Efficient and automatic implementation of the adjoint state method

Mark S. Gockenbach
Michigan Technological University

and
Daniel R. Reynolds
Rice University

and
William W. Symes
Rice University

Combinination of object-oriented programming with automatic differentiation techniques facilitates the solution of data fitting, control, and design problems driven by explicit time stepping schemes for initial-boundary value problems. The C++ class `fdtd` takes a complete specification of a *single step*, along with some associated code, and assembles from it a complete simulator, along with the linearized and adjoint simulations. The result is a (nonlinear) operator in the sense of the Hilbert Class Library (HCL), a C++ software package for optimization. The HCL operator so produced links directly with any of the HCL optimization algorithms. Moreover the performance of simulators constructed in this way is equivalent to that of optimized Fortran implementations.

Categories and Subject Descriptors: G.1.8 [**Numerical Analysis**]: Partial Differential Equations—*finite difference methods*; G.4 [**Mathematical Software**]: *efficiency, user interface*

This work was supported in part by the National Science Foundation under grants DMS-9627355, DMS-9973423, and DMS-9973308, by the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23, as part of the prime contract (W-7405-ENG-36) between the Department of Energy and the Regents of the University of California, and by The Rice Inversion Project.
Name: Mark S. Gockenbach
Address: Department of Mathematical Sciences, Michigan Technological University, Houghton, MI 49931; msgocken@mtu.edu
Name: Daniel R. Reynolds
Address: Department of Computational and Applied Mathematics, Rice University, Houston, TX 77251-1892; reynoldd@caam.rice.edu
Name: William W. Symes
Address: Department of Computational and Applied Mathematics, Rice University, Houston, TX 77251-1892; symes@caam.rice.edu

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Object-oriented design, optimization, simulation

1. INTRODUCTION

Explicit marching or time-stepping schemes based on finite difference or finite element discretization of initial/boundary value problems have several appealing properties for modeling of wave motion and front propagation. Such problems arise in simulation of seismic waves in the Earth's crust, acoustic waves in the ocean, shock and rarefaction waves in gases and fluids, flame fronts in reacting gases, and in less obvious technological domains such as image processing and mathematical morphology. These marching schemes have roughly the same sampling requirements in time and space as do the phenomena that they model. They can accommodate (in principle) nearly any degree of material heterogeneity, and their basic convergence behavior is mostly well-understood and easy to control (although some mysteries remain open on both counts). Especially when their grids are logically rectangular, such codes can take advantage of many compiler optimizations and automatic parallelizations. Finally, one may employ such schemes to implement various data-fitting algorithms to estimate internal structure of the material from boundary measurements, to control parts of the fields via proper choice of parameters, and to design materials with prescribed responses. Linearization of the mappings (from parametric material description to boundary measurements) defined by these schemes is an important component of these data fitting algorithms, as are the adjoints of these linearized maps. See [Claerbout 1992] for the importance of adjoints in geophysical data-fitting problems, for example.

The use of time stepping schemes as drivers in data-fitting exercises encounters two impediments, however, both well known to practitioners. First, while the code for linearized maps is relatively easy to write, the code for their adjoints is much more difficult, and naive designs for the adjoint computation can easily lead to gross inefficiency. Second, when the basic simulator and its linearized and adjoint linearized derivative codes are successfully coded in procedural style (for example in Fortran), linkage to high-quality numerical optimization code is problematic. The optimization algorithms require a specific data structure, usually simple arrays, whereas the simulation of wave motion will generally require considerably more structure, for example definition of grids, which must be kept available but out of the way of the optimization software. Thus the linkage demands the construction of elaborate, fragile, and non-reusable packaging procedures. This process is so tedious that many applications scientists simply write their own optimization software, adapted to the data structures appropriate to their work (for example [Giering 1999]). In the process they neglect standard numerical software libraries, products of decades of effort.

This paper presents an attempt to resolve both of these problems, through the definition of a *abstract time stepping scheme*, implemented as a C++ class **fdtd** ("**F**inite **D**ifference **T**ime **D**omain", a standard term for time stepping methods in electromagnetic simulation). The class implements the general structure of an explicit time-stepping scheme, as well as its linearization and two variants of the adjoint scheme, one appropriate for conservative problems, the other for dissipative problems. In particular the dissipative adjoint scheme encodes Andreas Griewank's optimal checkpointing scheme [Griewank 1992]. The class supplies the overall structure of the scheme; the user is expected to supply an operator (the *stencil operator*) defining a single time step, along with its linearization and adjoint. The stencil operator is simple enough for most schemes that almost any modern automatic differentiation toolbox, for example ADIFOR [Bischof et al. 1992] or TAMC [Giering 1999], can return linearized, and in some cases adjoint, code. So the user may well need to write only the basic simulator: all of the associated code for derivatives and adjoints can be generated automatically and fit into prepared places in the **fdtd** template.

Furthermore, the **fdtd** class defines an *operator* in the sense of the Hilbert Class Library (HCL), a C++ software package for optimization [Gockenbach and Symes 1996; Gockenbach et al. 1999]. HCL defines vectors, operators, functions, and associated mathematical objects as C++ objects, with the minimal structure appropriate for each. It therefore creates a *standard platform* for implementations of optimization and other algorithms, giving these algorithms a very wide degree of applicability. For example the notion of vector codified in HCL encompasses both small Fortran-style arrays (of any dimension) and out-of-core data sets with sizes measured in Gigabytes.

