A Finite Difference Time Stepping Class

William W. Symes and Chaoming Zhang *

ABSTRACT

The abstract C++ class `FDTD` expresses the common characteristics of explicit finite difference schemes for initial boundary value problems. The advantages of abstract specification are especially apparent when linearized and adjoint simulation modules are required, as is the case for example when the simulator drives the optimization of fit error to solve a parameter estimation problem.

Introduction

Explicit finite difference schemes have several appealing properties for simulation of wave motion in materials with relatively simple boundaries but considerable internal heterogeneity, such as the Earth's crust. These time-stepping schemes have roughly the same sampling requirements in time and space, as do the waves that they model. They can accomodate (in principle) nearly any degree of heterogeneity, and their basic convergence behaviour is well-understood and easy to control (although some mysteries remain open on both counts). They are efficiently coded in Fortran, especially when their grids are logically rectangular, and such code 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 of waves ("inversion"). Linearization of the mappings defined by these schemes is relatively straightforward, and the adjoints of these linearized maps are efficiently computed by means of the so called ajoint state method. See (Claerbout, 1992) for the importance of adjoints in geophysical estimation.

The use of finite difference schemes as drivers in data fitting excercises 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

*The Rice Inversion Project, Dept. of Computational and Applied Mathematics, Rice University, Houston Texas 77005 USA

successfully coded in classic Fortran style, linkage to high-quality numerical optimization code is problematic. The optimization algorithms require a specific data structure, usually simple arrays, and the simulation of wave motion will generally require considerably more structure, eg. 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 nonreusable 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. 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` (stands for "finite difference time domain"). The class records for use under many circumstances the general structure of not just the basic time stepping scheme for a finite difference simulation, but also its linearized scheme 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 just the various modules related to single time steps. These are simple enough for most schemes that almost any modern automatic differentiation toolbox, for example the ADIFOR package (Bischof et al., 1992), 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 will be generated automatically and fit into prepared places in the `FDTD` template.

Furthermore, our finite difference class defines an *operator* in the sense of the *Hilbert Class Library* or *HCL*, a software package for optimization and related tasks written in C++ (Gockenbach and Symes, 1996a; Gockenbach and Symes, 1996b). HCL defines vectors, operators, functions, and associated mathematical objects as C++ objects, with the minimal structure proper to the concept in each case. It therefore becomes a platform for implementations of optimization and other algorithms with 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 finite difference scheme specialized ("derived") from `FDTD` is automatically an operator in the HCL sense, it can be used in any of the HCL optimization algorithms without any further code being written. 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 structure of 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 class `A2C` derived from

FDTD, implementing 2D linear acoustic wave simulation. It will be evident from the description that there is no need to give up Fortran to gain the advantages of object-oriented programming - indeed the core numerical routines of A2C are written in Fortran. Some of these could be the output of an automatic differentiation package such as ADIFOR. Moreover the mixed C++/Fortran code retains all of the advantages of Fortran compiler options and even parallelization, as we show by explicit example - because almost all of the floating point arithmetic is still handled in Fortran!

Abstract Formulation of Time Stepping Schemes and Associated Linearized and Adjoint Schemes

All schemes may be considered to be two-level, by concatenating several time levels if necessary. Denote by U^n the nth time level of this aggregate state vector, and by P the parameters of the scheme, i.e. coefficient arrays and so on. Part of P will play the role of the input vector of a (nonlinear) operator defined by the scheme, while the rest will represent parameters which are to be regarded as constants of the experiment being simulated. This distinction will be discussed in the next section.

The time step is Δt, the stencil operator is $S[P, U]$. We assume that S is *autonomous affine*: i.e. for a linear operator $L[P]$ and a vector $E[P]^n$,

$$U^{n+1} = U^n + \Delta t L[P] U^n + E[P]^n$$

This decision excludes nonlinear schemes such as flux-limiter methods (Zhang and LeVeque, ; Zhang, b; Zhang, a; Fei,). Thus the design of the class will eventually need to be extended to accomodate such schemes. The present design includes most of the widely used schemes, such as Lax-Wendroff and leapfrog variants.

