Stanford Exploration Project, Report 79, November 16, 1997, pages 35–45

3-D data infill and extrapolation

Jon F. Claerbout[1]

keywords: *deconvolution, three-dimensional, missing data, algorithm*

ABSTRACT

Here I deal with data given on a uniform 3-D mesh (or 1-D or 2-D). I provide code for recovering missing data in the interior of the cube or abutting one wall of the 3-D data cube by a two-stage linear least squares process. First I fit a 3-D prediction-error filter (PEF) to the extended cube ignoring regression equations that involve missing elements. Then I estimate those missing elements by finding the data that minimizes the power out of the PEF. As in my other work since SEP-73, the volume can be broken into subcubes in which separate problems are solved and pieced together.

INTRODUCTION

Geophysicists have known for many years how to compute the prediction-error filter (PEF) for a time series. Given a prediction filter, it is fairly obvious how to extrapolate the signal beyond its ends or how to fill in gaps. (Classical theory focuses on instabilities that can arise, but these instabilities do not arise when using the methods found in my recent book (Claerbout, 1992), chapter 8. In essence, instability arises from using a prediction filter recursively. Instability is easily avoided by using a second stage of linear least squares to solve directly for the missing data.)

My book also shows examples of how to find missing portions of signals, both in their interior and exterior. It also shows some applications of multidimensional prediction-error filters. Additionally, the book illustrates the idea of fitting in small windows and choosing edge conditions so as to be able to merge windows fairly seamlessly. In SEP-73 I formalized and extended the ideas behind windowing as a means of coping with nonstationarity and I integrated it with prediction-error filtering in 2-D, and have since extended it to 3-D and prepared more examples (Claerbout, 1993b), (1993), (1992), (1992), and (1992).

Here I develop code to restore and extend a 3-D data cube by a two-stage linear least squares process. In the first stage I fit a 3-D prediction-error filter (PEF) to the

[1]**email:** not available

given cube. Regression equations that involve unknown missing data elements are weighted to zero before solving the least-squares problem by conjugate gradients. In a second stage, I take the PEF as known and I find missing elements in (or beyond) the cube to minimize the power out of the prediction-error filter.

POTENTIAL APPLICATIONS

The methodology and code here is readily extendable to four dimensions (prestack 3-D data) but first I'll seek 3-D and 2-D applications.

Shot continuation is an obvious task for a data cube extrapolation program. There are two applications of shot continuation. First is the obvious one of repairing holes in data in an unobtrusive way. Second is to cooperate with reflection tomographic studies such as that proposed by Mattias Schwab (Schwab, 1993).

Offset continuation is a well-developed topic because of its close link with dip moveout (DMO). DMO is heavily used in the industry. I do not know how the data-cube extrapolation code I am designing here would fit into DMO and stacking, but because these are such important processes, the appearance of a fundamentally new tool like this should be of interest. It is curious that the DMO operator is traditionally derived from theory, and the theory requires the unknown velocity function of depth, whereas here I propose estimating the offset continuation operator directly from the data itself, without the need of a velocity model.

Obviously, one application is to extrapolate off the sides of a **constant-offset section**. This would reduce migration semicircles at the survey's ends.

Another application is to extrapolate off the **cable ends** of a common midpoint gather or a common shot point gather. This could enhance multiple prediction or reduce artifacts in velocity analysis.

The application that drove me to putting the code in its present form is extending Kjartansson-style **tomography.**

CODE DESCRIPTION

The concepts are straightforward and do not require several pages of Greek symbols beyond what you see in my text book. It might be correct to say that these ideas have been around for a generation, but because of their complexity, they have not been embodied in computer code. I feel I have learned how to write this kind of code without getting bogged down and that it is worthwhile for me to sketch it out. I believe this paper article is complete enough for you to see how to reproduce this work, however, additional background is found in the article describing code I developed six months ago for a steep-dip deconvolution project (1993) and the complete code is found on the accompanying CD-ROM.

The first complicating issue is the nonstationarity that I cope with by "patching". I described patching in great detail in two dimensions in (Claerbout, 1992e). The only new ingredient is the third dimension, and the extension from the two dimensional code given there is so obvious that I do not show the listings of the three utility subroutines: `patch3()` for chopping things up and reassembling, `tent3()` for providing a weighting function that is triangle-like with ends at the output of prediction-error filtering, and `mkwallwt3()` which gives a weighting function necessary for reassembling the subcubes.

