_Op_d
{
   private:
      HCL_RnSpace_d * dom;
      HCL_RnSpace_d * rng;
      int M,N,L;
      double * NodeList;
      int * NodePtrs;
      int * ElList;
      double amin;

   public:

      EllipticSolOp2_d();
      EllipticSolOp2_d( const EllipticSolOp2_d & );
      EllipticSolOp2_d( char * fname );
      ~EllipticSolOp2_d();

      virtual HCL_VectorSpace_d & Domain() const { return *dom; }
      virtual HCL_VectorSpace_d & Range() const { return *rng; }

      virtual double MaxStep( const HCL_Vector_d & x,
      const HCL_Vector_d &dir) const;

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

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

HCL_EvaluateOp_d * EllipticSolOp2_d::Evaluate(
                              const HCL_Vector_d & x ) const
{
   if( x.Space() != Domain() )
   {
      cerr << "Error in EllipticSolOp2_d::Evaluate: input point is not "
              "in domain" << endl;
      cerr << "Space of input point" << endl;
      x.Space().Write( cerr );
```

```
      cerr << "Domain" << endl;
      Domain().Write( cerr );
      exit(1);
   }
   if( MaxStep( x,x ) < 0.0 )
   {
      cerr << "Error in EllipticSolOp2_d::Evaluate: input point "
              "is not in domain " << endl
           << "(though it belongs to the correct vector space)"
           << endl;
      exit(1);
   }
   HCL_RnVector_d & a = (HCL_RnVector_d&)x;
   return new EvalEllipticSolOp2_d( dom,rng,M,N,L,NodeList,
                                    NodePtrs,ElList,a );
}
```

Next we exhibit the definition and implementation of the evaluation class

EvalEllipticSolOp2.

Note that there is a corresponding class SecondDeriv1 implementing the second derivative; this class is exactly the same as SecondDeriv, except that instead of having the intermediate quantities ($K(a)$, u, and $B(u)$) computed in the constructor of the second derivative object, they are simply passed from the evaluation object when it calls the constructor. Also, $u = F(a)$ and $K(a)$, which are needed in each of the evaluation methods, are computed in the constructor of the evaluation object, while $B(u)$, which is needed for Deriv and SecondDeriv but not for Image, is computed in whichever of Deriv and SecondDeriv is called first. Here is the definition and implementation of EvalEllipticSolOp2.

```
class EvalEllipticSolOp2_d: public HCL_EvaluateOp_d
{
   private:
      HCL_RnSpace_d * dom;
      HCL_RnSpace_d * rng;
      int M,N,L;
      double * NodeList;
      int * NodePtrs;
      int * ElList;
      double amin;
      HCL_RnVector_d * a;
      HCL_SymSparseRCSMatOp_d * Ka;
      HCL_SparseRCSMatOp_d * Bu;
      HCL_RnVector_d * u;
```

```
      void GetBu() const;

   public:

      EvalEllipticSolOp2_d();
      EvalEllipticSolOp2_d( const EvalEllipticSolOp2_d & );
      EvalEllipticSolOp2_d( HCL_RnSpace_d * d,HCL_RnSpace_d * r,
                            int MM,int NN,int LL,double * NL,
                            int * NP,int * EL,
                            const HCL_RnVector_d & a );
      virtual ~EvalEllipticSolOp2_d();

      virtual HCL_VectorSpace_d & Domain() const { return *dom;; }
      virtual HCL_VectorSpace_d & Range() const { return *rng; }

      virtual void Image( HCL_Vector_d & y ) const;
      virtual HCL_LinearOp_d * Deriv() const;
      virtual HCL_BiLinearOp_d * SecondDeriv() const;