In the target problems, the field U is not the primary output of the simulation; rather the output is the result of applying a sampling or observation operator S, which in the cases of immediate interest is linear, autonomous, and independent of the parameters:

$$F[P]^n = SU^n$$

The *linearization* of the map $P \mapsto F[P]$ is the result of first order perturbation of the time stepping equations:

$$(DF[P]\Delta P)^n = S\Delta U^n$$

where

$$\Delta U^{n+1} = \Delta U^n + \Delta t\{(DL[P]\Delta P)U^n + L[P]\Delta U^n\} + (DL[P]\Delta P)^n$$

In order to define the adjoint, we need norms in domain and range. The range norm for F will be given by the inner product

$$< F_1, F_2 >_R = \Delta t \sum_n < F_1^n, F_2^n >_T$$

where $< \cdot, \cdot >_T$ is an inner product defined on the range of S. Denote by $< \cdot, \cdot >_D$ an inner product for the input vector part of the parameters P, and by $< \cdot, \cdot >_U$ an inner product on the state vector space (ie. U^n). Then for a vector H in the range of F,

$$< H, DF[P]\Delta P >_D = \Delta t \sum_n < H^n, (DF[P]\Delta P)^n >_T$$

$$= \Delta t \sum_n < H^n, S\Delta U^n >_T = \Delta t \sum_n < S^* H^n, \Delta U^n >_U$$

Suppose that the field V^n solves the *adjoint state system*

$$V^{n-1} = V^n + \Delta t L[P]^* V^n + \Delta t S^* H^n.$$

where the adjoint $L[P]^*$ is with respect to whatever inner product is chosen on the state vector, i.e. $< \cdot, \cdot >_U$. Then the above is

$$= \sum_n < (V^{n-1} - V^n - \Delta t L[P]^* V^n), \Delta U^n >_U$$

$$= \sum_n < V^{n-1}, \Delta U^n >_U - \sum_n < (I + \Delta t L[P])^* V^n, \Delta U^n >_U$$

$$= \sum_n < V^n, \Delta U^{n+1} - \Delta U^n - \Delta t L[P] \Delta U^n >$$

(provided that $U^n = 0$ for $n \leq 0$ and $V^n = 0$ for $n >> 0$, both of which we assume)

$$= \sum_n < V^n, \Delta t (DL[P]\Delta P) U^n >$$

Denote by $B[P, \cdot, \cdot]$ the "adjoint" of the bilinear map

$$(\Delta P, U) \mapsto (DL[P]\Delta P)U$$

i.e.

$$< B[P, U, V], \Delta P >_D = < V, (DL[P]\Delta P)U >_U$$

for all U, V, ΔP. Then we have finally shown that

$$< H, DF[P]\Delta P >_R = < \Delta t \sum_n B[P, U^n, V^n], \Delta P >$$

for any H, which implies that

$$DF[P]^* H = \Delta t \sum_n B[P, U^n, V^n]$$

. Thus the procedure for the computation of the adjoint to the linearized map $DF[P]^*$ is:

1. solve the simulation problem to produce $\{U^n\}$
2. apply the adjoint sampling operator S^* to H

3. solve the adjoint state equation for $\{V^n\}$
4. accumulate the sum of the bilinear products $B[P, U^n, V^n]$

A logistical problem immediately asserts itself: U is produced by stepping *forward* in time, V 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 V. The sum of the bilinear products B is accumulated, and when $n = 0$ is reached the gradient 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 (Symes et al., 1995), should be 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.