Since a time stepping scheme implemented using **fdtd** is automatically an operator in the HCL sense, it can linked directly any optimization algorithms implemented in HCL, with no need for further interface code. That is, the HCL classes automatically hide the natural data structures of the simulator from the optimization and other high-level code. Thus the user need write no fragile, single-use interface code.

In this paper we explain the **fdtd** class, beginning with the structure of the mathematical objects it is to implement and continuing with the overall structure of the code. We conclude with a detailed description of a concrete example, implementing 2D linear acoustic wave finite difference simulation. The core numerical routines of this simulator are written in Fortran; some of these are output of an automatic differentiation package (TAMC). Moreover the mixed C++/Fortran code retains all of the advantages of Fortran compiler optimizations and even (shared memory style) parallelization, and therefore exhibits essentially the same floating point throughput as a pure Fortran implementation—because almost all of the floating point arithmetic is still handled in Fortran! Of course other techniques for efficient coding of numerical kernels, for example expression templates [Haney, Crotinger, Karmesin, and Smith 1999], could play the same role as the Fortran kernels in our example.

The closely related work [Coleman, Santosa, and Verma 2000] takes essentially the same approach to differentiation of operators defined by time-stepping schemes: that is, application of AD at the stencil level, with the results "glued" together by a higher-level code template, thus avoiding various fatal inefficiencies of contemporary AD technology. Our approach differs from theirs in a couple of key respects. Our C++ class `fdtd` separates the data structures of the time-stepping problem much more cleanly from the high-level interface than does the MATLAB template described in [Coleman, Santosa, and Verma 2000]. Thus essential aspects of data structure, such as disk i/o, are transparent to `fdtd`, which cannot be the case for a MATLAB template. Also we have implemented the optimal checkpointing scheme introduced in [Griewank 1992] as an integral feature of `fdtd`.

2. ABSTRACT FORMULATION OF TIME-STEPPING SCHEMES

As mentioned in the previous section, one of the reasons for implementing time stepping schemes in a C++ class is to ease the use of optimization in data-fitting problems. For example, suppose that one or more coefficients in the underlying partial differential equation are to be estimated (a typical "inverse" problem) by the Output Least-Squares (OLS) technique. In this technique, the parameters are chosen to produce simulated data as close as possible (in a norm induced by an inner product) to observed data. Specifically, we solve

$$\min_c \frac{1}{2}\|G(c) - D^{obs}\|^2, \tag{1}$$

where $c \in C$ denotes the unknown parameters, D^{obs} is the observed data, and G is the *forward map*, that is, the operator embodying the mathematical model of the dependence of the data on the parameters. The forward map G involves simulation of physical space-time fields, followed by sampling. In a typical application, only parta of the fields are observable. We therefore assume that

$$G(c) = SU = \sum_{n=0}^{N} S_n U^n,$$

where $U^n \in \mathcal{U}$ is (related to) the nth time level of the simulated field, $S : \mathcal{U} \to \mathcal{D}$ is the *sampling operator*, and $\mathcal{D}$ is the data space (that is, $D^{obs} \in \mathcal{D}$).

Note that S is defined by

$$SU = \sum_{n=0}^{N} S_n U^n,$$

where $S_n : \mathcal{U} \to D$ for $n = 0, 1, \ldots, N$. That is, each time level of the computed field is sampled, and the results are accumulated as the data. This formalism provides an efficient way to abstractly represent several different sampling possibilities. For example, the entire time level U^n may be recorded for certain values of n, in which case S_n is the zero operator for all other values of n. Alternatively, every time level could be sampled at a few "receiver" locations (as in the typical seismic experiment), and the results recorded as time series. At the other extreme the entire history of the field could be retained. All of these possibilities can be accomodated within the above formalism by appropriate choice of S.

Most algorithms for solving (1) require the gradient of the OLS functional

$$J(c) = \frac{1}{2}||G(c) - D^{obs}||^2.$$

We have

$$\nabla J(c) = DG(c)^*(G(c) - D^{obs}),$$

so we require the adjoint of the derivative (linearization) $DG(c)$. We now discuss a formulation of explicit marching methods that eases the computation of this adjoint.

Any marching scheme can be considered to be formally two-level, by concatenating several time levels if necessary (an example of this is given in Section 4). Therefore, we can write

$$U^{n+1} = H_n(c, U^n), \;\; n = 0, 1, \ldots, N-1.$$

We call $H_n : C \times \mathcal{U} \to \mathcal{U}$ the *stencil operator.*

The *linearization* of the map $c \mapsto G(c)$ is the result of first-order perturbation of the time-stepping equations:

$$DG(c)\delta c = \mathcal{S}\delta U,$$

where

$$\delta U^0 = 0, \;\; \delta U^{n+1} = D_c H_n(c, U^n)\delta c + D_U H_n(c, U^n)\delta U^n$$

and $\delta U^n = (DU(c)\delta c)^n$. Note that if the original marching scheme is linear (really *affine*: linear plus constant), then it can be written as

$$U^{n+1} = A(c)U^n + F^n,$$

where $A(c) = D_U H(c, U)$ (H is independent of the time level n in this case). It follows that δU satisfies

$$\delta U^{n+1} = A(c)\delta U^n + (DA(c)\delta c)U^n.$$

Therefore, in this common case, the linearization is computed by a iteration identical to the original, except that the "right-hand side" F^n is replace by $(DA(c)\delta c)U^n$.