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

EvalEllipticSolOp2_d::EvalEllipticSolOp2_d( HCL_RnSpace_d * d,
                                  HCL_RnSpace_d * r,
                                  int MM,int NN,int LL,double * NL,
                                  int * NP,int * EL,
                                  const HCL_RnVector_d & aa )
{
   dom = d;d->IncCount();
   rng = r;r->IncCount();
   M = MM;
   N = NN;
   L = LL;
   NodeList = new double[4*M];
   int i;
   for( i=0;i<4*M;i++ ) NodeList[i] = NL[i];
   NodePtrs = new int[N];
   for( i=0;i<N;i++ ) NodePtrs[i] = NP[i];
   ElList = new int[3*L];
   for( i=0;i<3*L;i++ ) ElList[i] = EL[i];

   HCL_RnVector_d & a = (HCL_RnVector_d&)aa;
   SymSparseRCSMatrix_d * K = new SymSparseRCSMatrix_d( N,N,5*N );
   HCL_RnVector_d b( N );
```

```
   F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      a.Data(),K->rowstart,K->column,
                      K->data,K->nNonZero,K->nAlloc,b.Data() );
   Ka = new HCL_SymSparseRCSMatOp_d( K,1,rng );
   u = new HCL_RnVector_d( N );
   Ka->InvImage( b,*u );
   Bu = NULL;
}

EvalEllipticSolOp2_d::~EvalEllipticSolOp2_d()
{
   HCL_delete( Bu );
   HCL_delete( u );
   HCL_delete( Ka );
   delete[] ElList;
   delete[] NodePtrs;
   delete[] NodeList;
   HCL_delete( rng );
   HCL_delete( dom );
}

void EvalEllipticSolOp2_d::GetBu() const
{
   EvalEllipticSolOp2_d * t = (EvalEllipticSolOp2_d*)this;
   SparseRCSMatrix_d * B = new SparseRCSMatrix_d( N,M,5*M );
   F77NAME(assemble2)( (int&)(t->M),(int&)(t->N),(int&)(t->L),
                      t->NodeList,t->NodePtrs,t->ElList,
                      (t->u)->Data(),B->rowstart,B->column,
                      B->data,B->nNonZero,B->nAlloc );
   t->Bu = new HCL_SparseRCSMatOp_d( B,1,dom,rng );
}

void EvalEllipticSolOp2_d::Image( HCL_Vector_d & y ) const
{
   if( y.Space() != Range() )
   {
      cerr << "Error in EllipticSolOp2_d::Image: output vector is not "
              "in range" << endl;
      cerr << "Space of output vector" << endl;
      y.Space().Write( cerr );
      cerr << "Range:"<< endl;
      Range().Write( cerr );
      exit(1);
   }
```

```
   y.Copy( *u );
}

HCL_LinearOp_d * EvalEllipticSolOp2_d::Deriv() const
{
   if( !Bu ) GetBu();

   HCL_InvLinearOp_d * Kinv = new HCL_InvLinearOp_d( Ka );
   HCL_ScaleLinearOp_d * Neg =
               new HCL_ScaleLinearOp_d( &(Kinv->Range()),-1.0 );

   HCL_CompLinearOp_d * D =
           new HCL_CompLinearOp_d( &(Bu->Domain()),
                                   &(Kinv->Range()),3 );
   D->SetNext( Bu );
   D->SetNext( Kinv );
   HCL_delete( Kinv );
   D->SetNext( Neg );
   HCL_delete( Neg );