Design of the Simulator Abstract Base Class `FDTDOp`

The simulators of interest to us implement *nonlinear operators*, with domains and ranges in various Hilbert spaces. Our intention is to construct the abstract description of time stepping schemes as a derived class of the basic HCL operator class `HCL_Op`, see (Gockenbach and Symes, 1996a; Gockenbach and Symes, 1996b). Besides methods which return the domain and range vector spaces, the principal method of the `HCL_Op` class is `Image`, which applies the operator to an input vector and produces an output vector. Thus we must decide on part of the parameter vector P of the previous section which will be treated as input vector, and reserve the rest as a (static) set of parameters. `HCL_Op` contains derived classes `HCL_OpDeriv`, `HCL_OpDerivAdj`, and `HCL_OpDerivNorm`, expressing operators with derivative operators, operators with derivative and adjoint derivative operators, and operators with derivative, adjoint derivative, and normal (derivative followed by its adjoint) operators respectively. These types will be required for optimization applications.

The standalone class `FDTD` is the foundation for our implementation of finite difference simulators. The base class `FDTDOp` is an interface to `FDTD`, derived from `HCL_Op`. Its constructor takes a pointer to an `FDTD` as argument, and stores a copy `F` of this pointer as private data. Its `Image` method simply calls the image method from `FDTD`:

```
  virtual void Image( const HCL_Vector & x, HCL_Vector & y ) const {
    F->Image(x, y);
  }
```

The `FDTD` class is abstract, as it contains pure virtual methods. The time stepping loop however is concrete. The class does not have a useful constructor, but doesn't need one since it is abstract. Derived concrete classes for specific simulators should be provided with appropriate constructors which initialize all workspace needed in the finite difference calculations as private data. including grid geometry definition, allocation of memory for whatever grid sample arrays are needed to define U and whatever work arrays are needed by the stencil. The constructor for a concrete derived class will express the division of data into parameters and input vector, as discussed above: those inputs regarded as parameters will appear as arguments to the constructor, and scalars and arrays derived from them will be initialized by the constructor. It should *not* initialize those arrays which are part of the input vector x. The initializations depending on x must be part of the `Image` method.

An example of such a constructor is described below in the section on the 2D acoustic model.

The `Image` method is concrete, apart from the use of several abstract components. Here is an abbreviated version of the source listing, with error handling code deleted; a full listing is in the appendix.

```
  virtual void FDTD::Image( const HCL_Vector & x, HCL_Vector &  y ) {
#   (code to check that x and y are in appropriate spaces)
    Input(x);
    int nt = NumTimeSteps(); int it;
    for ( it = 0; it < nt; it++ ) {
      Stencil(it);
      RightHandSide(it);
      Sample(it);
      Update(it);
    }
    Output(y);
  }
```

This simple method will need some elaboration, but already embodies the basic framework of the class. The constructor performs all initializations not related to the input vector. The `Input(x)` method does the rest.

The methods used in the `Image` method are

- `virtual int Input( HCL_Vector & x )`: initializes arrays from the input vector; includes any necessary i/o and interpolation onto the finite difference grid set up by the constructor.
- `virtual int Stencil(it)`: implements

  $$U^+ \leftarrow U^c + \Delta t L[P] U^c$$

  Since U itself is not the output vector of the method, there is no need to keep its entire time history. Formally, two time levels ("current" U^c and "next" U^+) are enough. These could be initialized in the constructor or in the `Input` method; in fact the latter is more useful, as becomes apparent in the extension to multiexperiment simulators, see below. There is no need in this design to carry out explicitly the reduction to two level scheme: the update formula may be left in multilevel form if that is natural (as it is, for example, with leapfrog schemes), with the translation to two-level scheme implicit. That's an advantage of information hiding, one of the features of object oriented design.
- `virtual int RightHandSide(it)`: implements

  $$U^+ \leftarrow U^+ + E[P]^c$$

  where $E[P]^c$ is the current value of the right hand side vector.
- `virtual int Sample(it)`: stores SU^c in an appropriate array (or file, if necessary - this is not anticipated!) for $F[P]^n$, in the notation of the last section.
- `virtual int Update(it)`: switches U^c and U^+ and updates $E[P]^c$, along with any other internal arrays that need updating for the next time step. In practice these updates will be performed by pointer exchange, so that no data is actually moved.
- `virtual int Output( HCL_Vector & y )`: writes the internal storage of the sampled output into the vector `y`, representing $F[P]$, in whatever way is appropriate to its type. If `y` is an out of core object (as will typically be the case, for instance an `SGF`), then this involves disk i/o. It may also involve interpolation if the finite difference and output grids are different, as is usually the case.

