Stanford Exploration Project, Report 77, November 17, 1997, pages 315–327

Stanford Exploration Project, Report 77, November 17, 1997, pages 315–327

Short Note

Choice of integration method for anti-aliased Kirchhoff migration

Dimitri Bevc and Jon Claerbout[1]

keywords: *modeling, algorithm*

INTRODUCTION

In the last SEP report (Bevc and Claerbout, 1992), we described a fast anti-aliased Kirchhoff migration and modeling algorithm. The algorithm uses causal and anti-causal integration to form triangular weighting functions which depend on operator offset, velocity, and data sampling.

In this paper we examine how the particular implementation of causal and anticausal integration affects the migration and modeling results. The best result is obtained by defining the discrete integration operation in such a way that it can be represented as multiplication by a triangular matrix with 1 on the diagonal rather than 1/2.

KIRCHHOFF ALGORITHM WITH ANTI-ALIASING

A detailed description of how the migration and modeling algorithm works is given in the previous report (Bevc and Claerbout, 1992). The subroutine `kfastaax()` on page 321 is called with `conj==0` for modeling and with `conj==1` for migration. The version of `kfastaax()` presented here differs from the previous version in that the input is not transposed. This does not result in any substantial speed penalty but it allows velocity to vary with midpoint. In order to compensate for the Hankel tail, a causal half-order derivative is applied to the data before calling the subroutine in the modeling mode, and an anticausal half-order derivative is applied after calling the subroutine in the migration mode.

[1]**email:** not available

The choice of integration methods

Double time integration of the data traces is performed before migration and after modeling in subroutine `kfastaax()` by calling subroutine `doubint()` on page 321. For `conj==0` causal integration is followed by anticausal integration and for `conj==1` anticausal integration is followed by causal integration. This strategy for creating triangular weighting functions along the operator trajectory by double integration is applied to 3-D migration by Lumley (1993) and to 3-D DMO by Blondel (1993). Theoretical aspects of anti-aliasing and integration along a line are discussed by Nichols (1993).

The main issue of this report is to analyze the difference between using `causintold()` and `causintnew()` on page 322. In the last report we used `causintnew()`, but earlier algorithms which perform similar operations with `trimo()` use `causintold()` (Claerbout, 1992).

Causal integration is defined as

$$y(t) = \int_{-\infty}^{t} x(\tau) d\tau \ .$$

The discrete version of this integral can be represented by the following matrix equation:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \\ y_8 \\ y_9 \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & \frac{1}{2} & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & \frac{1}{2} & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \frac{1}{2} & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \frac{1}{2} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \\ x_8 \\ x_9 \end{bmatrix} \tag{1}$$

This is implemented by `causintnew()` on page 322. With 1/2 on the diagonal the summation $\int_{-\infty}^{t} + \int_{t}^{\infty} = \int_{-\infty}^{\infty}$ is exactly true, so it seems that `causintnew()` is an improvement over `causintold()` which has 1 on the diagonal.

In terms of Z-transforms, `causintnew()` is represented by the bilinear transform

$$Y(Z) \ = \ \frac{1}{2} \frac{1 \ + \ Z}{1 \ - \ Z} \, X(Z) \, , \tag{2}$$

`causintold()` is represented by

$$Y(Z) \ = \ \frac{1}{1 \ - \ Z} \, X(Z) \, . \tag{3}$$

Both Equations have poles at $Z = 1$, corresponding to $\omega = 0$. Equation (2) has a zero at $Z = -1$ which is the Nyquist frequency $\omega = \pi$. The result of putting the zero in the numerator of equation (2) is that `causintnew()` acts to low-pass filter the input signal more than `causintold()`.

The net effect of modeling with `causintnew()`, including half-order differentiation can be represented in terms of Z-transform as multiplication by

$$\frac{-\,(1\,+\,Z)^2\,\sqrt{(1\,-\,Z)}}{4\,(1\,-\,Z)^2}\,. \tag{4}$$

Similarly, modeling with `causintold()`, can be represented as multiplication by

$$\frac{-Z\,\sqrt{(1\,-\,Z)}}{(1\,-\,Z)^2}\,. \tag{5}$$

Figure 1 illustrates the actions of `causintold()` and `causintnew()`. Triangles are created by integrating three spikes causally and then anticausally. The triangle created by double application of `causintold()` is plotted on the second line of Figure 1. It is sharper and narrower than the triangle created by double application of `causintnew` and displayed on the third line of Figure 1. The last two lines in Figure 1 are the result of the application of equations (5) and (4). The application of `causintnew` to implement equation (4) results in a smoother and broader phase shifted triangle.

