Stanford Exploration Project, Report 89, November 12, 1997, pages 193–203

The adjoint of the viscous wave equation

Jon Claerbout[1]

keywords: *velocity, heat, sources, heat flow, wave equation*

ABSTRACT

The scalar wave equation is a linear operator whose output is any subset of the wavefield solution and whose input is the source function. Here we show how to compute the adjoint operator, i.e., that whose input is the wavefield subset and whose output is the sources.

INTRODUCTION

Mathematical physics presents us with many partial differential equations, the heat flow equation, the acoustic wave equation, seismic, electromagnetic, and others, many whose essential expression is found in the viscous scalar wave equation. Concepts of imaging tell us to identify a linear operator, essentially a matrix, and then use its transpose (adjoint) to make an image.

This paper shows how to build the adjoint of the viscous wave equation from its finite-difference representation. Inversion techniques use the operator and its adjoint to solve a wide variety of problems. Here we cover basic principles only, up to and including the stage of coding the operator and its adjoint and proving the adjoint's precision to be "full word".

DIFFERENTIAL EQUATIONS, CONVOLUTION, AND ADJOINTS

Simple one-dimensional equations of physics (such as vibrating strings) when discretized become tridiagonal matrices, each row with basically the same three elements (that arise from the expression of the second derivative operator). Where material properties are constant, the three elements are exactly the same from row to row. Likewise the convolution matrix representing filtering has rows identical to each other, except for the shift that aligns them along the diagonal. Examine a typical

[1]**email:** jon@SEP.Stanford.EDU

row of any such matrix.

$$\begin{bmatrix} a & b & c & & & \\ & a & b & c & & \\ & & a & b & c & \\ & & & a & b & c \end{bmatrix}' = \begin{bmatrix} a & & & \\ b & a & & \\ c & b & a & \\ & c & b & a \\ & & c & b \\ & & & c \end{bmatrix} \tag{1}$$

Notice that transposing such a matrix reverses the elements in the row. The reversal implies that the adjoint of convolution is crosscorrelation. Because of the reversal, the transpose of the first derivative matrix is the negative of the first derivative matrix itself (except at the edges). Generalizing, the discrete forms of differential equations representing physics in Cartesian coordinates show that any one-dimensional differential equation is its own adjoint if we set aside complications of boundaries, axis polarity, and material property variations. We will see something similar in higher dimensional spaces.

Basic ideas and notation

The basic idea of adjoint is that the adjoint of a row vector is a column vector. So the adjoint of the equation $y = ax_1 + bx_2$ is two equations, $\hat{x}_1 = ay$ and $\hat{x}_2 = by$.

Since we often combine operators by addition, we often code them in the form $y \leftarrow y + ax_1 + bx_2$. A convenient notation comes from the C computer language where such assignments are written as `y += a*x1+b*x2`. I find the "+=" notation expecially convenient for describing adjoints.

HEAT FLOW EQUATION

The one-dimensional heat flow equation is

$$\frac{\partial q}{\partial t} \quad = \quad \sigma \frac{\partial^2 q}{\partial x^2} \ + \ s(x,t) \tag{2}$$

where q is temperature, σ is heat conductivity divided by heat capacity and s is a possible source of heat. Discretizing equation (2) with the usual explicit finite difference operator gives

$$q_{t+1}^x \quad += \quad q_t^x \ + \ \sigma(q_t^{x-1} - 2q_t^x + q_t^{x+1}) \ + \ s_t^x \tag{3}$$

where t takes integer values, σ is like the σ in the differential equation but here it includes $\Delta t / \Delta x^2$. Ignoring the source s_t^x, the adjoint of (3) is

$$\begin{array}{lcr} q_t^x & += & q_{t+1}^x \\ q_t^{x-1} & += & \sigma q_{t+1}^x \\ q_t^x & += & -2\sigma q_{t+1}^x \\ q_t^{x+1} & += & \sigma q_{t+1}^x \end{array} \tag{4}$$

Below is a Ratfor subroutine that transforms between q_t and q_{t+1}, denoted by `qq()` and `rr()`. (Unfortunately Ratfor does not admit the $+=$ notation.)

```
subroutine heatstep( adj, add, sigma, qq,nx,  rr   )
integer              adj, add,           nx,      x
real                           sigma, qq(nx), rr(nx)
call        adjnull( adj, add,        qq,nx,  rr,nx)
do x= 2, nx-1 {
        if( adj == 0) {
                rr(x) = rr(x) + qq(x) + ( qq(x-1) - 2*qq(x) + qq(x+1)) * sigma
        } else {
                qq(x  ) = qq(x)   + rr(x)
                qq(x-1) = qq(x-1) + rr(x) * sigma
                qq(x  ) = qq(x  ) - rr(x) * sigma * 2
                qq(x+1) = qq(x+1) + rr(x) * sigma
                }
        }
return; end
```