Note that `Stencil` may in fact do all of the work of a single step, obviating the need for `RightHandSide`, `Sample`, and `Update`. In that case the latter three methods are implemented as no-ops.

It's tempting to make U^n into a `HCL_Vector`, and use the linear algebra formalism implemented in the Hilbert Class Library. However in the interest of efficiency, we elect to embed the linear algebra in the member functions of the `FDTD` class. These will usually be written in Fortran and optimize better on most platforms than would their C++ equivalents.

Abstract Classes for Derivatives: `FDTDOpDeriv`, `FDTDOpDerivAdj`, and `FDTDOpDerivNorm`

These are derived from the corresponding classes in `HCL_Op`. The principal additional method in each case returns a parameterized linear operator. For example the derivative expresses $DF[P]\Delta P$ in the notation of the first section. This is a linear operator on ΔP parameterized by P. The `HCL_ParamLinearOp` class was invented to describe such situations. Its principal method is `HCL_LinearOp * Forward(const HCL_Vector & x)`, which returns a pointer to a linear operator, in this case $DF[P]$. The linear operator class `HCL_LinearOp` has an `Image` method which applies the linear operator.

For the (forward) derivative, the basis of this somewhat ramified structure is a standalone class `FDTDDeriv`. It looks much like `FDTDOp` but has an additional method `DerivImage`, which encodes the timestepping loop for the derivative operator. This method is concrete, and looks much like `Image`:

```
  virtual void FDTDDeriv::DerivImage( const HCL_Vector & x,
                                      const HCL_Vector & dx,
                                      HCL_Vector &  y ) {
#   (code to check that x, dx, and y are in appropriate spaces)
    DerivInput(x,dx);
    int nt = NumTimeSteps(); int it;
    for ( it = 0; it < nt; it++ ) {
      DerivStencil(it);
      DerivRightHandSide(it);
      DerivSample(it);
      DerivUpdate(it);
    }
    DerivOutput(y);
  }
```

The additional pure virtual methods `DerivInput`, `DerivStencil`, `DerivRightHandSide`, `DerivSample`, `DerivUpdate`, and `DerivOutput` serve similar functions to those used in the `Image` method, and in fact in some cases they are the same: in that case no new code need be written. An example of this sort of economy occurs in the 2D acoustic simulator, as explained below.

The `FDTDOpDeriv` class is naturally built out of several auxiliary classes, beginning with `FDTDDeriv`.

The class `FDTDDerivLinOp` is derived from `HCL_LinearOp`, and expresses the derivative image as a linear operator. Its constructor takes a `HCL_Vector` (representing P) and an `FDTDDeriv` object (DF) as arguments, and stores pointers to these (`x`, `DF`) as private data.

Its `Image` method calls the `DerivImage` method of the `FDTDDeriv` object to implement $\Delta P \mapsto DF(P)\Delta P$:

```
virtual void Image( const HCL_Vector & dx, HCL_Vector & y ) const {
  DF->DerivImage(*x,dx,y);
}
```

The class `FDTDDerivParamLinOp` is derived from `HCL_ParamLinearOp`. Its constructor takes an `FDTDDeriv` object as its argument, and stores a pointer to this object as private data. Its `Forward` method takes an `HCL_Vector` as argument, and returns a pointer to an `FDTDDerivLinOp` object:

```
virtual HCL_LinearOp * Forward( const HCL_Vector & x ) const {
  FDTDDerivLinOp * A = new FDTDDerivLinOp( x, DF );
  return A;
}
```