Outputs of linear operators are first erased with subroutine `adjnull()` on page 162, rearranged from `conjzero()` in my book. Convolution without flowing off the ends of data is done in my book with subroutine `cinloi()` and here in 3-dimensions it is done with subroutine `cinloi3()` on the current page. There is an analogous program `cinlof3()` for when the adjoint space is the filter instead of the input. Subroutine `cinlof3()` is omitted here because it differs in only one or two lines from `cinloi3()` on this page and from `cinlof()` in my book.

```
# CINLOI3 --- COnvolution INternal with Lags.  Output is adjoint to INPUT.
#
subroutine cinloi3( adj,add, lag1,lag2,lag3, aa,na1,na2,na3,  xx,n1,n2,n3, yy)
integer             adj,add, lag1,lag2,lag3,    na1,na2,na3,     n1,n2,n3
real                                         aa(na1,na2,na3), xx(n1,n2,n3)
real                                                          yy(n1,n2,n3)
integer y1,y2,y3, x1,x2,x3, a1,a2,a3
call adjnull(       adj, add,                   xx,n1*n2*n3,  yy,n1*n2*n3)

do a3=1,na3 {  do y3= 1+na3-lag3, n3-lag3+1 {  x3= y3 - a3 + lag3
do a2=1,na2 {  do y2= 1+na2-lag2, n2-lag2+1 {  x2= y2 - a2 + lag2
do a1=1,na1 {  do y1= 1+na1-lag1, n1-lag1+1 {  x1= y1 - a1 + lag1
        if( adj == 0 )
                yy( y1,y2,y3) = yy( y1,y2,y3) + aa( a1,a2,a3) * xx( x1,x2,x3)
        else
                xx( x1,x2,x3) = xx( x1,x2,x3) + aa( a1,a2,a3) * yy( y1,y2,y3)
        }}  }}  }}
return; end
```

Next I incorporate the idea that when dealing with second order statistics like autocorrelation functions there is no sense of the direction of flow of time. Prediction forward should use the same filter as prediction backward. This is an idea that will seem natural to people who have worked in time-series analysis. I have not established convincingly that this idea is genuinely required in two and three dimensions, but I believe it to be, and I incorporate it in subroutine `burg23()` on the current page.

```
# Convolve 3-D filter onto forward and backward data.   Dual is filter.
#       output residual partitioned into normal and backward parts.
#       output residual(,) aligns with data(,) at filter coef aa(lg1,lg2,lg3)
#
subroutine burg23( adj,add, lg1,lg2,lg3,    data,n1,n2,n3,  aa,a1,a2,a3, resid)
integer i1,i2,i3,  adj,add, lg1,lg2,lg3,         n1,n2,n3,     a1,a2,a3
real                                       data(n1,n2,n3), aa(a1,a2,a3)
```

```
real                                    resid( n1,n2,n3,  2)
temporary real                           back( n1,n2,n3)
call adjnull(      adj,add,  aa,a1*a2*a3, resid, n1*n2*n3 * 2)
do i3= 1, n3 {
do i2= 1, n2 {
do i1= 1, n1 {  back( n1-i1+1, n2-i2+1, n3-i3+1) = data( i1, i2, i3)
                }}}
call cinlof3( adj, 1, lg1,lg2,lg3, data,n1,n2,n3, aa,a1,a2,a3, resid         )
call cinlof3( adj, 1, lg1,lg2,lg3, back,n1,n2,n3, aa,a1,a2,a3, resid(1,1,1,2))
return; end
```

In three dimensions, the idea is that the same filter should be applicable when all the coordinate axes are simultaneously reversed. Obviously, specialized applications will allow additional restrictions because of symmetry.

The subroutine `setfree()` on this page sets up the general form of a multidimensional prediction error filter.

```
# Flag the free coefficients in a 3-D prediction-error filter
#
subroutine setfree( lag1,lag2,lag3, afree, a1,a2,a3, niter)
integer   i1,i2,i3, lag1,lag2,lag3,        a1,a2,a3, niter
real                              afree( a1,a2,a3)
#       Constrain to zero the half-volume before lag3   (usually external)
#       Constrain to zero the half-plane  before lag2
#       Constrain to zero the half-line   before lag1
#       Constrain         the predicted point.
do i1= 1, a1
do i2= 1, a2
do i3= 1, a3
        afree(i1,i2,i3) = 1.
do i3=1,min(a3,lag3-1){ do i2=1,a2{ do i1=1,a1{ afree(   i1,  i2,  i3)=0.}}}
do i2=1,min(a2,lag2-1){ do i1=1,a1{            afree(   i1,  i2,lag3)=0.}}
do i1=1,min(a1,lag1-1){                        afree(   i1,lag2,lag3)=0.}
                                               afree (lag1,lag2,lag3)=0.
# force lateral prediction
#do i1=1,a1 { afree(i1,lag2,lag3)=0.0 }

niter = 0
do i1= 1, a1
do i2= 1, a2
do i3= 1, a3
        if( afree(i1,i2,i3) > 0.)
                niter = niter + 1              # count them
return; end
```