Let us think about the adjoint calculation. It says we can consider in turn each temperature value q^x_{t+1}. From this value we augment (or diminish) neighboring temperatures according to the adjoint prescription. Strangely, for constant σ and far from boundaries, this adjoint process is equivalent to the usual heat flow process itself (but backwards in time). The reason it is equivalent is that each process, forward and adjoint, is the same matrix multiplication, the forward grouped by rows and the adjoint grouped by columns.

Sources for the heat equation and its adjoint

Above shows how a heat flow operator transforms the input q_t to the output q_{t+1}. What we really want to do is completely different. We want to define the input as the source function and we want to define the output as some arbitrary subset of q_t, q_{t+1}, q_{t+2}, etc. The adjoint we seek will carry us from the specified subset of q_t back to an image $\hat{s}_t$ of the sources s_t.

Defining $H = [1 + \sigma delta_{xx}]$ (where $delta_{xx}$ is the second difference operator) reduces equation (3) to

$$q_t \quad = \quad H_t q_{t-1} + s_t \tag{5}$$

where each q_t is a function of x. Recursive use of equation (5) over three time levels gives the matrix form

$$\mathbf{H}^{-1}\mathbf{q} \quad = \quad \begin{bmatrix} 1 & . & . & . \\ -H_1 & 1 & . & . \\ . & -H_2 & 1 & . \\ . & . & -H_3 & 1 \end{bmatrix} \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \quad = \quad \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix} \quad = \quad \mathbf{s} \tag{6}$$

A recursive solution begins at the top with $q_0 = s_0$ and propagates downward. The idea for computing is first to load up q_t with the sources $q_t \leftarrow s_t$ and then add in the heat-flow equation solution as you move along with

$$q_t \quad += \quad H_t q_{t-1} \qquad \text{increasing } t \tag{7}$$

The adjoint of H is H'. We define the image of the sources $\tilde{s}_t$ by the recursion

$$\tilde{s}_{t-1} \quad = \quad H'_t \; \tilde{s}_t + q_{t-1} \qquad \text{decreasing } t \tag{8}$$

which can be summarized as

$$(\mathbf{H}^{-1})'\tilde{\mathbf{s}} \quad = \quad \begin{bmatrix} 1 & -H'_1 & . & . \\ . & 1 & -H'_2 & . \\ . & . & 1 & -H'_3 \\ . & . & . & 1 \end{bmatrix} \begin{bmatrix} \tilde{s}_0 \\ \tilde{s}_1 \\ \tilde{s}_2 \\ \tilde{s}_3 \end{bmatrix} \quad = \quad \begin{bmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \end{bmatrix} \quad = \quad \mathbf{q} \tag{9}$$

Equation (6) implies $\mathbf{q} = \mathbf{Hs}$ and equation (9) implies $\tilde{\mathbf{s}} = ((\mathbf{H}^{-1})')^{-1}\mathbf{q}$. Basic mathematics tells us that $(\mathbf{H}^{-1})' = (\mathbf{H}')^{-1}$ which means that $((\mathbf{H}^{-1})')^{-1} = \mathbf{H}'$ so equation (9) implies $\tilde{\mathbf{s}} = \mathbf{H}'\mathbf{q}$. Thus the operators defined by recursive solutions of equations (6) and (9) are adjoints.

As before, the computing methodology is to preload $\tilde{s}_t \leftarrow q_t$ and then use C-style augmenting assignments

$$\tilde{s}_{t-1} \quad += \quad H'_t \; \tilde{s}_t \tag{10}$$

```
subroutine heatsteps( adj, sigma, ss, nx,nt,  qq )
integer              adj,           nx,nt,         it
real                        sigma, ss( nx,nt), qq( nx,nt)
temporary real                     tt( nx,nt)
call    adjnull( adj, 0,           ss, nx*nt,  qq, nx*nt)
if( adj == 0) {
        call copy( nx*nt, ss, tt)
        do it= 2, nt
                call heatstep( 0,1, sigma, tt(1,it-1), nx, tt(1,it))
        call copy( nx*nt, tt, qq)
} else {
        call copy( nx*nt, qq, tt)
        do it= nt,2,-1
                call heatstep( 1,1, sigma, tt(1,it-1), nx, tt(1,it))
        call copy( nx*nt, tt, ss)
        }
return; end
```