   return D;
}

HCL_BiLinearOp_d * EvalEllipticSolOp2_d::SecondDeriv() const
{
   if( !Bu ) GetBu();
   return new EllipticSolOpSecondDeriv1_d( dom,rng,M,N,L,NodeList,
                                           NodePtrs,ElList,u,Ka,Bu );
}
```

4.1 The final alternate implementation

There is one more way to implement an operator in HCL, which is really a hybrid of the first and third implementations (explained in Sections 2 and 4). In the case that there is no particular reason to create distinct classes to represent $DF(a)$ and/or $D^2F(a)$, but when one needs to create a distinct evaluation class for efficiency reasons, one may proceed as follows. Instead of overriding the virtual functions `Deriv` and `SecondDeriv` of `HCL_EvaluateOp` (as we did in the previous section), the protected image methods `DerivImage`, `DerivAdjImage`, `SecondDerivImage`, and `SecondDerivPartialAdjImage` of `HCL_EvaluateOp` can be overridden. (These methods are implemented as errors in `HCL_EvaluateOp`.) The default implementations of `Deriv` and `SecondDeriv`, which are provided in `HCL_EvaluateOp`, then create instances of `HCL_OpDefaultDeriv` and `HCL_OpDefaultSecondDeriv`, respectively, which call the protected image methods in the evaluation class to perform all computations. This mechanism, using the default derivative classes and the protected image methods, works in almost exactly the same fashion as in did in the implementation presented in Section 2, except that the protected image methods are now found in the evaluation class.

It is also possible to treat the first and second derivatives differently. One can be implemented using the protected image methods, and the other using a distinct class, either an HCL tool class, or a user-defined class. This is the preferred method for our model operator F, since there is a natural way to implement $DF(a)$ using the HCL too class `HCL_CompLinearOp`, but there is no need to use a separate class to implement $D^2F(a)$. (In Section 3, we did define a separate class for $D^2F(a)$, but only so that we could store intermediate quantities rather than recompute them on each call to an image method. Now that we have introduced the evaluation object for this purpose, there is no reason not to use the protected image methods to implement $D^2F(a)$.)

The optimal implementation of the model operator F, then, consists of two classes, one derived from `HCL_Op`, which we call

`EllipticSolOp3`,

and one derived from `HCL_EvaluateOp`, which we call

`EvalEllipticSolOp3`.

The first derivative $DF(a)$ is implemented using `HCL_CompLinearOp`, as in Sections 3 and 4, and the second derivative $D^2F(a)$ is implemented using the protected image methods of `HCL_EvaluateOp` and `HCL_OpDefaultSecondDeriv`. For the sake of completeness, we give the complete definition and implementation here, even though most of the code is identical to that of earlier implementations.

```
class EvalEllipticSolOp3_d: public HCL_EvaluateOp_d
{
   private:
      HCL_RnSpace_d * dom;
      HCL_RnSpace_d * rng;
      int M,N,L;
      double * NodeList;
      int * NodePtrs;
      int * ElList;
      double amin;
      HCL_RnVector_d * a;
      HCL_SymSparseRCSMatOp_d * Ka;
      HCL_SparseRCSMatOp_d * Bu;
      HCL_RnVector_d * u;

      void GetBu() const;

   protected:
      virtual void SecondDerivImage( const HCL_Vector_d & dx1,
                                     const HCL_Vector_d & dx2,
```

```
                                          HCL_Vector_d & dy ) const;
      virtual void SecondDerivPartialAdjImage(
                         const HCL_Vector_d & dx1,
                         const HCL_Vector_d & dy,
                               HCL_Vector_d & dx2 ) const;
   public:

      EvalEllipticSolOp3_d();
      EvalEllipticSolOp3_d( const EvalEllipticSolOp3_d & );
      EvalEllipticSolOp3_d( HCL_RnSpace_d * d,HCL_RnSpace_d * r,
                            int MM,int NN,int LL,double * NL,
                            int * NP,int * EL,
                            const HCL_RnVector_d & a );
      virtual ~EvalEllipticSolOp3_d();

      virtual HCL_VectorSpace_d & Domain() const { return *dom;; }
      virtual HCL_VectorSpace_d & Range() const { return *rng; }

      virtual void Image( HCL_Vector_d & y ) const;
      virtual HCL_LinearOp_d * Deriv() const;

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

class EllipticSolOp3_d: public HCL_Op_d
{
   private:
      HCL_RnSpace_d * dom;
      HCL_RnSpace_d * rng;
      int M,N,L;
      double * NodeList;
      int * NodePtrs;
      int * ElList;
      double amin;

   public:

      EllipticSolOp3_d();
      EllipticSolOp3_d( const EllipticSolOp3_d & );
      EllipticSolOp3_d( char * fname );
      ~EllipticSolOp3_d();

      virtual HCL_VectorSpace_d & Domain() const { return *dom; }
      virtual HCL_VectorSpace_d & Range() const { return *rng; }
```

```
      virtual double MaxStep( const HCL_Vector_d & x,
      const HCL_Vector_d &dir) const;

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

      virtual ostream & Write( ostream & str ) const;
};
EvalEllipticSolOp3_d::EvalEllipticSolOp3_d()
{
   cerr << "Error: EvalEllipticSolOp3_d default constructor "
           "not implemented"
        << endl;
   exit(1);
}

EvalEllipticSolOp3_d::EvalEllipticSolOp3_d(
                      const EvalEllipticSolOp3_d & )
{
   cerr << "Error: EvalEllipticSolOp3_d copy constructor "
           "not implemented"
        << endl;
   exit(1);
}

EvalEllipticSolOp3_d::EvalEllipticSolOp3_d( HCL_RnSpace_d * d,
                                  HCL_RnSpace_d * r,
                                  int MM,int NN,int LL,double * NL,
                                  int * NP,int * EL,
                                  const HCL_RnVector_d & aa )
{
   dom = d;d->IncCount();
   rng = r;r->IncCount();
   M = MM;
   N = NN;
   L = LL;
   NodeList = new double[4*M];
   int i;
   for( i=0;i<4*M;i++ ) NodeList[i] = NL[i];
   NodePtrs = new int[N];
   for( i=0;i<N;i++ ) NodePtrs[i] = NP[i];
   ElList = new int[3*L];
   for( i=0;i<3*L;i++ ) ElList[i] = EL[i];
```

```
   HCL_RnVector_d & a = (HCL_RnVector_d&)aa;
   SymSparseRCSMatrix_d * K = new SymSparseRCSMatrix_d( N,N,5*N );
   HCL_RnVector_d b( N );
   F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                      a.Data(),K->rowstart,K->column,
                      K->data,K->nNonZero,K->nAlloc,b.Data() );
   Ka = new HCL_SymSparseRCSMatOp_d( K,1,rng );
   u = new HCL_RnVector_d( N );
   Ka->InvImage( b,*u );
   Bu = NULL;
}

EvalEllipticSolOp3_d::~EvalEllipticSolOp3_d()
{
   HCL_delete( Bu );
   HCL_delete( u );
   HCL_delete( Ka );
   delete[] ElList;
   delete[] NodePtrs;
   delete[] NodeList;
   HCL_delete( rng );
   HCL_delete( dom );
}

void EvalEllipticSolOp3_d::GetBu() const
{
   EvalEllipticSolOp3_d * t = (EvalEllipticSolOp3_d*)this;
   SparseRCSMatrix_d * B = new SparseRCSMatrix_d( N,M,5*M );
   F77NAME(assemble2)( (int&)(t->M),(int&)(t->N),(int&)(t->L),
                      t->NodeList,t->NodePtrs,t->ElList,
                      (t->u)->Data(),B->rowstart,B->column,
                      B->data,B->nNonZero,B->nAlloc );
   t->Bu = new HCL_SparseRCSMatOp_d( B,1,dom,rng );
}

void EvalEllipticSolOp3_d::SecondDerivImage( const HCL_Vector_d & dx1,
                                             const HCL_Vector_d & dx2,
                                             HCL_Vector_d & dy ) const
{
   if( dx1.Space() != Domain() )
   {
      cerr << "Error in EvalEllipticSolOp3_d::SecondDerivImage: first "
              "input perturbation is not in Domain" << endl;
```