We now return to the general case to compute the adjoint. We require inner products for the spaces $\mathcal{C}$ and $\mathcal{U}$; these inner products will be denoted $(\cdot,\cdot)_{\mathcal{C}}$ and $(\cdot,\cdot)_{\mathcal{U}}$, respectively. The field U belongs to $\mathcal{U}^{N+1}$, and we define the inner product on $\mathcal{U}^{N+1}$ by

$$(U, V)_{\mathcal{U}^{N+1}} = \sum_{n=0}^{N} (U^n, V^n)_{\mathcal{U}}.$$

For convenience, write A_n for $D_U H_n(c, U^n)$ and δF^{n+1} for $D_c H_n(c, U^n)\delta c$, $\delta F^0 = 0$, so that the linearized scheme can be written as

$$\delta U^0 = 0, \ \delta U^{n+1} - A_n \delta U^n = \delta F^{n+1}, \ n = 0, 1, \ldots, N-1.$$

We can also write this as

$$M\delta U = \delta F,$$

where $M : \mathcal{U}^{N+1} \to \mathcal{U}^{N+1}$ is the block linear operator

$$M = \begin{bmatrix} I & 0 & 0 & \cdots & 0 \\ -A_0 & I & 0 & \cdots & 0 \\ 0 & -A_1 & I & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & -A_{N-1} & I \end{bmatrix}$$

(note that M depends on c, but we suppress this dependence). We then see that $\delta U = M^{-1}\delta F$, and recognize that the explicit time-stepping scheme is equivalent to solving $M\delta U = \delta F$ by forward substitution.

To complete the computation of $DG(c)^*$, write B for the operator mapping δc to δF:

$$(B\delta c)^n = \begin{cases} 0, & n = 0 \\ D_c H_{n-1}(c, U^{n-1})\delta c, & n = 1, 2, \ldots, N \end{cases}$$

(again we suppress the fact that B depends on c). We now see that

$$DG(c) = \mathcal{S} M^{-1} B,$$

and so

$$DG(c)^* = B^* M^{-*} \mathcal{S}^*.$$

We assume that $\mathcal{S}$ and $\mathcal{S}^*$ are supplied by the user along with the stencil operator H_n and its derivatives and adjoints $D_c H_n(c, U)$, $D_U H_n(c, U)$, $D_c H_n(c, U)^*$, and $D_U H_n(c, U)^*$, and now show how to compute $DG(c)^*$ from these pieces.

We have $DG(c)^*\delta D = B^* M^{-*} \mathcal{S}^* \delta D$. Write $\delta V = \mathcal{S}^*\delta D$. A simple calculation shows that

$$(S^* D)^n = S_n^* D.$$

We now compute $M^{-*}\delta V$. It is easy to see from our choice of inner product on $\mathcal{U}^{N+1}$ that M^* is the block linear operator

$$M^* = \begin{bmatrix} I & -A_0^* & 0 & \cdots & 0 \\ 0 & I & -A_1^* & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & I & -A_{N-1}^* \\ 0 & 0 & \cdots & 0 & I \end{bmatrix}.$$

Write $\delta W = M^{-*}\delta V$, so that δW solves $M^*\delta W = \delta V$. Since M^* is block upper triangular, we solve for δW by back substitution, which is equivalent to the following reverse time-stepping scheme:

$$\delta W^N = \delta V^N,\ \delta W^{n-1} = A^*_{n-1}\delta W^n + \delta V^{n-1},\ n = N, N-1, \ldots, 1.$$

We refer to δW as the *adjoint state* and to the equation $M^*\delta W = \delta V$ as the *adjoint state equation.*

Next we compute B^*. We have

$$\begin{aligned}
(B\delta c, \delta W)_{\mathcal{U}^{N+1}} &= (0, \delta W^0)_{\mathcal{U}} + \sum_{n=1}^{N} (D_c H_{n-1}(c, U^{n-1})\delta c, \delta W^n)_{\mathcal{U}} \\
&= \sum_{n=1}^{N} (\delta c, D_c H_{n-1}(c, U^{n-1})^* \delta W^n)_C \\
&= (\delta c, \sum_{n=1}^{N} D_c H_{n-1}(c, U^{n-1})^* \delta W^n)_C.
\end{aligned}$$

This shows that

$$B^*\delta W = \sum_{n=1}^{N} D_c H_{n-1}(c, U^{n-1})^* \delta W^n.$$

Thus the procedure for computing $DG(c)^*\delta D$, for $\delta D \in \mathcal{D}^{N+1}$, is:

(1) Solve the simulation problem to produce the field U (needed in steps 3b and 3c).

(2) Set δc to zero.

(3) For $n = N, N-1, \ldots, 1:$
   - (a) Compute $\delta V^n = S^*_n \delta D$.
   - (b) Compute δW^n by taking one step (backward in time) on the adjoint state equation (or simply $\delta W^N = \delta V^N$).
   - (c) Add $D_c H_{n-1}(c, U^{n-1})^* \delta W^n$ to the output vector δc.

A logistical problem immediately asserts itself: U is produced by stepping *forward* in time, δW by stepping *backwards*. Unless the state space has small dimension (which we most definitely do not want to assume!), storage of the entire time history of the reference field U is out of the question.

For conservative problems, in which some norm of U is more or less preserved during its evolution, backwards time stepping is stable. This is the case for the acoustic example explored later in this paper. Then the final value only of U^n (at the largest time step) is used as the initial data to run the scheme for U backwards (decreasing n) simultaneously with the adjoint state scheme for δW. The sum of the terms $D_c H_{n-1}(c, U^{n-1})^* \delta W^n$ is accumulated, and when $n = 0$ is reached the adjoint has been computed.