The problem with the above subroutine is that it wastes memory, assuming that we can store the temperature field over all space and time. A more realistic assumption is that we can store it over all space but only at two time levels. Moving the `copy` commands into the inner loops, we require memory for only the field at time t and time $t+1$.

```
subroutine heatsteps( adj, sigma, ss, nx,nt,  qq )
integer               adj,           nx,nt,        it
real                       sigma, ss( nx,nt), qq( nx,nt)
temporary real                                          tt( nx)
call    adjnull( adj, 0,          ss, nx*nt,  qq, nx*nt)
if( adj == 0)  {
                call copy( nx, ss(1, 1), qq(1,1))              # load initial
        do it= 2, nt {
                call copy( nx, ss(1,it),                   tt) # load source
                call heatstep( 0,1, sigma, qq(1,it-1), nx, tt)
                call copy( nx, tt,        qq(1,it))            # extract
                }
} else {
                call copy( nx,   qq(1,nt),        ss(1,nt))   # load initial
        do it= nt-1,1,-1 {
                call copy( nx,   qq(1,it), tt)                # load
                call heatstep( 1,1, sigma, tt, nx, ss(1,it+1))
                call copy( nx,   tt,              ss(1,it))   # extract source
                }
        }
return; end
```

Notice that the input `ss(,)` and output `qq(,)` still fill all of time and space so they appear to be impractically large. What saves us is that in practice there will not be sources everywhere in time and space nor will there be recording of the thermal field everywhere either. Thus in practice the `copy` calls will be reduced to copying the application operator inputs and outputs which should be much smaller.

WAVE PROCESS CHAIN

We have seen how the heat flow operator H marches one step in time. Next we will find a pressure operator P and a velocity operator V. We will represent the viscous scalar wave equation by the product HVP so the adjoint will obviously be $P'V'H'$. The product concatenation is done by subroutine `wavecat()` below:

```
# Wave operator by concatenation of pressure, velocity, and heat-flow equations
#
subroutine wavecat( adj, add, sigma,kappa,ru,rv, q,nx,ny,    r )
integer            adj, add,                      nx,ny
real    sigma, kappa(nx,ny),ru(nx,ny),rv(nx,ny), q(nx,ny,3), r(nx,ny,3)
temporary real                                   s(nx,ny,3), t(nx,ny,3)
call       adjnull( adj, add,                    q,nx*ny*3,  r,nx*ny*3)
if( adj == 0) {
        call pressure ( 0, 0, kappa,  q, nx,ny, s )
        call velocity ( 0, 0, ru,rv,  s, nx,ny, t )
        call viscosity( 0, 1, sigma,  t, nx,ny, r )
} else {
        call viscosity( 1, 0, sigma,  t, nx,ny, r )
        call velocity ( 1, 0, ru,rv,  s, nx,ny, t )
        call pressure ( 1, 1, kappa,  q, nx,ny, s )
```

```
        }
return; end
```

The pressure operator

The pressure operator from physics from the "equation of state" that the time rate of change of the pressure is an external source $s(x,y,t)$ minus the divergence of the velocity times the incompressibility K.

$$\frac{\partial p}{\partial t} \quad = \quad s(x,y,t) \; - \; K\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) \tag{11}$$

In the heat flow equation the field variable is a scalar (the temperature). For wave propagation in two dimensions, the wavefield variable has three components, pressure and two components of material velocity. The pressure operator P transforms $p_t^{x,y}$ to $p_{t+1}^{x,y}$ but it is an identity matrix for the two components of velocity (which will actually be transformed in a later process). We distribute the wavefield variables throughout space with this tiling:

$$\begin{array}{ccccccccc}
p & u & p & u & p & u & p & u & \longrightarrow x \\
v & & v & & v & & v & & \\
p & u & p & u & p & u & p & u & \\
v & & v & & v & & v & & \\
p & u & p & u & p & u & p & u & \\
v & & v & & v & & v & & \\
 & & & & & & & & \\
\downarrow & & & & & & & & \\
y & & & & & & & &
\end{array} \tag{12}$$

We define the basic tile for this discrete mesh as

$$\begin{array}{|c|c|} \hline p & u \\ \hline v & \\ \hline \end{array} \tag{13}$$

Computationally we refer to this basic tile on the inputs as $\vec{q}(x,y)$ and on the outputs as $\vec{r}(x,y)$ although the three components in the tile are at slightly different physical locations from one another. Expressing the pressure equation for these tiles we have

$$p_{t+1}^{x} \quad = \quad s_t^x + p_t^x - \frac{K\Delta t}{\Delta x}\left[(u_t^{x,y} - u_t^{x-1,y}) + (v_t^{x,y} - v_t^{x,y-1})\right] \tag{14}$$