Finally the `FDTDOpDeriv` class is derived from `HCL_OpDeriv`, as desired. Its constructor takes an `FDTDDeriv` object as its argument, and stores a pointer `F` to this object as private data, as did the previous two classes. Its `Image` method simply calls the `Image` method from `F`, and its `Deriv` method returns a pointer to a `FDTDDerivParamLinOp`:

```
virtual HCL_ParamLinearOp * Deriv( ) const {
  FDTDDerivParamLinOp DF = new FDTDDerivParamLinOp( F );
  return DF;
}
```

A Note concerning Memory Management: The HCL convention is that methods return *references* to objects when no dynamic memory allocation occurs in their construction. HCL methods return *pointers* when `new` appears somewhere in the code. Thus the programmer knows to delete any pointers returned by HCL methods. For example any code which accesses the `Deriv` method of `FDTDOpDeriv` should delete the pointer it returns when no longer needed.

The adjoint of the linearized map bifurcates according to the scheme used to access the background field, as explained in the first section. Accordingly we define two standalone foundation classes: `FDTDDerivAdjC` and `FDTDDerivAdjD`, with the final `C` standing for "conservative" and the `D` for "dissipative".

The adjoint image method for `FDTDDerivAdjC`, stripped to bare essentials, is

```
  virtual void FDTDDerivAdjC::AdjImage( const HCL_Vector & x,
                                        const HCL_Vector & d,
                                        HCL_Vector &  y ) {
#   (code to check that x, d, and y are in appropriate spaces)
    AdjInput(x, d);
    int nt = NumTimeSteps(); int it;
    for ( it = 0; it < nt; it++ ) {
      Stencil(it);
      RightHandSide(it);
      Update(it);
    }
    for ( it = nt-1; it >= 0; it-- ) {
      // note that these stencils contain the background
      // scheme as well - i.e. the adjoint and background
      // fields are marched backwards in time simultaneously
      AdjStencil(it);
      AdjRightHandSide(it);
      AdjAccumulate(it);
      AdjUpdate(it);
    }
    AdjOutput(y);
  }
```

In some cases one can take advantage of symmetry in the basic marching scheme, to use the same stencil to march the background fields backwards in time as forwards. Of course this works well only if the scheme is conservative, and not at all if it is significantly dissipative. An example is presented in the next section.

The adjoint image method for `FDTDDerivAdjD` is somewhat more complex. The computation proceeds in *sweeps*: in each sweep, the reference field is computed up to a certain number of *checkpoints*, at each of which the background field is saved. At the final checkpoint in each sweep, the adjoint for the remaining steps is accumulated by recomputing the background field for each step. Then the next sweep does one fewer recomputation, and so on. The following code has in it one method which will be concrete, namely `CheckPoints`, which implements the Griewank checkpointing algorithm (Griewank, 1992). This algorithm is intricate, and its explanation goes beyond the scope of this article. We refer the reader to the original reference for a description.

```
  virtual void FDTDDerivAdjD::AdjImage( const HCL_Vector & x,
                                        const HCL_Vector & d,
                                        HCL_Vector &  y ) {
#   (code to check that x, d, and y are in appropriate spaces)
    AdjInput(x, d);
```

```
int nt = NumTimeSteps(); int it; int it1;
int ncp;           // number of checkpoints (saved data)
int cpl[ncp];      // time levels of checkpoints
int saved[nt];     // = 1 if level is saved, = 0 else
int icp;           // counter for checkpts
int ccpl;
int nleft=nt;
for (it=0;it<nt;it++) {
  saved[it]=0;
}
Save(0,0);
saved[0]=1;
while (nleft > 0) {
  // calculate checkpoints
  CheckPoints(nleft,cpl);
  for (icp = 0; icp < ncp-1; icp++ ) {
    // for each checkpoint compute and save if not already
    if (!saved[cpl[icp+1]]) {
      for (it=cpl[icp]+1;it<=cpl[icp+1];i++) {
        Stencil(it);
        RightHandSide(it);
        Update(it);
      }
      Save(icp+1); // save current Cauchy data
      saved[it]=1; // flag save
    }
  }
  // accumulate adjoint from last checkpoint to nleft
  ccpl=cpl[ncp-1];
  for (it = nleft; it > ccpl; it--) {
    Retrieve(ncp-1); // load Cauchy data at last checkpoint
    for (it1 = ccpl; it1 < it; it++) {
      Stencil(it);
      RightHandSide(it);
      Update(it);
    }
    // note that these are "pure adjoint" stencils, i.e. do not
    // march the background field backwards, in contrast to those
    // used in the conservative adjoint code
    AdjStencil(it);
    AdjRightHandSide(it);
    AdjAccumulate(it);
    AdjUpdate(it);
```

```
    }
    nleft=ccpl;
  }
  AdjOutput(y);
}
```