The effect of using `causintnew` is to low-pass filter the result which is obtained by using `causintold()`. This effect is even more evident when the routines are used in conjunction with `kfastaax()` on synthetic data.

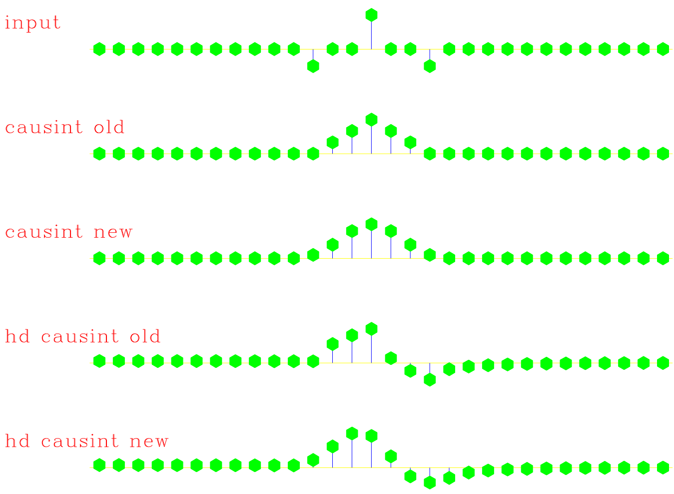

Figure 1: The first line is the input to the double causal integration subroutine. Second and third lines are triangles created by applying double integration using `causintold()` and `causintnew()`. The fourth and fifth lines are the result of applying the half-order differential before double integration. The triangles created with `causintold()` are narrower and sharper than the ones created with `causintnew()`. dimitri2-caus [ER]

Synthetic example of modeling and migration

A simple input model and the result of migrating the synthetic data generated from it without anti-aliasing is displayed in Figure 2. The artifacts due to operator aliasing are the smiles on the point diffractors and the prominent precursor above the flat event.

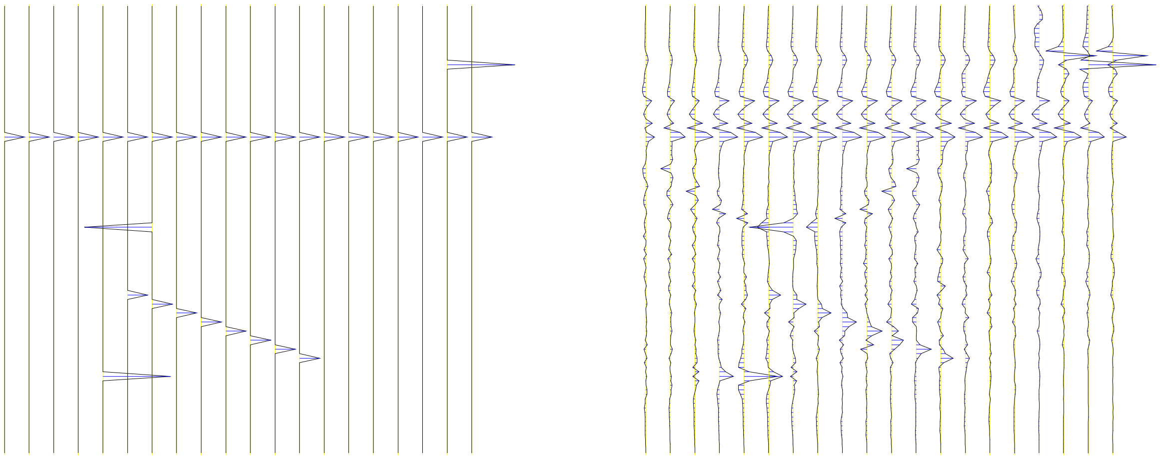

Figure 2: The plot on the left is the model input. The synthetic image on the right is made by first modeling the data with simulated field arrays and then migrating the resulting synthetic without anti-aliasing (`anti==0`). dimitri2-migalias [ER]

Figure 3 is a comparison of modeling and migration using `causintnew()` and `causintold()`. The operator aliasing artifacts are eliminated in both cases, but the result obtained with `causintnew()` (upper two panels) has much less resolution than the result obtained with `causintold()` (lower two panels). The image obtained with `causintold()` (lower right side of Figure 3) is a better representation of the model (Figure 2) than the image obtained with `causintnew()`.