For more details about this concept, see PVI leading up to page 198. Roughly, the idea is that a prediction-error filter allows nonzero values only on one side of the prediction point, and in three dimensions, "one side" means a halfspace.

The first stage of the least-squares estimation is computing the prediction-error filter by subroutine `pef3()` on the current page using the conjugate-gradient strategy in my book (Claerbout, 1992).

```
# Find prediction-error filter avoiding missing data.
#
subroutine pef3( dfre,data,n1,n2,n3, niter, lag1,lag2,lag3,afre, aa,a1,a2,a3 )
integer                 n1,n2,n3, niter, lag1,lag2,lag3,         a1,a2,a3
real   dfre(n1,n2,n3),data(n1,n2,n3),           afre(a1,a2,a3), aa(a1,a2,a3)
integer     i1,i2,i3, a123, r123,     iter
temporary real da( a1,a2,a3), dr( n1*n2*n3*2), rr( n1*n2*n3*2), ww( n1*n2*n3*2)
temporary real sa( a1,a2,a3), sr( n1*n2*n3*2), wr( n1*n2*n3*2)
            a123 = a1*a2*a3;                r123 = n1*n2*n3*2
call null( rr, r123);   call null( aa, a123);          aa(lag1,lag2,lag3) = 1.
call null( dr, r123);   call null( da, a123);        afre(lag1,lag2,lag3) = 1.
call         burg23 ( 0, 0, lag1,lag2,lag3, dfre,n1,n2,n3, afre,a1,a2,a3, ww )
                                                     afre(lag1,lag2,lag3) = 0.
do i1= 1, r123
        if( ww(i1) > 0.0) { ww(i1) = 0.0 }    # free data hits free filter.
        else              { ww(i1) = 1.0 }
call         burg23 ( 0, 0, lag1,lag2,lag3, data,n1,n2,n3,  aa,a1,a2,a3, rr )
call         diag(    0, 0,                 ww, r123,              rr, wr )
call         scale (    -1.,    r123,                                  wr )
do iter= 0, niter {
        call diag(    0, 0,                 ww, r123,              wr, dr )
        call burg23 ( 1, 0, lag1,lag2,lag3, data,n1,n2,n3,  da,a1,a2,a3, dr )
        do i3= 1, a3
        do i2= 1, a2
        do i1= 1, a1 {  da(i1,i2,i3) = da(i1,i2,i3) * afre(i1,i2,i3)}
        call burg23 ( 0, 0, lag1,lag2,lag3, data,n1,n2,n3,  da,a1,a2,a3, dr )
        call diag(    0, 0,                 ww, r123,              dr, dr )
        call cgstep( iter,  a123 , aa,da,sa,
                            r123 , wr,dr,sr )
        }
call         burg23 ( 0, 0, lag1,lag2,lag3, data,n1,n2,n3,  aa,a1,a2,a3, rr )
do i3=1,a3  { call snap('aa.H', a1, a2, aa(1,1,i3)) }          # snapshot view
return; end
```

Subroutine `pef3()` resembles the two dimensional `pef2()` in PVI on page 193, but there are some new features. First, it is three-dimensional like `moplan3()` in SEP-77. Second, it will ignore regression equations that involve missing data values.

To allow the data volume to contain an arbitrary mixture of known and unknown data values, we need a mask volume `dfre` the size of the data volume with mask values +1.0 where data is missing and 0.0 where data is known. Likewise we make a filter mask `afre` where nonzero filter coefficients are +1.0 and zero valued filter coefficients are 0.0. Then, near the beginning of subroutine `pef3()`, these masks are convolved by `burg23()` as we later plan to convolve the data. Where the mask output is non zero is where residuals in the filter estimation problem should be weighted by zero otherwise the weighting function `ww()` is set to unity. I have not had time to worry about what happens when too much data is missing so that all weights are zero. I need to think about stabilization or supplementing the data volume by a synthetic "training dataset" or by a "prior filter."