The Concrete Derived Class `a2c`: (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

$$\text{R.H.S.} =$$

$$\frac{f^n}{\Delta z \Delta x}((1-r_x)(1-r_z)\delta_{i,i_s}\delta_{j,j_s} + r_x(1-r_z)\delta_{i,i_s}\delta_{j,j_s+1} + (1-r_x)r_z\delta_{i,i_s+1}\delta_{j,j_s} + r_x r_z \delta_{i,i_s+1}\delta_{j,j_s+1})$$

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.

Fortran source for the finite difference stencil, evaluation of the right hand side, and sampling forms the substrate of our implementation. The class definition file for `FDTD` specifies the necessary member functions which must be implemented to make a concrete derived class. The most complex is the class constructor `A2C`, while the `Input` and `Stencil` functions are typical of the rest. Therefore we discuss these three in detail.

Since `A2C` implements an operator from one vector space (of velocity grid functions) to another (of seismogram grid functions), its domain and range are necessary components of its definition. The Hilbert Class Library makes the specification of domains and ranges and the like possible through the provision of *vector space* classes. That is, a vector space is an object in the HCL framework. Vector spaces at their most abstract have only two methods: they can be compared, through an overloaded use of the comparison operator

(`==`), and they can return member vectors. This level of abstraction is implemented in the `HCL_VectorSpace` class. It is companion to the basic abstract vector class `HCL_Vector`, and indeed every concrete HCL vector class comes as a pair - vector spaces defined in one class, vectors in the other.

We use a concrete `HCL_Vector` class to define the inputs and outputs for finite difference codes: indeed this *Sampled Grid Function* class was designed precisely with this application in mind. Besides the methods inherited from `HCL_VectorSpace`, `SGFSpace` has a `Parameters` method which returns a table of defining geometrical information about the grid. The grids in question are assumed to be uniform and hypercubical, as is the case for the simple finite difference application described here.

The constructor therefore takes as arguments the vector spaces defining the input and output grids. Since the source time series is regarded as a parameter in this design, it is also an argument of the constructor, as is another table specifying other parameters not defined by the input and output spaces, such as debugging and verbosity flags:

```
  A2C( SGFSpace * VelocityGrid, SGFSpace * SourceReceiverGeometry,
       SGF * SourceTimeSeries, table * JobTable );
```

The principal role of the constructor in our design is to initialize the scalars and arrays needed in the calls to the various Fortran subroutines which implement the finite difference scheme. These scalars are *private data* of the class, i.e. are not directly accessible to other program components. For example the number of grid points in the x direction, N_x, is such private data. So the following appears in the class definition:

```
class A2C: public FDTD {
private:
....
  int nx;                   // number of horizontal space steps
....
public:
....
};
```

The constructor initializes `nx` by reading its value from the parameter table of the `SGFSpace` defining the input velocity grid (`VelocityGrid`). Similarly the other scalars characterizing the grid are read from the input vector space objects, and auxiliary scalars computed, either inline or through calls to Fortran subroutines as is convenient. For example, the time origin of the source time series need not be $t_{\min}$, the initial time for the output recording. Thus the total number of finite difference steps may be larger than the total number of time samples in the output, and so must be computed from the inputs.