CONCLUSION

Performing causal and anticausal integration with `causintold()` on page 321 results in better modeling and imaging. A simple and effective recursive algorithm results from defining the integration operation to correspond to multiplication by a triangular matrix which has 1 on the diagonal. This formulation for causal and anticausal integration produces good results when used to build triangular weights for anti-aliased Kirchhoff migration and modeling.

REFERENCES

Bevc, D., and Claerbout, J., 1992, Fast anti-aliased Kirchhoff migration and modeling: SEP–**75**, 91–96.

Blondel, P., 1993, Constant-velocity anti-aliasing three-dimensional integral dip moveout: SEP–**77**, 49–58.

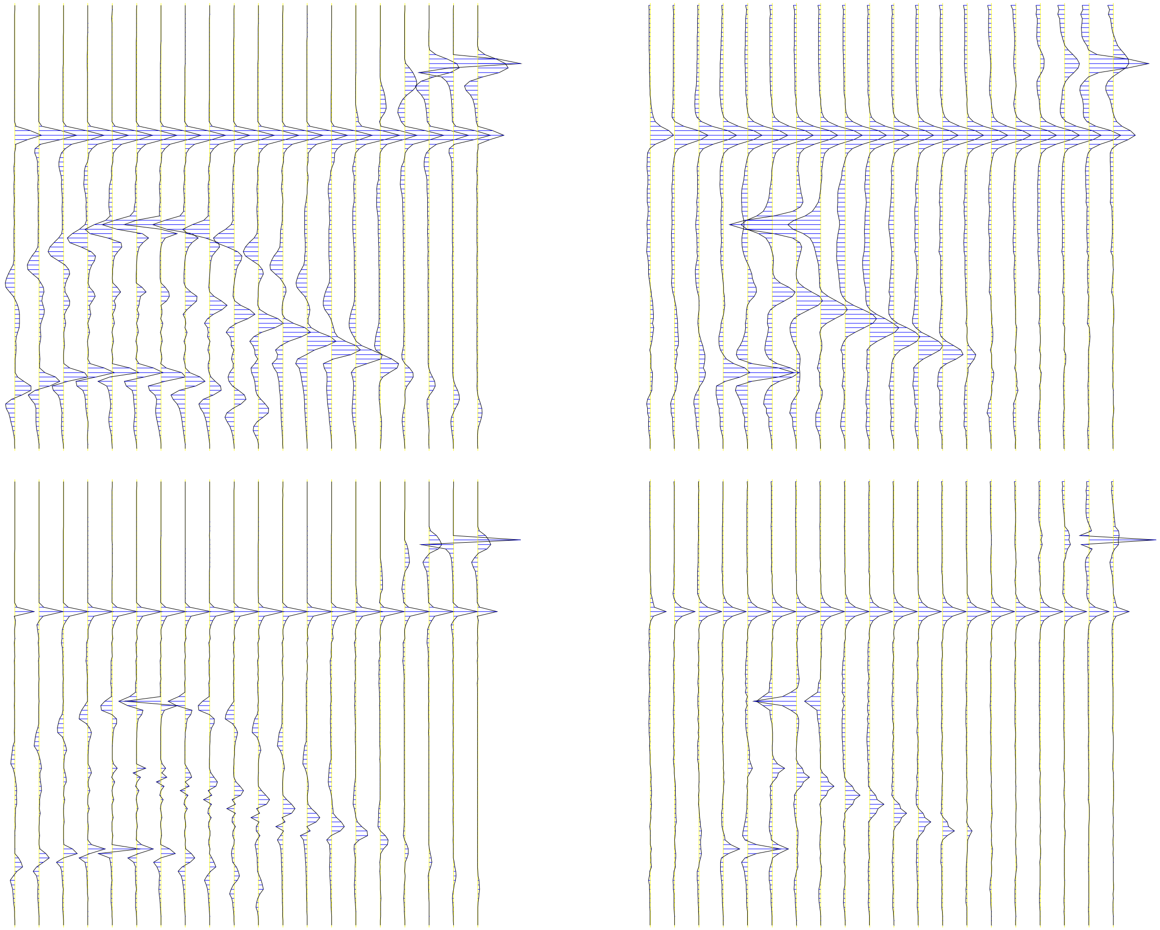

Figure 3: Comparison of modeling and migration results using the two different versions of the integration program. Modeling `conj==0` is on the left and migration `conj==1` is on the right. The top panels were made using the integration program with 1/2 on the diagonal, and the bottom panels were made with 1 on the diagonal. $\boxed{\text{dimitri2-migaa}}$ [ER]

Claerbout, J. F., 1992, Anti aliasing: SEP–**73**, 371–390.