If the underlying continuous system is dissipative, this simple approach will likely not work, as the scheme stands a good chance of being backwards unstable. In that case, a *checkpointing scheme* due to Andreas Griewank [Griewank 1992], extended by Joakim Blanch [Blanch et al. 1998], is employed. The idea is to combine saving ("checkpointing") various time levels U^n to use as intermediate initial data to restart the computation of U during the solution of the adjoint state system. A complete description of the algorithm appears below.

3. THE CLASS `FDTD`

3.1 The Hilbert Class Library

The Hilbert Class Library (HCL) [Gockenbach and Symes 1996; Gockenbach et al. 1999] is a library of C++ classes representing the basic mathematical objects (vectors, functions, operators, etc.) that are used in formulating and solving optimization problems. HCL allows optimization algorithms to be implemented in an abstract fashion. Practically speaking, this means that the optimization code does not deal directly with the data structures or interfaces used by application code. Therefore, optimization algorithms implemented using HCL classes can be used with application code of arbitrary complexity.

In order to solve the OLS problem (1) using an HCL optimization algorithm, it is necessary to represent the forward map G described in a class derived from `HCL_Op`. The class `HCL_Op` is the abstract base class for nonlinear operators in the HCL framework; it mandates that a nonlinear operator G must have the following properties:

—it can identify its domain and range (as vector spaces);
—it can compute $G(c)$ and $DG(c)$. The image $G(c)$ is a vector (that is, an instance of `HCL_Vector`) and the derivative $DG(c)$ is a linear operator (an instance of `HCL_LinearOp`).

The linear operator base class, `HCL_LinearOp`, requires that a linear operator be able to:

—identify its domain and range (as vector spaces);
—compute its image on a vector in the domain;
—compute the image of the adjoint operator on a vector in the range.

In order to create an instance of `fdtd` implementing a specific marching scheme, one must implement the stencil operator H_n, as well as its derivatives and their adjoints, as an operator in the HCL framework. In addition, the sampling operator must be implemented as a linear operator in the HCL sense. We have created a two base classes to facilitate the implementation of these operators:

(1) `StencilOp`, derived from `HCL_OpProductDomain`. The base class represents operators depending on more than one independent variable (therefore, for instance, partial derivatives are meaningful).
(2) `SamplingOp`, derived from `HCL_LinearOp`.

These are both abstract base classes, meaning that they provide a template for creating the necessary derived classes. The actual work involved in creating a derived class is limited to the implementation of the following methods:

—`StencilOp`
  —`CheckStability` Takes as input the PDE parameter and checks to see if the stability condition is satisfied. `fdtd` calls this method once before performing a simulation.
  —`Image` Computes $H_n(c, U^n)$ and overwrites U^n with the result U^{n+1}.
  —`PartialDerivImageState` Computes $D_U H_n(c, U^n)\delta U^n$ and overwrites δU^n with the result.
  —`PartialDerivAdjImageState` Computes $D_U H_n(c, U^n)^* \delta W^{n+1}$ and overwrites δW^{n+1} with the result.
  —`PartialDerivImageAddControl` Computes $D_c H_n(c, U^n)\delta c^n$ and adds it to a given vector.
  —`PartialDerivAdjImageAddControl` Computes $D_c H_n(c, U^n)^* \delta W^{n+1}$ and adds it to a given vector.
—`SampleOp`
  —`ImageAdd` Computes $S_n U^n$ and adds it to a given vector.
  —`AdjImageAdd` Computes $S_n^* \delta D$ and adds it to a given vector.

Note that the user is required to provide precisely those manipulations that are needed by fdtd. For example, the operation

$$y \leftarrow y + S_n U^n$$

is needed to compute the sum

$$SU = \sum_{n=0}^{N} S_n U^n$$

efficiently. Similarly, examination of the above formulas will show why we need

$$y \leftarrow D_U H_n(c, U^n)\delta U^n,$$

but

$$y \leftarrow y + D_c H_n(c, U^n)\delta c^n.$$

The design of `fdtd` and the helper classes `StencilOp` and `SampleOp` ensures that the user must provide only those calculations that are special to his or her problem. There is a small amount of "boilerplate" code that is also required to package these computations as HCL classes. The effort required to define the classes derived from `StencilOp` and `SampleOp` has a side benefit: HCL has methods for checking the correctness of the partial derivatives of H_n and the adjoints of S_n and the derivatives of H_n. Errors in coding derivatives and adjoints are all too common, and the HCL framework makes it easy to localize these errors, which is the first and sometimes the hardest part of correcting them.

In addition to the classes mentioned above, one or more vector classes will be involved. The user must choose vector classes, derived from `HCL_Vector`, to represent the vectors appearing in the problem: c, U^n, U, and D. In many cases, these will be standard concrete classes found in HCL, and there will be no programming required. For some applications, especially those with complicated and extensive data structures, users may find it worthwhile to define a vector class implementing data structures special to their application. The ability of HCL to work with arbitrary data structures is one of its most powerful features.

3.2 The implementation of `fdtd`

The class `fdtd` is derived from `HCL_Op`, and has the following methods:

—Class constructor:

```
fdtd::fdtd( StencilOp * H,SamplingOp * S,
            HCL_Vector * CauchyData,
            HCL_VectorSpace * DataSpace,int N );
```

The objects pointed to by `H` and `S` must be previously created instances of `StencilOp` and `SampleOp`; as these are user-defined, it is not possible to say precisely how they are constructed, although we show an example in Section 4. The object pointed to by `CauchyData` is a vector and represents U^0. The object pointed to by `DataSpace` represents the data space (the range of the operator G). Finally, the integer `N` is the number of time steps.