Subroutine `pef3()` iteratively applies the filter routine `burg23()` on page 37.

Notice wherever subroutine `burg23()` appears, we also see a call to subroutine `diag()` which is a diagonal matrix multiply to apply a weighting function. An example of a tutorial code with a weighting function is the one-dimensional code for subroutine `iner()` in PVI. (The code for `diag()` found in PVI was altered to allow the inputs to overlay the outputs.)

Subroutine `miss3()` on the current page is a three-dimensional example of the one-dimensional subroutine `miss1()` found in PVI. The three-dimensional subroutine is much like the three-dimensional filter estimation subroutine `pef3()` on page 38.

An open question is how many conjugate gradient iterations are needed in missing data programs. When estimating filters, I set the iteration count `niter` at the number of free parameters which for a $3 \times 3 \times 3$ filter is about 25. Theoretically, this gives me the exact solution. The number of free parameters in the missing data estimation, however, could be very large implying impractical compute times for the exact solution. I default the number of iterations for finding missing data to 25. Where gaps are small, they fill much more quickly. Where they are large, this is not enough. I chose the 25 to roughly balance the amount of time I spend computing the PEF with the time using it. I suspect that gaps smaller than 5 points are accurately filled, but at greater distances the filled value is noticeably smaller than the theoretical solution.

In subroutine `miss3()` on the current page I left in place a particular prior filter `bb(*,*,*)=` $(\partial_{23} \approx 0)$. Because it is multiplied by ϵ =`eps=0` it has no effect, but I left it in place as a starting point for further experimentation as will be discussed later in connection with the results.

```
# Given PEF aa() and known data cube,  find xx = known + missing
#
subroutine miss3( xfre, xx, n1,n2,n3, niter, lag1,lag2,lag3, aa,a1,a2,a3 )
integer           x123,    n1,n2,n3, niter, lag1,lag2,lag3,    a1,a2,a3
integer                    i1,i2,i3,  iter
real     xfre(n1,n2,n3), xx(n1,n2,n3),                      aa(a1,a2,a3),eps
real                                                           bb(1 ,2 ,2 )
integer                                                           b1,b2,b3
temporary real                   rr( n1*n2*n3, 2)
temporary real     dx( n1,n2,n3), dr( n1*n2*n3, 2)
temporary real     sx( n1,n2,n3), sr( n1*n2*n3, 2)
                                                     eps= 0.0; b1=1; b2=2; b3=2
        bb(1,1,1) =   eps;   bb(1,2,1) = -eps
        bb(1,1,2) =  -eps;   bb(1,2,2) =  eps
                                          x123= n1 * n2 * n3
call null( dr, 2* x123);    call null( dx, x123)
call null( rr, 2* x123)
        call cinloi3( 0,0,lag1,lag2,lag3,aa,a1,a2,a3, xx,n1,n2,n3, rr(1,1))
        call cinloi3( 0,0,   1,   1,   1,bb,b1,b2,b3, xx,n1,n2,n3, rr(1,2))
        call scale( -1.,                   2 * x123,              rr         )
do iter= 0, niter {
        call cinloi3( 1,0,lag1,lag2,lag3,aa,a1,a2,a3, dx,n1,n2,n3, rr(1,1))
        call cinloi3( 1,1,   1,   1,   1,bb,b1,b2,b3, dx,n1,n2,n3, rr(1,2))
        do i1= 1, n1 {
        do i2= 1, n2 {
```

```
        do i3= 1, n3 {  dx(i1,i2,i3) = dx(i1,i2,i3) * xfre(i1,i2,i3)
                        }}}
        call cinloi3( 0,0,lag1,lag2,lag3,aa,a1,a2,a3, dx,n1,n2,n3, dr(1,1))
        call cinloi3( 0,0,   1,   1,   1,bb,b1,b2,b3, dx,n1,n2,n3, dr(1,2))
        call cgstep( iter,   x123, xx,dx,sx,
                           2*x123, rr,dr,sr )
        }
return; end
```