```
   cerr << "Space of first input perturbation" << endl;
   dx1.Space().Write( cerr );
   cerr << "Domain" << endl;
   Domain().Write( cerr );
   exit(1);
}
if( dx2.Space() != Domain() )
{
   cerr << "Error in EvalEllipticSolOp3_d::SecondDerivImage: second "
           "input perturbation is not in Domain" << endl;
   cerr << "Space of second input perturbation" << endl;
   dx2.Space().Write( cerr );
   cerr << "Domain" << endl;
   Domain().Write( cerr );
   exit(1);
}
if( dy.Space() != Range() )
{
   cerr << "Error in EvalEllipticSolOp3_d::SecondDerivImage: output "
           "perturbation is not in range" << endl;
   cerr << "Space of output perturbation" << endl;
   dy.Space().Write( cerr );
   cerr << "Range" << endl;
   Range().Write( cerr );
   exit(1);
}
HCL_RnVector_d & da = (HCL_RnVector_d&)dx1;
HCL_RnVector_d & db = (HCL_RnVector_d&)dx2;
HCL_RnVector_d & du = (HCL_RnVector_d&)dy;

SymSparseRCSMatrix_d * K1 = new SymSparseRCSMatrix_d( N,N,5*N );
HCL_RnVector_d b1( N );
F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                   da.Data(),K1->rowstart,K1->column,
                   K1->data,K1->nNonZero,K1->nAlloc,b1.Data() );
HCL_SymSparseRCSMatOp_d * Kda =
       new HCL_SymSparseRCSMatOp_d( K1,1,rng );

HCL_RnVector_d du1( N ),du2( N ),tmp1( N ),tmp2( N );
Kda->Image( *u,tmp1 );
Ka->InvImage( tmp1,du1 );

K1 = new SymSparseRCSMatrix_d( N,N,5*N );
F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
```

```
                      db.Data(),K1->rowstart,K1->column,
                      K1->data,K1->nNonZero,K1->nAlloc,b1.Data() );
   HCL_SymSparseRCSMatOp_d * Kdb =
          new HCL_SymSparseRCSMatOp_d( K1,1,rng );

   Kdb->Image( *u,tmp2 );
   Ka->InvImage( tmp2,du2 );

   Kda->Image( du2,tmp1 );
   Kdb->Image( du1,tmp2 );
   HCL_delete( Kdb );
   HCL_delete( Kda );

   tmp1.Add( tmp2 );

   Ka->InvImage( tmp1,du );
}

void EvalEllipticSolOp3_d::SecondDerivPartialAdjImage(
                           const HCL_Vector_d & dx1,
                           const HCL_Vector_d & dy,
                                 HCL_Vector_d & dx2 ) const
{
   if( dx1.Space() != Domain() )
   {
      cerr << "Error in EllipticSolOpSecondDeriv1_d::"
              "SecondDerivPartialAdjImage: first " << endl
           << "input perturbation is not in Domain" << endl;
      cerr << "Space of first input perturbation" << endl;
      dx1.Space().Write( cerr );
      cerr << "Domain" << endl;
      Domain().Write( cerr );
      exit(1);
   }
   if( dx2.Space() != Domain() )
   {
      cerr << "Error in EllipticSolOpSecondDeriv1_d::"
              "SecondDerivPartialAdjImage: "
              "output perturbation is not in Domain" << endl;
      cerr << "Space of output perturbation" << endl;
      dx2.Space().Write( cerr );
      cerr << "Domain" << endl;
      Domain().Write( cerr );
      exit(1);
```

```
}
if( dy.Space() != Range() )
{
   cerr << "Error in EllipticSolOpSecondDeriv1_d::"
           "SecondDerivPartialAdjImage: "
           "second input perturbation is not in range" << endl;
   cerr << "Space of second input perturbation" << endl;
   dy.Space().Write( cerr );
   cerr << "Range" << endl;
   Range().Write( cerr );
   exit(1);
}
HCL_RnVector_d & da = (HCL_RnVector_d&)dx1;
HCL_RnVector_d & db = (HCL_RnVector_d&)dx2;
HCL_RnVector_d & dv = (HCL_RnVector_d&)dy;

if( !Bu ) GetBu();

HCL_RnVector_d du( N ),tmp1( N ),tmp2( N ),tmp3( M ),tmp4( N );