—`Image` The method

```
fdtd::Image( const HCL_Vector & c,HCL_Vector & D );
```

performs the operation

$$D \leftarrow G(c);$$

the vectors `c` and `D` must belong to the domain and range of G, respectively, and `c` must have previously been assigned a meaningful value. Here is the code implementing `Image` (stripped of nonessentials such as error-checking):

```
// Check to see that stability condition is satisfied.

if( !H->CheckStability( c ) )
{
   cerr << "Error in fdtd_d::Image1: stability test "
           "failed; terminating" << endl;
   exit(1);
```

```
}

H->InitializeControl( &c );
H->StateVector().Copy( *CauchyData );
D.Zero();
S->SetTime( 0 );
S->ImageAdd( H->StateVector(),D );
int n;
for( n=1;n<=N;n++ )
{
   H->SetTime( n-1 );
   H->Image();
   S->SetTime( n );
   S->ImageAdd( H->StateVector(),D );
}
```

—`DerivImage` The method

```
fdtd::DerivImage( const HCL_Vector & c,const HCL_Vector & dc,
                  HCL_Vector & dD );
```

performs the operation

$$\delta D \leftarrow DG(c)\delta c.$$

The code for this method is very similar to that for `Image`.

—`DerivAdjImage` The method

```
fdtd::DerivAdjImage( const HCL_Vector & c,
                     const HCL_Vector & dD,
                           HCL_Vector & dc );
```

performs the operation

$$\delta c \leftarrow DG(c)^{*}\delta D.$$

When this command is invoked, the code checks the `ConservativeFlag` parameter; if it is set, the scheme is assumed to be conservative, in which case the more efficient backward time-stepping scheme can be used to computed U^n during the solution of the adjoint state equation. Here is the code (again, stripped of nonessentials) for the (internal) method used by `fdtd` to perform this computation:

```
 // First, do the forward simulation and
// save the final time level.

H->InitializeControl( &c );
H->StateVector().Copy( *CauchyData );

int n;
for( n=1;n<=N;n++ )
{
   H->SetTime( n-1 );
   H->Image();
}
```

```
// Now, solve the adjoint state equation,
// stepping backward in time.

x.Zero();
S->SetTime( N );
S->AdjImage( D,H->StatePerturbation() );

for( n=N;n>=1;n-- )
{
   // Take one step (backward in time)
   // on the forward simulation
   H->SetTime( n-1 );
   H->PartialInvImage();

   // Accumulate contribution to the end result.
   H->SetTime( n-1 );
   H->PartialDerivAdjImageAddControl( x );

   // Take one step (backward in time)
   // on the adjoint state equation
   if( n > 1 )
   {
      H->PartialDerivAdjImageState();
      S->SetTime( n-1 );
      S->AdjImageAdd( D,H->StatePerturbation() );
   }
}
```

If the scheme is dissipative, then the more complicated algorithm implementing checkpointing must be used to compute $DG(c)^*\delta D$. The code implementing this algorithm is considerably more complex, since the steps shown above must be interspersed with commands to integrate forward in time from the nearest checkpoint to compute U^n as needed.

3.3 Using `fdtd`

We have introduced what may seem a bewildering plethora of classes. We now summarize how a prospective user, who wishes to implement a specific time stepping scheme, interacts with these classes. The user must create two classes, one derived from `StencilOp` (to represent H_n) and the other from `SampleOp` (to implement S_n). The core numerical code for these classes may be implemented in Fortran, C, C++, or perhaps even another language, and may employ runtime libraries, compiler directives, or compiler-generated loop optimizations to achieve efficient scalar and/or parallel execution. The stencil and sampling classes exist merely to serve as interfaces between `fdtd` and the numerical kernels. The stencil operator is the workhorse; it must implement methods for computing $H_n(c,U)$, $D_cH_n(c,U)\delta c$,

$D_U H_n(c,U)\delta U$, $D_c H_n(c,U)^*\delta D$, and $D_U H_n(c,U)^*\delta D$. Recall that all of these quantities relate to a single time step, and so are relatively simple compared to the overall scheme.

Having defined these classes, one can create the operator G simply by creating instances of the stencil and sample operators in a C++ program and passing them to the constructor of the `fdtd` class.

4. A CONCRETE EXAMPLE

4.1 The (2,4) Leapfrog Scheme for 2D Acoustic Seismic Simulation

The 2D acoustic line radiator provides a simple example of the scheme explained in the preceding sections. The partial differential equation to be solved is

$$\frac{1}{v^2(x,z)}\frac{\partial^2 p}{\partial t^2} - \frac{\partial^2 p}{\partial x^2} - \frac{\partial^2 p}{\partial z^2} = f(t)\delta(x-x_s)\delta(z-z_s)$$

in the space-time domain

$$x_{\rm min} \le x \le x_{\rm max},\ z_{\rm min} \le z \le z_{\rm max},\ t_{\rm min} \le t \le t_{\rm max}$$

with initial condition

$$p(x,z,t) \equiv 0,\ t \le t_{\rm min}$$

and Dirichlet boundary conditions

$$p(x_{\rm min},z,t) = p(x_{\rm max},z,t) \equiv 0,\ z_{\rm min} \le z \le z_{\rm max},\ t_{\rm min} \le t \le t_{\rm max}$$

and

$$p(x,z_{\rm min},t) = p(x,z_{\rm max},t) \equiv 0,\ x_{\rm min} \le x \le x_{\rm max},\ t_{\rm min} \le t \le t_{\rm max}.$$

In this problem the field $p(x,z,t)$ models the pressure per unit length in the transverse (y) direction, and $f(t)$ is the time-varying strength of an energy source concentrated on a line parallel to the y axis. Since the sound velocity $v(x,z)$ in the fluid and the initial pressure field are both independent of y, so is the field at all times.