In subroutine `pressure()` below the input pressure and velocity field is `q(x,y,*)` and the output is `r(x,y,*)`. Equation (14) is expressed in the subroutine on the line beginning with `r(x,y,p)=`.

```
# operator of pressure change from divergence of velocity
#
```

```
subroutine pressure( adj, add, kappa,       q,nx,ny,    r )
integer              adj, add,               nx,ny,       x, y,    p,   u,   v
real                           kappa(nx,ny),q(nx,ny,3), r(nx,ny,3)
call        adjnull( adj, add,              q,nx*ny*3,  r,nx*ny*3)
                                                                p=1; u=2; v=3
do x= 2, nx-1 {
do y= 2, ny-1 {
    if( adj == 0) {
        r(x,y,p)= r(x,y,p) + q(x,y,p) - (q(x,y,u) - q(x-1,y  ,u)) * kappa(x,y) -
                                        (q(x,y,v) - q(x  ,y-1,v)) * kappa(x,y)
        r(x,y,u)= r(x,y,u) + q(x,y,u)
        r(x,y,v)= r(x,y,v) + q(x,y,v)
    } else {
        q(x  ,y  ,p)= q(x  ,y  ,p) + r(x,y,p)
        q(x  ,y  ,u)= q(x  ,y  ,u) - r(x,y,p) * kappa(x,y)  + r(x,y,u)
        q(x-1,y  ,u)= q(x-1,y  ,u) + r(x,y,p) * kappa(x,y)
        q(x  ,y  ,v)= q(x  ,y  ,v) - r(x,y,p) * kappa(x,y)  + r(x,y,v)
        q(x  ,y-1,v)= q(x  ,y-1,v) + r(x,y,p) * kappa(x,y)
        }
    }}
return; end
```

Notice that the identity matrix "pass through" of the velocity components shows up as two extra assignments in the forward operator and thus as two added terms in the adjoint operator.

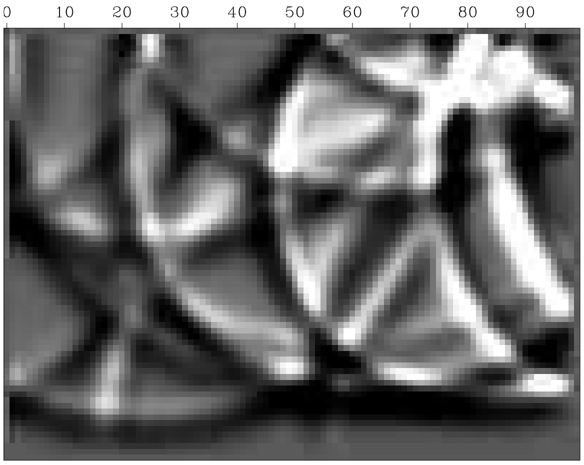

Figure 1: A few frogs hopped into the pond. Chirup! jon1-frog [ER]

The velocity operator

For the velocity operator V we use Newton's equations stating that the time rate of momentum change is the external momentum source minus the pressure gradient.

$$\rho \frac{\partial u}{\partial t} = s_u(x,y,t) - \frac{\partial p}{\partial x} \tag{15}$$

$$\rho \frac{\partial v}{\partial t} = s_v(x,y,t) - \frac{\partial p}{\partial y} \tag{16}$$

The spatial tiling we have chosen gives us two independent locations for storing density ρ, one atop the horizontal velocity u, the other atop the vertical velocity v. It may seem non physical to have one density for vertical accelerations and another for horizontal ones, but it is worth maintaining the two densities as separate entities. Waves arise from many physical paradigms besides the acoustical one and some of these may require that we maintain this possibility to introduce anisotropy. Another way to introduce anisotropy in the present codes is to have two compressibilities K in the pressure subroutine. These extra parameters might also be helpful if we were trying to improve the numerical representation of the differential operators. Subroutine `velocity()` uses `rhou(,)` for $\Delta t/(\Delta x \rho)$ at u and `rhov(,)` for $\Delta t/(\Delta x \rho)$ at v. Equation (15) in discrete form

$$u_{t+1}^{x} \quad = \quad s_t^x + u_t^x - \rho_u^{-1}(\Delta t/\Delta x)(p_t^{x+1,y} - p_t^{x,y}) \tag{17}$$