Subroutine room() on this page works on each subvolume. First it finds the PEF and then it uses the PEF to determine the missing data.

```
# Find 3-D PEF. Use it to find missing data.
#
subroutine room( dfre,data,n1,n2,n3,nitpef,nitmis,lag1,lag2,lag3,afre, a1,a2,a3)
integer                   n1,n2,n3,nitpef,nitmis,lag1,lag2,lag3,      a1,a2,a3
real    dfre(n1,n2,n3), data(n1,n2,n3),                          afre( a1,a2,a3)
temporary real                                                      aa(a1,a2,a3)

call pef3(  dfre,data, n1,n2,n3, nitpef, lag1,lag2,lag3,afre, aa,a1,a2,a3)

call miss3( dfre,data, n1,n2,n3, nitmis, lag1,lag2,lag3,      aa,a1,a2,a3)

return; end
```

The patching business (Claerbout, 1992e) handled by subroutine house() on the current page is like the other applications of two-dimensional and three-dimensional filters. In most applications, the patching code looks the same, but in subroutine house() on this page there is an additional feature, the mask of the missing data points is also patched.

```
# house --- chop up house, make nice rooms, and reassemble house.
#
subroutine house(   n1,n2,n3, w1,w2,w3, k1,k2,k3, lag1,lag2,lag3,
                                   afre,a1,a2,a3, nitpef,nitmis, dfre,data,dext)
integer             n1,n2,n3, w1,w2,w3, k1,k2,k3, nitpef,nitmis
integer     i1,i2,i3,   lag1,lag2,lag3, a1,a2,a3
          real   afre(a1,a2,a3), dfre(n1,n2,n3), data(n1,n2,n3), dext(n1,n2,n3)
temporary real windwt(w1,w2,w3), pdfr(w1,w2,w3), pdat(w1,w2,w3)
call    tent3( a1,a2,a3, lag1,lag2,lag3, windwt, w1,w2,w3)
call null(                                                       dext,n1*n2*n3)
do i3= 1, k3 {
do i2= 1, k2 {
do i1= 1, k1 {
        call patch3( 0,0, i1,i2,i3, k1,k2,k3, dfre, n1,n2,n3, pdfr, w1,w2,w3)
        call patch3( 0,0, i1,i2,i3, k1,k2,k3, data, n1,n2,n3, pdat, w1,w2,w3)
        call room(pdfr,pdat,w1,w2,n3,nitpef,nitmis,lag1,lag2,lag3,afre,a1,a2,a3)
        call diag( 0, 0, windwt, w1*w2*w3, pdat, pdat )
        call patch3( 1,1, i1,i2,i3, k1,k2,k3, dext, n1,n2,n3, pdat, w1,w2,w3)
        }}}
call mkwallwt3(k1,k2,k3, windwt, w1,w2,w3, dfre, n1,n2,n3)     # dfre = wallwt
call        diag( 0, 0,   dfre, n1*n2*n3, dext, dext )
return; end
```

The main program serves to connect the subroutines to our file system. To avoid clutter, The main program makes the mask of unknown data values by assuming that zero valued data is unknown. It adds one part in 10^{-4} noise to the data to accommodate the deficiencies of the CG algorithm in PVI when faced with all zero data. At the last moment, the main program checks where the output is zero because of the filter not running off the sides of the data. At these locations, the output is replaced by the input.

RESULTS AND BOUNDARY CONDITIONS

Figure 1 was made with a $3 \times 3 \times 3$ filter.