We approximate the wave equation with a finite difference scheme of second order in time and fourth order in space, of leapfrog (centered difference) type, as explained for example in [Levander 1988]. Choose grid steps Δx, Δz, and Δt, and write

$$p^n_{i,j} \simeq p(x_{\rm min} + j\Delta x, z_{\rm min} + i\Delta z, t_{\rm min} + n\Delta t)$$

We assume that the steps are chosen so that for integers N_x, N_z, and N_t,

$$x_{\rm max} - x_{\rm min} = N_x\Delta x,\ z_{\rm max} - z_{\rm min} = N_z\Delta z,\ t_{\rm max} - t_{\rm min} = N_t\Delta t,$$

Reflecting the initial conditions, we set

$$p^0_{i,j} = p^{-1}_{i,j} \equiv 0,\ i = 0,...,N_z,\ j = 0,...,N_x.$$

The basic finite difference stencil is

$$p_{i,j}^{n+1} = -p_{i,j}^{n-1} + 2p_{i,j}^n + \Delta t^2 v_{i,j}^2 \left(\frac{4}{3}\nabla^2(1)p_{i,j}^n - \frac{1}{3}\nabla^2(2)p_{i,j}^n \right) + \text{R.H.S.}$$

where

$$\nabla^2(k)u_{i,j} \equiv \frac{u_{i+k,j} + u_{i-k,j} - 2u_{i,j}}{(k\Delta z)^2} + \frac{u_{i,j+k} + u_{i,j-k} - 2u_{i,j}}{(k\Delta x)^2}$$

approximates the Laplace operator on subgrids, and $v_{i,j} \simeq v(x_{\min} + j\Delta x, z_{\min} + i\Delta z)$. This stencil applies straightforwardly to the "interior" gridpoints ($1 < i < N_z - 1, 1 < j < N_x - 1$. On the two rows of gridpoints next to the boundary, we use the method of images to extend the field as an odd function, which is consistent with the Dirichlet problem for the wave equation as presented here. This trick maintains the full interior accuracy of the scheme at the boundary; we leave the detailed formulation to the reader.

A discrete representation of the source (right hand side) is obvious if the source coordinates (x_s, z_s) specify a grid point. However we did not want to constrain the placement of the source in such an artificial way, so we opted for source specification by *adjoint interpolation.* If $x_{\min} + j_s\Delta x \leq x_s \leq x_{\min} + (j_s + 1)\Delta x$ and similarly for z, we set

$$\begin{aligned}\text{R.H.S.} = \; & \frac{f^n}{\Delta z \Delta x}((1 - r_x)(1 - r_z)\delta_{i,i_s}\delta_{j,j_s} + \\ & r_x(1 - r_z)\delta_{i,i_s}\delta_{j,j_s+1} + (1 - r_x)r_z\delta_{i,i_s+1}\delta_{j,j_s} + r_x r_z \delta_{i,i_s+1}\delta_{j,j_s+1})\end{aligned}$$

where

$$r_z = \frac{z - (z_{\min} + i_s\Delta z)}{\Delta z}, \; r_x = \frac{x - (x_{\min} + j_s\Delta x)}{\Delta x}$$

This grid function, when discretely "integrated" against the sample array of a smooth function, produces a second order accurate (in Δz, Δx) approximation to the value of the function at (x_s, z_s) - this is simply the error in bilinear interpolation.

We also use bilinear interpolation to approximate the samples of the pressure field at arbitrary output locations, to record a synthetic seismogram.

4.2 Implementation

The Fortran subroutine `a2cptsrc`, which applies the finite difference stencil (i.e. evaluates $H_n(c, U^n)$), is the basis of our implementation. This routine takes as input some grid parameters, the array of velocity samples, the value $f(t_n)$ and arrays `p0` and `p1`, which hold p^{n-1} and p^n, respectively. The subroutine then overwrites `p0` with p^{n+1}.

The AD package TAMC [Giering 1999] was used to produce four further Fortran subroutines, implementing the linearizations of `a2c24ptsrc` with respect to the velocity and the pair (p^{n-1}, p^n), and the adjoints of these linearizations. A small amount of hand-editing the the TAMC output was necessary because TAMC does not always generate code for the desired operation (for example, all TAMC adjoint code adds the computed result to the output variable, which is not necessarily the desired action).

The class `a2c24StencilOp` was then defined to represent H_n. By way of example, we present the code for the method `Image`, which computes $H_n(c, U^n)$ and overwrites U^n with the result. Note that since the `fdtd` formalism requires a one-step scheme, the "state" U^n represents the pair (p^{n-1}, p^n), while U^{n+1} is (p^n, p^{n+1}). If the input to the Fortran routine `a2c24ptsrc` is (p^{n-1}, p^n), then the output is (p^{n+1}, p^n) (since the first array is overwritten with the result). Therefore, after the Fortran routine is called, the pointers to the internal vectors representing the two time levels are swapped.

Note also that `a2c24StencilOp` must be initialized with the "control" variable (the velocity) and the time level n before `Image` is invoked, and the the state variables are internal to the stencil operator class (that is, `a2c24StencilOp` manages the memory for these variables). All of the vectors in this problem are represented by instances of `SGFVector`, a concrete HCL vector class that implements functions sampled on a regular rectangular grid.