appears in subroutine `velocity()` at the line beginning with `r(x,y,u)=`.

```
# Operator of momentum change from gradient of pressure
#
subroutine velocity( adj, add, rhou,rhov,  q,nx,ny,    r )
integer             adj, add,              nx,ny,       x, y,  p,   u,   v
real rhou(nx,ny), rhov(nx,ny), q(nx,ny,3), r(nx,ny,3)
call        adjnull( adj, add,             q,nx*ny*3,  r,nx*ny*3)
                                                                p=1; u=2; v=3
do x= 2, nx-1 {
do y= 2, ny-1 {
    if( adj == 0) {
        r(x,y,u)= r(x,y,u) + q(x,y,u) - ( q(x+1,y  ,p) - q(x,y,p)) * rhou(x,y)
        r(x,y,v)= r(x,y,v) + q(x,y,v) - ( q(x  ,y+1,p) - q(x,y,p)) * rhov(x,y)
        r(x,y,p)= r(x,y,p) + q(x,y,p)
    } else {
        q(x  ,y  ,u)= q(x  ,y  ,u) + r(x,y,u)
        q(x  ,y  ,p)= q(x  ,y  ,p) + r(x,y,u) * rhou(x,y) + r(x,y,p)
        q(x+1,y  ,p)= q(x+1,y  ,p) - r(x,y,u) * rhou(x,y)
        #
        q(x  ,y  ,v)= q(x  ,y  ,v) + r(x,y,v)
        q(x  ,y  ,p)= q(x  ,y  ,p) + r(x,y,v) * rhov(x,y)
        q(x  ,y+1,p)= q(x  ,y+1,p) - r(x,y,v) * rhov(x,y)
        }
    }}
return; end
```

This time we pass through the pressure unchanged so we also see an added pressure term in the adjoint.

Viscosity

Viscosity is included mainly for numerical analysis so I chose σ to be a constant. To create viscosity, the heat flow equation can be applied to the pressure. Unfortunately we cannot use the heat flow subroutine itself because the viscosity subroutine also needs to pass through the velocity. Starting from the heatflow subroutine, including the pass through, and increasing from one to two dimensions gives subroutine `viscosity()`

```
# Viscosity operator = heatflow on pressure and pass through velocity
#
subroutine viscosity( adj, add, sigma, qq,nx,ny,  rr   )
integer               adj, add,          nx,ny,        x, y,     p,   u,   v
real                             sigma, qq(nx,ny,3), rr(nx,ny,3)
                                                               p=1; u=2; v=3
call        adjnull(  adj, add,       qq,nx*ny*3,  rr,nx*ny*3)
do x= 2, nx-1 {
do y= 2, ny-1 {
        if( adj == 0) {
                rr(x,y,p) = rr(x,y,p) + qq(x,y,p) +
                        ( qq(x-1,y  ,p) - 2*qq(x,y,p) + qq(x+1,y  ,p)) * sigma +
                        ( qq(x  ,y-1,p) - 2*qq(x,y,p) + qq(x  ,y+1,p)) * sigma
                rr(x,y,u) = rr(x,y,u) + qq(x,y,u)
                rr(x,y,v) = rr(x,y,v) + qq(x,y,v)
        } else {
                qq(x,y,u) = qq(x,y,u) + rr(x,y,u)
                qq(x,y,v) = qq(x,y,v) + rr(x,y,v)
                qq(x,y,p) = qq(x,y,p) + rr(x,y,p) * ( 1 - sigma * 4)
                qq(x-1,y,p) = qq(x-1,y,p) + rr(x,y,p) * sigma
                qq(x+1,y,p) = qq(x+1,y,p) + rr(x,y,p) * sigma
                qq(x,y-1,p) = qq(x,y-1,p) + rr(x,y,p) * sigma
                qq(x,y+1,p) = qq(x,y+1,p) + rr(x,y,p) * sigma
                }
        }}
return; end
```

Dot-product test of viscous wave equation

As with the heat equation we need sources for wavefields. I prepared a subroutine for stepping forward the HVP operator while adding in sources and used it for the dot-product test. I find the test $y'(Ax) = (A'y)'x$ using random numbers for x and y leads to the expected six digits of numerical accuracy in a 100^3 test.

I am not showing the wave stepping subroutine here because it is just like `heatsteps()` but with minor added complications.

POSSIBLE DEMONSTRATIONS

- anisotropy modeling
- random media modeling
- velocity gradient guided waves modeling

- reverse-time migration
- inversion

ACKNOWLEDGMENTS

I would like to thank Sergey Fomel and Arnaud Berlioux for proofreading and I would like to thank Sergey Fomel for showing me how to remove some clutter from the second version of the `heatsteps2()` subroutine.