Lumley, D. E., 1993, Anti-aliased Kirchhoff 3-D migration: A salt intrusion example: SEP–**77**, 1–18.

Nichols, D., 1993, Integration along a line in a sampled space: SEP–**77**, 283–294.

APPENDIX A

The code used to generate the figures in this report is presented here.

```
# Kirchhoff migration and diffraction.  (fast anti-aliased)
#
subroutine kfastaax(conj,add,anti,velocity,t0,dt,dx,nt,nx, modl,       data)
integer             conj,add,                      nt,nx
real                         anti,velocity,t0,dt,dx,     modl(nt,nx),data(nt,nx)
real        z, t, h,       tp, tm,   amp, anti, slope
integer ix,iz,it,ih,      itp,itm,  xstart, xend
temporary real ss(nt,nx)

call conjnull(      conj, add,                         modl,nt*nx, data,nt*nx)
call null(     ss,nt*nx)

if(conj!=0)
        do ix=1,nx
                call doubint( 1, 0, nt, ss(1,ix), data(1,ix))
do ih= -nx, nx {        h = dx * ih                     # h = offset
        do iz= 2, nt {  z = t0 + dt * (iz-1)            # z = travel-time depth
                        t = sqrt( z**2  +  (h/velocity)**2 )
        slope = anti * ( (1./velocity)**2 ) * h / (t+dt)
        tp = t + abs(slope * dx) + dt;   itp = 1.5 + (tp-t0) / dt
        tm = t - abs(slope * dx) - dt;   itm = 1.5 + (tm-t0) / dt
        if( itm <= 1  )  next
        if( itp >= nt )  break
        amp = 1.0 * sqrt( nt*dt/(t+dt)) * (z+dt)/(t+dt) * (dt/(dt+tp-tm)) ** 2
        xstart = 1-ih;          xstart = max0(  1, xstart)
        xend   = nx-ih;         xend   = min0( nx, xend)
        call spot3x(conj,1, -amp,2*amp,-amp, t0,dt, tm,t,tp,xstart,xend,ih,iz,nt,nx,modl,ss)
                }
        }
if(conj==0)
        do ix=1,nx
                call doubint( 0, 0, nt, ss(1,ix), data(1,ix))
return; end


# Double integration, first causal, then anticausal.
#
subroutine doubint( conj, add, n,             pp   , qq  )
integer             conj, add, n;       real  pp(n), qq(n)
temporary real tt(n)
call conjnull(      conj, add,                pp,n,  qq,n)
if( conj == 0 ) {       call causint( 0,   0, n,pp,  tt  )
                        call causint( 1, add, n,qq,  tt  )
                }
else {                  call causint( 1,   0, n,tt,  qq  )
                        call causint( 0, add, n,tt,  pp  )
                }
return; end
```

```
# causal integral the old way with 1's on diagonal (SEP73 p375)
#
subroutine causintold( conj, add,        n,pp,  qq   )
integer i,          n, conj, add;   real pp(n), qq(n )
temporary real tt( n)
call conjnull(      conj, add,        pp,n,  qq,n )
if( conj == 0){                         tt(1) = pp(1)
                                do i= 2, n
                                        tt(i) = tt(i-1) + pp(i)
                                do i= 1, n
                                        qq(i) = qq(i) + tt(i)
                }

else {                                  tt(n) = qq(n)
                                do i= n, 2, -1
                                        tt(i-1) = tt(i) + qq(i-1)
                                do i= 1, n
                                        pp(i) = pp(i) + tt(i)
                }
return; end


# causal integral with 1/2's on the diagonal
#
subroutine causintnew( conj, add,        n,pp,  qq   )