Here, then, is the code for `a2c24StencilOp::Image`. Note that the names of classes are appended with "`_d`", indicating that the classes are based on double precision arithmetic.

```
 void a2c24StencilOp_d::Image()
{
   if( !control )
   {
      cerr << "Error in a2c24StencilOp_d::Image: "
              "control vector has not been"
              " initialized" << endl;
      exit(1);
   }
   // The velocity field will be c,
   // and (u^{n-1},u^n) will be (u,v).
   SGFVector_d & c = (SGFVector_d&)(*control);
   SGFVector_d & u = (SGFVector_d&)*p0;
   SGFVector_d & v = (SGFVector_d&)*p1;

   // Now get the source data and apply the stencil.

   double fn = F->Data()[TimeLevel];
   F77NAME(a2c24ptsrc)( (int&)nx,(int&)nz,(double&)dx,
          (double&)dz,(double&)dt,c.Data(),fn,(int&)is,
          (int&)js,(double&)rz,(double&)rx,u.Data(),
```

```
              v.Data() );

        // Finally, switch the pointers to update the state vector

        SGFVector_d * tmp = p0;
        p0 = p1;
        p1 = tmp;
    }
```

The first five arguments of a2c24ptsrc pertain to the grid geometry, and are private data members of a2c24StencilOp, initialized on construction. Since private data cannot be altered except by class members, it is regarded as "constant" for outside code that accesses it explicitly. The Fortran time step subroutine is such "external" code. Since Fortran does not provide "constant" subroutine arguments, we must strip these parameters of their protection when passing them to Fortran. This is the function of the (int &) syntax, which "casts away" the constant nature of the arguments. Similar remarks apply to the arguments is, js, rz, and rx, which specify the location of the point source by grid cell and local (cell) coordinates. The data arrays of the velocity and the two state vectors are identified with c.Data(), etc.

The methods for implementing the other required operations ($D_c H_n(c, U^n)\delta c$, $D_U H_n(c, U^n)\delta U^n$, etc.) is similar.

All significant arithmetic in our implementation is confined to subprograms written in Fortran. This approach forces some architecture dependence into the code, including naming conventions for calling Fortran subroutines from C++. We write portable (if ugly) code by using the C preprocessor to define the correct logical name for each program unit, and encapsulate the appropriate (and sometimes quite complicated) linking rules in configuration makefile fragments. Thus the macro "F77NAME" applied to the name a2c24ptsrc.

The other important method for a2c24StencilOp is the constructor. This is the class method that tells the compiler how to create a variable of this type. In this case, the constructor must be provided with a description of the grid, the point source f, and the coordinates (x_s, z_s) of the source. The grid description determines the vector space to which the state U^n belongs; therefore, the grid is represented by a vector space class, namely, an instance of SGFSpace. The point source f is an instance of SGFVector. The a2c24StencilOp constructor records the various grid parameters for easy access, and creates the domain and range of the operator (recall that in the HCL framework, each operator class has methods Domain and Range). The range of H_n is $\mathcal{U} \times \mathcal{U}$, where $U^n \in \mathcal{U}$, and so the range is an instance of the class HCL_GenericProductSpace, an HCL class that represents products of arbitrary factors. Similarly, the domain is $\mathcal{C} \times (\mathcal{U} \times \mathcal{U})$ and is represented by another instance of HCL_GenericProductSpace.

The sampling operator for this application is based on a C++ class `TimeSample`, written to sample at arbitrary locations functions represented by the `SGFVector`. The sample locations need not lie on grid points, so bilinear interpolation is applied. The implementation of this class is straightforward and will not be presented here.

Here is most of a main program for performing a simulation using `fdtd` and `a2c24StencilOp`. The omitted code is similar to what is shown (extracting parameters from the space definitions, error-checking, etc.).

```
main( int argc,char ** argv )
{
   if ( argc != 2 )
   {
      cerr << "usage: seismogram.x jobfile" << endl;
      cerr << "The jobfile must contain the fields 'Velocity', "
              "'Source', 'DataSpace', and 'DataFile'" << endl;
      exit(1);
   }
   Table ParamTable( argv[1] );
   char name[81];
   if( ParamTable.GetValue( "Velocity",name ) )
   {
      cerr << "Error in seismogram.x: file "
           << argv[1] << " is missing the "
              "field 'Velocity'" << endl;
      exit(1);
   }
   SGFVector_d c( name );
   SGFSpace_d & grid = (SGFSpace_d&)c.Space();
   .
   .
   .
   if( ParamTable.GetValue( "Source",name ) )
   {
      cerr << "Error in seismogram.x: file "
           << argv[1] << " is missing the "
              "field 'Source'" << endl;
      exit(1);
   }
   SGFVector_d f( name );

   int dim;
   if( f.GetValue( "dim",dim ) )
   {
      cerr << "Error in seismogram.x: file "
           << name << " is missing the "
```

```
                "field 'dim'" << endl;
         exit(1);
      }
      if( dim != 1 )
      {
         cerr << "Error in seismogram.x: file "
              << name << " must contain a point "
                 "source" << endl;
         exit(1);
      }
      .
      .
      .
      // Create the stencil operator

      a2c24StencilOp_d H( &grid,&f,xs,zs );

      // Create the cauchydata

      HCL_GenericProductSpace_d U2( &grid,2 );
      HCL_GenericProductVector_d cauchydata( &U2 );
      cauchydata.Zero();

      // Create the sampling operator, which records a
      // time series at each receiver location defined
      // in DataSpace.  Note that since the "current"
      // time level in fdtd is really (p^{n-1},p^n),
      // we must first extract the second component p^n
      // and then pass it to the TimeSample operator.
      //  This is precisely the function of the class
      // SGFExtractAndSample.

      SGFExtractAndSample_d S( 2,&grid,2,&DataSpace );

      // Create the fdtd operator by passing it the stencil and
      // sampling operators, the Cauchy data, the data space,
      // and the number of time steps.

      fdtd_d G( &H,&S,&cauchydata,&DataSpace,nt-1 );
      int dp;
      if( ParamTable.GetValue( "DispFlag",dp ) )
         dp = 0;
      G.Parameters().PutValue( "DispFlag",dp );

      G.Image( c,Data );
   }
```

grid	FDTD (s)	Fortran (s)	ratio
161×161	8.244	7.960	1.0357
321×321	86.4233	85.9967	1.0050

Table 1. Times on a Sun SPARC ultra 60 workstation.