The constructor also allocates memory to hold the pressure sample and other arrays appearing in the finite difference scheme. The velocity array cannot be initialized on construction, as the velocity is the *input vector* hence is not accessed until the `Image` method is invoked. However the source time series can be initialized, as can the arrays containing the bilinear interpolation coefficients for the source and receivers.

The `Image` method of the `FDTD` class is invoked as the method of the same name for `A2C`, as it is concrete (modulo the component methods, which are defined in `A2C`, and no method in `A2C` is defined to override it. The first task of this method is to check that the input vector belongs to the domain of the operator, which was recorded during construction. This query is possible, as HCL provides each vector with a method which returns the space to which it belongs, as well as a comparison operator for vector spaces.

The next step in the `Image` method is the call to `Input`, whose function is to encapsulate the i/o necessary to read the input velocity array from disk into the buffer allocated to hold it during construction. Since only the square of the velocity actually appears in the finite difference formula, the sample values are squared at this time, to avoid redundant arithmetic later.

One of the overarching principles of the design is to carry out each step in the computation as early as possible. For example once the velocity array is available the stability conditions of the scheme can be checked; we carry this check out with a call to a Fortran subroutine in the `Input` method, as soon as the array is read into core.

The `Stencil` method carries out a single time step, less the addition of the right hand side. The entire source is quite simple, and we reproduce it here:

```
int A2C::Stencil(int it) const {
  int ier=0;
  fdstep_((int &)nz,(int &)nx,(float &)dz,(float &)dx,(float &)dt,
  velbuf,&u[curr],&u[prev],ier);
  return ier;
}
```

Thus this method is simply an interface or "wrapper" for the Fortran subroutine `fdstep`, which actually does all of the arithmetic. The first five arguments of `fdstep` pertain to the grid geometry, and are private data members of `A2C`, 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.

The sixth argument is a `float *` pointing to the beginning of the velocity array. Such things are exactly the C/C++ translation of Fortran arrays, whereas the references (`&`) are translations of scalars. The next two arguments are pointers to the two segments of the buffer allocated to hold the pressure grid, which are switched at each time step. Every method returns an integer error flag in the `FDTD` design. Thus the last argument is a Fortran error flag, which is passed back as the return value, since nothing else happens in this method.

Thus all significant arithmetic is confined in our implementation to subprograms written in Fortran. This approach forces some architecture dependence into the code. For example the logical name convention used above, of adding an underscore after the name of the Fortran subroutine when invoking it from C++, is correct for current Unix OS from Sun Microsystems and Silicon Graphics, and for Linux (using `f2c`), but incorrect for AIX from IBM (for which no underscore should be added), and also for Cray Unicos and HPUX (for which the subroutine name should be capitalized, with no underscore). We write "robust" (if ugly) versions of the 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.

The reward for this trouble is easy access to the many optimizations offered by modern Fortran compilers. For example, we compared the `A2C` implementation of FDTD with a pure Fortran implementation in a linearized forward map problem with a 357×784 grid and 2850 time steps. Both codes were compiled on an SGI PowerChallenge 75 MHz R8000 machine with 4 CPUs using the O3 optimization level and the parallelizing Fortran compiler (`-pfa` compiler flag). The `A2C` implementation completed (including i/o) in 50 s (wallclock time), the "pure Fortran" in 55 s. The discrepancy appeared to be due mostly to data handling and i/o, which is done somewhat more efficiently in the `A2C` implementation. When the `A2C` code was recompiled without the parallelizing compiler, the time for the same job increased by a factor of 5, to 263 s. Thus the parallel speedup was superlinear, as is frequently observed on this machine due to its cache design (in parallel mode the caches for the individual processors are pooled, resulting in a substantially higher cache hit rate). In any case `A2C` achieved roughly 600 Mflops in parallel execution, representing 50% efficiency for this machine. Clearly the mixed language design did not get seriously in the way of optimization and parallelization in this example.