integer i,          n, conj, add;   real pp(n), qq(n )
temporary real tt( n)
call conjnull(      conj, add,        pp,n,  qq,n )
if( conj == 0){                         tt(1) = pp(1) / 2.
                                do i= 2, n
                                        tt(i) = tt(i-1) + (pp(i) + pp(i-1)) / 2.
                do i= 1, n
                  qq(i) = qq(i) + tt(i)
                }

else {                                  tt(n) = qq(n) / 2.
                                do i= n, 2, -1
                                        tt(i-1) = tt(i) + (qq(i) + qq(i-1)) / 2.
                do i= 1, n
                  pp(i) = pp(i) + tt(i)
                }
return; end


#  Scaled linear interpolation.
#  Modified for fast triangular weighted anti-aliased migration. DB 8-14-92
#  This routine loops over ix.
#
subroutine spot3x(conj,add, scm,sc,scp, t0,dt, tm,t,tp, xstart,xend,ih,iz,nt,nx,val,ss)
integer           conj,add,                             xstart,xend,ih,iz,nt,nx
real                        scm,sc,scp, t0,dt, tm,t,tp
real val(nt,nx),ss(nt,nx)
integer itm,it, itp, itcm,itc,itcp, ix
real    tcm,tcp,tc,  frm,fr,frp
```

```
temporary real vecm(nx),   vec(nx),  vecp(nx)
temporary real vecm1(nx), vec1(nx), vecp1(nx)

call conjnull(    conj, add,                     val,nt*nx, ss,nt*nx)
if (conj == 0){
        call null(vecm ,nx)
        call null(vec  ,nx)
        call null(vecp ,nx)
        call null(vecm1,nx)
        call null(vec1 ,nx)
        call null(vecp1,nx)
        }
tcm = (tm-t0) / dt;     itcm = tcm;      itm = 1 + itcm;        frm = tcm - itcm
tc  = (t -t0) / dt;     itc  = tc;       it  = 1 + itc;         fr  = tc  - itc
tcp = (tp-t0) / dt;     itcp = tcp;      itp = 1 + itcp;        frp = tcp - itcp

if(1 <= it & it < nt) {
if( conj != 0) {
        do ix=1,nx {
                 vecm(ix) = ss(itm  ,ix)
                  vec(ix) = ss(it   ,ix)
                 vecp(ix) = ss(itp  ,ix)
                vecm1(ix) = ss(itm+1,ix)
                 vec1(ix) = ss(it+1 ,ix)
                vecp1(ix) = ss(itp+1,ix)
                }
        }
do ix=xstart,xend {
if( conj == 0) {
         vecm(ix+ih) = vecm(ix+ih)  + (1.-frm) * val(iz,ix) * scm
          vec(ix+ih) = vec(ix+ih)   + (1.-fr ) * val(iz,ix) * sc
         vecp(ix+ih) = vecp(ix+ih)  + (1.-frp) * val(iz,ix) * scp
        vecm1(ix+ih) = vecm1(ix+ih) +  frm     * val(iz,ix) * scm
         vec1(ix+ih) = vec1(ix+ih)  +  fr      * val(iz,ix) * sc
        vecp1(ix+ih) = vecp1(ix+ih) +  frp     * val(iz,ix) * scp
        }
else    {
        val(iz,ix)=val(iz,ix)+((1.-fr )*vec(ix+ih )+fr *vec1(ix+ih ))*sc
        val(iz,ix)=val(iz,ix)+((1.-frm)*vecm(ix+ih)+frm*vecm1(ix+ih))*scm
        val(iz,ix)=val(iz,ix)+((1.-frp)*vecp(ix+ih)+frp*vecp1(ix+ih))*scp
        }
}
if( conj == 0) {
        do ix=1,nx {
                ss(itm  ,ix) = vecm(ix)  + ss(itm  ,ix)
                ss(it   ,ix) = vec(ix)   + ss(it   ,ix)
                ss(itp  ,ix) = vecp(ix)  + ss(itp  ,ix)
                ss(itm+1,ix) = vecm1(ix) + ss(itm+1,ix)
                ss(it+1 ,ix) = vec1(ix)  + ss(it+1 ,ix)
                ss(itp+1,ix) = vecp1(ix) + ss(itp+1,ix)
                }
        }
}
else
```

```
        call erexit('spot3: at boundary')
return; end
```