4.3 Efficiency

A natural question is whether the overhead of C++ and object-oriented programming is significant. Above we have stated that it is not, since most of the execution time will be spent in the Fortran subroutines which perform the core calculations. We now justify our claim in two ways: direct timing comparisons and profiling.

We implemented the leapfrog scheme described above in a Fortran subroutine `a2c24sim`; this routine performs a complete simulation (finite difference time-stepping plus sampling). We then "wrapped" it as an HCL operator class, so that the `Image` method, which accomplishes the same this as the `fdtd Image` method described above, does not involve any looping or other manipulations in C++—just a single call to a Fortran subroutine. We then compared the total time required for a simulation, using both codes (`fdtd` and `a2c24sim`). The results on a Sun SPARC Ultra 60 workstation are given in Table 1. The results suggest that the overhead is not more than a few percent, and that this overhead becomes completely negligible for a large grid. In particular, virtually no overhead would be expected for a 3D problem.

As a second method of assessing the efficiency of the `fdtd` framework, we profiled the execution of the `Image`, `DerivImage`, and `DerivAdjImage` methods of `fdtd` (that is, the computations of $G(c)$, $DG(c)\delta c$, and $DG(c)^*\delta D$). This was done using the GNU tool `gprof` with the code compiled under `g++` and `g77`. The results showed that in every case, and for a small grid (81×81) 98% or more of the total execution time was spent in the Fortran subroutines; the overhead of C++ was at most 2%. Note that `gprof` uses statistical sampling and the results are not perfectly reliable (and not necessarily completely compatible with direct timing results). Nonetheless, we believe that the combination of direct timing and profiling presented here provides conclusive evidence that the use of C++ (and specifically HCL) to manage the high-level manipulations does not imply a performance penalty.

5. CONCLUSIONS

We have outlined the construction of a C++ class `fdtd` implementing explicit time-stepping (marching) methods. We have also shown, for an example (2D acoustics) how to organize a specific finite difference scheme as a derived class, with essentially all of the floating point arithmetic written in Fortran. This mixed language approach retains the floating point efficiency of Fortran but provides the abstract interface of a C++ class. Other methods of efficient numerical kernel construction could be accomodated just as easily.

The abstract class approach has two principal advantages. First, it encodes the checkpointing scheme needed in the computation of adjoint maps, at the most abstract level. Thus this aspect of automatic differentiation, which is really beyond the capability of most contemporary AD packages to implement efficiently, is "hard wired", and AD can be used on the simpler component routines which are well within its effective scope. Second, the code produces an *operator object* which is input for a variety of optimization algorithms in the HCL package. Thus the design of **fdtd** should facilitate wide-ranging exploration of both physical modeling assumptions and optimization techniques for design, inverse, and control problems in wave propagation, fluid flow, image processing, and other applications.

REFERENCES

BISCHOF, C., CARLE, A., CORLISS, G., GRIEWANK, A., AND HOVLAND, P. 1992. Adifor: Generating derivative code from fortran programs. *Scientific Programming 1*, 1 – 29.

BLANCH, J. O., SYMES, W. W., AND VERSTEEG, R. 1998. A numerical study of linear viscoacoustic inversion. In R. G. KEYS AND D. J. FOSTER Eds., *Comparison of Seismic Inversion Methods on a Single Real Data Set*, pp. 13–44. Tulsa, Oklahoma, USA: Society of Exploration Geophysicists.

CLAERBOUT, J. F. 1992. *Earth Soundings Analysis: Processing versus Inversion*. Blackwell Scientific Publications, Boston.

COLEMAN, T., SANTOSA, F., AND VERMA, A. 2000. Efficient calculation of Jacobian and admoint vector products in the wave propagational inverse problem using automatic differentiation. *J. Comp. Phys. 157*, 234-255.

GIERING, R. 1999. *Tangent linear and Adjoint Model Compiler, Users manual 1.4*,

`http://puddle.mit.edu/~ralf/tamc`

GOCKENBACH, M. S., PETRO, M. J., AND SYMES, W. W. 1999. C++ classes for linking optimization with complex simulations. *ACM Transactions on Mathematical Software 25*, 2, 191–212.

GOCKENBACH, M. S. AND SYMES, W. W. 1996. The Hilbert Class Library: a library of abstract C++ classes for optimization and inversion. *Computers and Mathematics with Applications 32*, 1–13.

GRIEWANK, A. 1992. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software 1*, 35–54.

HANEY, S., CROTINGER, J., KARMESIN, S., AND SMITH, S. 1999. PETE: the portable expression template engine. *Dr. Dobb's Journal'* (No. 304), pp. 88 ff, October 1999.

LEVANDER, A. 1988. Fourth order finite difference P-SV seismograms. *Geophysics 53*, 1425–1434.