The derivative and adjoint classes are built in a similar way: constructors initialize private data encompassing all of the necessary finite difference parameters, stencil and other methods functioning as "wrappers" for Fortran subroutines. This design makes application of automatic differentiation to create the Fortran for the derivative and adjoint classes very straightforward. Contemporary AD tools are not able to treat reliably problems of the complexity of full finite difference time stepping schemes. However they are entirely capable of generating source for derivatives of simpler functions, such as the finite difference stencil presented here. We have used both hand-coded and automatic differentiation produced derivative stencils in `A2C`. We used ADIFOR (Bischof et al., 1992) to produce the automatic derivative code by Fortran source transformation. The performance of automatically coded components was virtually identical to that of the hand-coded components. The design of `FDTD` opens the possibilty of extended automatic code generators based on tools like ADIFOR: the simulator author need supply only the subroutines implementing the stencil and other components of the time-stepping method, with all other code being supplied automatically, through simple scripts which set up the derived class from `FDTD`, and through application of ADIFOR and similar AD tools to create derivative and adjoint code. The authors hope to address this possibility in a future publication.

Conclusion

We have outlined the construction of an abstract C++ class `FDTD` defining explicity finite difference time stepping methods. In at least one simple example (2D acoustics) we have shown how to organize of 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. The abstract class approach has two principal advantages. First, it encodes already 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 it 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 inverse problems in wave propagation.

Acknowledgement and Disclaimer: This work was partially supported by the National Science Foundation under grant numbers DMS 9404283 and DMS 9627355; the Office of Naval Research under grant number N00014-96-1-0156; the Air Force Office of Scientific Research, Air Force Materials Command, USAF, under grant number F49620-95-1-0339; and The Rice Inversion Project. TRIP Sponsors for 1997 are Advance Geophysical, Amoco Production Co., Conoco Inc., Cray Research Inc., Discovery Bay, Exxon Production Research Co., Interactive Network Technologies, Mobil Research and Development Corp., and Shell International Research. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, the Office of Naval Research, the National Science Foundation, or the US Government.

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.

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

Fei, T. Elimination of numerical dispersion in finite-difference modeling and migration by flux-corrected transport. *PhD Dissertation*, Colorado School of Mines, 1994.

Gockenbach, M. S. and Symes, W. W. (1996a). The Hilbert Class Library: a library of abstract C++ classes for optimization and inversion. *Computers and Mathematics with Applications*, 32.

Gockenbach, M. S. and Symes, W. W. (1996b). The Hilbert Class Library: a library of abstract C++ classes for optimization and inversion. In *Fourth Object Oriented Numerics Conference: OON-SCI'96 Proceedings.*

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

Levander, A. (1988). Fourth order finite difference P-SV seismograms. *Geophysics*, 53:1425–1434.

Symes, W., Blanch, J., and Versteeg, R. (1995). A numerical study of linear inversion in layered viscoacoustic media. In Keys, R. and Foster, D., editors, *Comparison of Seismic Inversion Methods on a Single Real Dataset.* Society of Exploration Geophysicists. In press.

Zhang, C. The immersed interface methods for elastic wave propagation in heterogeneous materials. CAAM TR 96-29, Rice University, submitted to *Wave Motion.*

Zhang, C. Immersed interface methods for hyperbolic systems of partial differential equations with discontinuous coefficients. *PhD Dissertation*, University of Washington, 1996
(`ftp://amath.washington.edu/pub/chaoming/dissertation.ps.Z`).

Zhang, C. and LeVeque, R. J. The immersed interface method for acoustic wave equations with discontinuous coefficients. Accepted by *Wave Motion.*
(`ftp://amath.washington.edu/pub/rjl/papers/rjl-zhang:acou.ps.Z`).