Ka->InvImage( dv,tmp1 );

SymSparseRCSMatrix_d * K1 = new SymSparseRCSMatrix_d( N,N,5*N );
HCL_RnVector_d b( N );
F77NAME(assemble)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                   da.Data(),K1->rowstart,K1->column,
                   K1->data,K1->nNonZero,K1->nAlloc,b.Data() );
HCL_SymSparseRCSMatOp_d * Kda =
       new HCL_SymSparseRCSMatOp_d( K1,1,rng );
Kda->Image( tmp1,tmp2 );
Kda->Image( *u,tmp4 );
Ka->InvImage( tmp4,du );
HCL_delete( Kda );

Ka->InvImage( tmp2,tmp4 );
Bu->AdjImage( tmp4,db );

SparseRCSMatrix_d B( N,M,5*M );
F77NAME(assemble2)( (int&)M,(int&)N,(int&)L,NodeList,NodePtrs,ElList,
                   du.Data(),B.rowstart,B.column,B.data,B.nNonZero,
                   B.nAlloc );
HCL_SparseRCSMatOp_d Bdu( &B,0,dom,rng );
Bdu.AdjImage( tmp1,tmp3 );
db.Add( tmp3 );
```

```
}

void EvalEllipticSolOp3_d::Image( HCL_Vector_d & y ) const
{
   if( y.Space() != Range() )
   {
      cerr << "Error in EllipticSolOp3_d::Image: output vector is not "
              "in range" << endl;
      cerr << "Space of output vector" << endl;
      y.Space().Write( cerr );
      cerr << "Range:"<< endl;
      Range().Write( cerr );
      exit(1);
   }
   y.Copy( *u );
}

HCL_LinearOp_d * EvalEllipticSolOp3_d::Deriv() const
{
   if( !Bu ) GetBu();

   HCL_InvLinearOp_d * Kinv = new HCL_InvLinearOp_d( Ka );
   HCL_ScaleLinearOp_d * Neg =
                    new HCL_ScaleLinearOp_d( &(Kinv->Range()),-1.0 );

   HCL_CompLinearOp_d * D =
                 new HCL_CompLinearOp_d( &(Bu->Domain()),
                                         &(Kinv->Range()),3 );
   D->SetNext( Bu );
   D->SetNext( Kinv );
   HCL_delete( Kinv );
   D->SetNext( Neg );
   HCL_delete( Neg );

   return D;
}

ostream & EvalEllipticSolOp3_d::Write( ostream & str ) const
{
   str << "EvalEllipticSolOp3_d with domain" << endl;
   Domain().Write( str );
   str << "and range" << endl;
   Range().Write( str );
   return str;
```

```
}

EllipticSolOp3_d::EllipticSolOp3_d()
{
   cerr << "Error: EllipticSolOp3_d default constructor not implemented"
        << endl;
   exit(1);
}

EllipticSolOp3_d::EllipticSolOp3_d( const EllipticSolOp3_d & )
{
   cerr << "Error: EllipticSolOp3_d copy constructor not implemented"
        << endl;
   exit(1);
}

EllipticSolOp3_d::EllipticSolOp3_d( char * fname )
{
   ifstream meshfile( fname );
   meshfile >> amin >> M >> N >> L;

   NodeList = new double[4*M];
   NodePtrs = new int[N];
   ElList = new int[3*L];

   for( int i=0;i<M;i++ )
      for( int j=0;j<4;j++ )
         meshfile >> NodeList[j*M+i];
   for( int i=0;i<N;i++ )
      meshfile >> NodePtrs[i];
   for( int i=0;i<L;i++ )
      for( int j=0;j<3;j++ )
         meshfile >> ElList[j*L+i];

   dom = new HCL_RnSpace_d( M );
   rng = new HCL_RnSpace_d( N );
}

EllipticSolOp3_d::~EllipticSolOp3_d()
{
   HCL_delete( rng );
   HCL_delete( dom );
   delete[] ElList;
   delete[] NodePtrs;
```