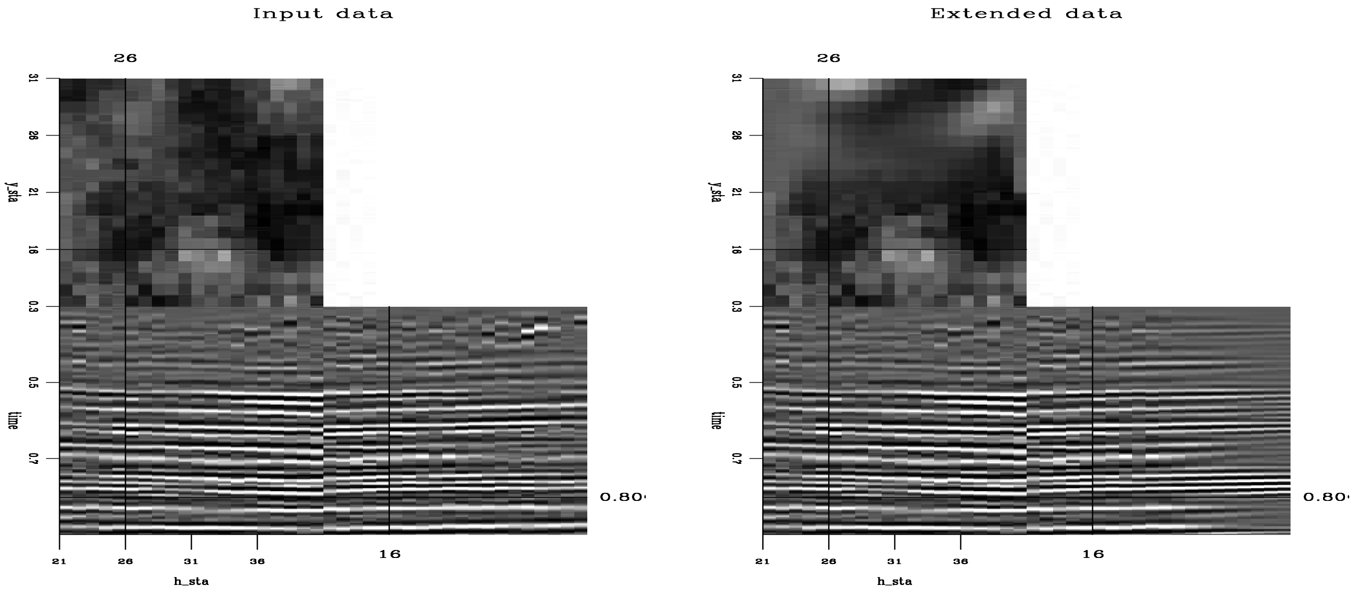

Figure 1: Original cube on left. The back half of the cube was chopped off. Right shows the reconstruction. Press button for X11movie. jon3-fullcube [ER]

Looking at movies I got a better idea of the quality of the extension than you will be able to get from looking at the paper figure. I believe it is fair to say that the extended cube is generally more pleasing than it appears on the paper copy. An exception is the side boundary planes which are poor, although this also is not clear from the paper figure.

As a last resort, I adjusted the triangle windows to match the outputs of a PE filter instead of the input data. Even this is not a satisfactory solution, as careful inspection, not apparent on the given figure, shows that the second-from-end side panels are also poorer than expected.

An early decision was made to use side boundary conditions where the filter does not flow off the data. I call these “don’t care” or “internal” end conditions, although neither name is quite right. Such ends are essential for internal boundaries to be seamless. The boundaries surrounding the entire mass, however, probably should be handled in a different manner because a growing solution at the sides is a likely result. Also, if the data contains large internal gaps, difficulty is likely to arise where patches accidentally cross the gaps. A robust solution to the missing data problem would involve designing subvolume shapes and sizes based on the shapes and sizes of the data gaps. This is beyond my present needs, however. Slow growth with distance is not a problem if you are not planning to extrapolate a long distance. I believe that the extrapolation of the 3-axis in Figure 1 is growing slowly with distance and that this can be controlled in either of two ways: (1) using a different boundary condition, namely, zero value some distance away, or with stabilization. You can experiment with the stabilization that I left switched off in subroutine `miss3()` on page 40. I define stability as the data having finite energy as its volume tends to infinity. Stabilization is certain with an impulsive prior filter, i.e., `bb( b1=1, b2=1, b3=1)`. Other priors have directions that they extrapolate data in a stable manner. With "don't care" or "internal" end conditions, I believe the *only* filter that assures stability for data volume expansion in *all* directions is the impulse function.

RECOMMENDATIONS FOR FUTURE WORK

The program developed here is a very interesting one and it deserves far more testing. I am dropping this code as soon as it works and switching to tomography where I need this code.

ACKNOWLEDGMENTS

I would like to thank Ray Abma for his continuing interest in this work and for first setting up the 3-D test case.

REFERENCES

Claerbout, J. F., 1992a, Crossline regridding by inversion: SEP–**73**, 421–428.

Claerbout, J. F., 1992b, Earth Soundings Analysis: Processing Versus Inversion: Blackwell Scientific Publications.

Claerbout, J. F., 1992c, Filling data gaps using a local plane-wave model: SEP–**73**, 401–408.

Claerbout, J. F., 1992d, Information from smiles: Mono-plane-annihilator weighted regression: SEP–**73**, 409–420.

Claerbout, J. F., 1992e, Nonstationarity and conjugacy: Utilities for data patch work: SEP–**73**, 391–400.

Claerbout, J. F., 1993a, 3-D local-monoplane annihilator: SEP–**77**, 19–26.

Claerbout, J. F., 1993b, Steep-dip deconvolution: SEP–**77**, 245–256.

Schwab, M., 1993, Shot gather continuation: SEP–**77**, 117–130.