```
   delete[] NodeList;
}

double EllipticSolOp3_d::MaxStep( const HCL_Vector_d & x,
                                  const HCL_Vector_d &dir) const
{
   if( x.Space() != Domain() )
   {
      cerr << "Error in EllipticSolOp3_d::MaxStep: input point is not "
              "in domain" << endl;
      cerr << "Space of input point" << endl;
      x.Space().Write( cerr );
      cerr << "Domain" << endl;
      Domain().Write( cerr );
      exit(1);
   }
   if( dir.Space() != Domain() )
   {
      cerr << "Error in EllipticSolOp3_d::MaxStep: input direction "
              "is not in domain" << endl;
      cerr << "Space of input direction" << endl;
      dir.Space().Write( cerr );
      cerr << "Domain" << endl;
      Domain().Write( cerr );
      exit(1);
   }
   HCL_RnVector_d & a = (HCL_RnVector_d&)x;
   HCL_RnVector_d & da = (HCL_RnVector_d&)dir;

   HCL_FPArch_d arch;
   double alpha = arch.large;
   for( int i=1;i<=M;i++ )
      if( a(i) < amin )
         return -1.0;
      else if( da(i) < 0.0 )
         alpha = min( alpha,(amin-a(i))/da(i) );
   return alpha;
}

HCL_EvaluateOp_d * EllipticSolOp3_d::Evaluate(
                   const HCL_Vector_d & x ) const
{
   if( x.Space() != Domain() )
   {
```

```
        cerr << "Error in EllipticSolOp3_d::Evaluate: input point is not "
                "in domain" << endl;
        cerr << "Space of input point" << endl;
        x.Space().Write( cerr );
        cerr << "Domain" << endl;
        Domain().Write( cerr );
        exit(1);
     }
     if( MaxStep( x,x ) < 0.0 )
     {
        cerr << "Error in EllipticSolOp3_d::Evaluate: input point is not "
                "in domain " << endl
             << "(though it belongs to the correct vector space)"
             << endl;
        exit(1);
     }
     HCL_RnVector_d & a = (HCL_RnVector_d&)x;
     return new EvalEllipticSolOp3_d( dom,rng,M,N,L,NodeList,
                                      NodePtrs,ElList,a );
  }

  ostream & EllipticSolOp3_d::Write( ostream & str ) const
  {
     str << "EllipticSolOp3_d with domain" << endl;
     Domain().Write( str );
     str << "and range" << endl;
     return Range().Write( str );
  }
```

5 Summary

As detailed in this paper, there are several methods for implementing operators (simulators) 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.

The model problem presented in this paper required that two classes be written in order to attain good runtime performance. We wish to emphasize that for many types of simulation, such as that based on finite differences, the simpler implementation, as described in Section 2, gives perfectly satisfactory performance and only requires the implementor to write one class. The following guide summarizes four basic models for implementing an operator in HCL. As explained at the beginning of Section 4.1, some variation on these models is possible. Nevertheless, an implementor should be able to choose on of these models so as to obtain good runtime performance with minimal programming effort.

Guide to implementing an operator G in HCL

1. (a) **Indications:**
      i. The computation of $G(x)$, $DG(x)$, and $D^2G(x)$ do not share common (non-trivial) intermediate quantities.
      ii. There are no convenient (existing) classes for implementing the derivatives.

   (b) **Method:** Create a single class, `GOp`, derived from `HCL_Op`. Override the protected member functions
      - `Image1`,
      - `DerivImage`,
      - `DerivAdjImage`,
      - `SecondDerivImage`, and
      - `SecondDerivPartialAdjImage`
      - (and `DerivInvImage`, `DerivInvAdjImage`, if desired).

      Do *not* override the member functions `Evaluate`, `Deriv1`, and `SecondDeriv1`.

   (c) **Comments:** The evaluation object produced by `Evaluate` will be an instance of `HCL_OpDefaultEval`, and the derivatives will be instances of `HCL_OpDefaultDeriv` and `HCL_OpDefaultSecondDeriv`.

2. (a) **Indications:**
      i. The computation of $G(x)$, $DG(x)$, and $D^2G(x)$ do not share common (non-trivial) intermediate quantities.
      ii. The derivative operators can be implemented using existing classes.

   (b) **Method:** Create a single class, `GOp`, derived from `HCL_Op`. Override the protected member functions `Image1`, `Deriv1`, `SecondDeriv1`. Do *not* override the protected member functions
      - `DerivImage`,
      - `DerivAdjImage`,
      - `SecondDerivImage`,
      - `SecondDerivPartialAdjImage`,
      - `DerivInvImage`, and
      - `DerivInvAdjImage`.

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

   (c) **Comments:** The evaluation object produced by `Evaluate` will be an instance of `HCL_OpDefaultEval`, and the derivatives will be instances of the classes chosen by the implementor. Note that a variation on this method is to implement one of the derivatives, say $DG(x)$ using an existing class (this requires overriding `Deriv1`), and implement the other derivative using the protected image methods; for $D^2G(x)$ this means overriding `SecondDerivImage` and `SecondDerivPartialAdjImage`. See Section 4.1 for an example of this.

3. (a) **Indications:**
      i. The computation of $G(x)$, $DG(x)$, and $D^2G(x)$ share common (nontrivial) intermediate quantities.
      ii. The derivative operators can be implemented using existing classes.
   (b) **Method:** Create two classes, `GOp` (derived from `HCL_Op`) and `EvalGOp` (derived from `HCL_EvaluateOp`). Override the public member function `Evaluate` of `HCL_Op` to call the constructor of `EvalGOp`. Do not override any of the `HCL_Op` functions
      - `Image1`,
      - `Deriv1`,
      - `SecondDeriv1`,
      - `DerivImage`,
      - `DerivAdjImage`,
      - `DerivInvImage`,
      - `DerivInvAdjImage`.

      Override the public member functions
      - `Image`,
      - `Deriv`,
      - `SecondDerivImage`

      of `HCL_EvaluateOp`. The derivative methods create instances of the classes chosen by the implementor.
   (c) **Comments:** The evaluation object produced by `Evaluate` will be an instance of `EvalGOp`, and the derivatives will be instances of the classes chosen by the implementor.

4. (a) **Indications:**
      i. The computation of $G(x)$, $DG(x)$, and $D^2G(x)$ share common (nontrivial) intermediate quantities.
      ii. There are no convenient (existing) classes for implementing the derivatives.
   (b) **Method:** Create two classes, `GOp` (derived from `HCL_Op`) and `EvalGOp` (derived from `HCL_EvaluateOp`). Override the public member function `Evaluate` of `HCL_Op` to call the constructor of `EvalGOp`. Do not override any of the `HCL_Op` functions
      - `Image1`,
      - `Deriv1`,
      - `SecondDeriv1`,
      - `DerivImage`,
      - `DerivAdjImage`,
      - `DerivInvImage`,
      - `DerivInvAdjImage`.

      Override the public member function `Image`, and the protected image methods

- `DerivImage`,
- `DerivAdjImage`,
- `SecondDerivImage`,
- `SecondDerivPartialAdjImage`
- (and `DerivInvImage`, `DerivInvAdjImage`, if desired)

of `HCL_EvaluateOp`. Do not override the public member functions `Deriv` and `SecondDeriv` of `HCL_EvaluateOp`.

(c) **Comments:** The evaluation object produced by `Evaluate` will be an instance of `EvalGOp`, and the derivatives will be instances of `HCL_OpDefaultDeriv` and `HCL_OpDefaultSecondDeriv`. As with the first two methods, one can combine these last two methods by implementing one derivative with an existing class and the other using the protected image methods.

References

[1] Mark S. Gockenbach. An overview of HCL1.0.

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

[3] 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.