Chapter 1

Aliasing and spatial prediction

In chapter **??**, using the principle of minimum energy we found filters given signals, we found signals given filters, and, given enough constraints, we found both at the same time. Mysteriously, one of the requirements is that the filter be a one-sided **autoregression** filter, i.e., the prediction-error filter (PEF). Since the output of this filter is white, the filter itself "learns" the spectrum of the data where the data is given, so taking this filter to areas where data is not given, we are able to estimate missing data of the appropriate spectrum which seamlessly melds the known region to the unknown. In this chapter, we extend these ideas to two dimensions. We first see that simple two-dimensional filters manipulate the spectrum of dips present in a data plane. A first goal is to uncover the form of the mysterious **whitening filter** in two dimensions. Then we write subroutines, analogous to those in chapter **??** to find the 2-D PEF and the missing data in 2-D. We test these on a variety of curious data sets including interpolation between satellite tracks and interpolation of sea-bottom topography (Seabeam). Finally, since the world is generally filled with plane waves with not many propagation angles in a given area we see how it is often possible to overcome the famous **Nyquist** sampling requirement.

1.1 MULTIDIMENSIONAL FILTERS

Convolution in two dimensions is just like convolution in one dimension except that convolution is done on two axes. The input and output data are planes of numbers and the filter is also a plane. A two-dimensional filter is a small plane of numbers that is convolved over a big data plane of numbers.

Suppose the data set is a collection of seismograms uniformly sampled in space. In other words, the data is numbers in a (t, x)-plane. For example, the following filter destroys any wavefront aligned along the direction of a line containing both the "+1" and the "−1".

$$\begin{array}{cc} -1 & \cdot \\ \cdot & \cdot \\ \cdot & 1 \end{array} \tag{1.1}$$

The next filter destroys a wave with a slope in the opposite direction:

$$\begin{array}{cc} \cdot & -1 \\ 1 & \cdot \end{array} \tag{1.2}$$

Convolving the above two filters together, we get

$$\begin{array}{ccc} \cdot & -1 & \cdot \\ 1 & \cdot & \cdot \\ \cdot & \cdot & 1 \\ \cdot & -1 & \cdot \end{array} \tag{1.3}$$

which destroys waves of both slopes.

A **two-dimensional filter** that can be a **dip-rejection filter** like (1.1) or (1.2) is

$$\begin{array}{cc} a & \cdot \\ b & \cdot \\ c & 1 \\ d & \cdot \\ e & \cdot \end{array} \tag{1.4}$$

where the coefficients (a, b, c, d, e) are to be estimated by least squares in order to minimize the power out of the filter. (In the filter table, the time axis runs vertically.)

Fitting the filter to two neighboring traces that are identical but for a time shift, we see that the filter coefficients (a, b, c, d, e) should turn out to be something like $(-1, 0, 0, 0, 0)$ or $(0, 0, -.5, -.5, 0)$, depending on the dip (stepout) of the data. But if the two channels are not fully coherent, we expect to see something like $(-.9, 0, 0, 0, 0)$ or $(0, 0, -.4, -.4, 0)$. To find filters such as (1.3), I adjust coefficients to minimize the power out of filter shapes like

$$\begin{array}{ccc} v & a & \cdot \\ w & b & \cdot \\ x & c & 1 \\ y & d & \cdot \\ z & e & \cdot \end{array} \tag{1.5}$$

With 1-dimensional filters we think mainly of power spectra and with 2-dimensional filters we can think of temporal spectra and spatial spectra. What is new, however, is that in two dimensions we can think of dip spectra, (which is when a 2-dimensional spectrum has a particularly common form, namely when energy organizes on radial lines in the (ω, k_x)-plane). As a short (three term) 1-dimensional filter can devour a sinusoid, we have seen that simple 2-dimensional filters can devour a small number of dips. This is interesting because, as we will see and as you might suspect from our 1-dimensional studies, simple 2-D and 3-D filters can help us fill in missing data in a plane or a volume. The basic problem of observational seismology is that we do not have enough receivers.

1.1.1 Prediction-error filters in two dimensions

Knowledge of an autocorrelation function is equivalent to knowledge of a spectrum. The two are simply related by Fourier transform. A spectrum or an autocorrelation function encapsulates an important characteristic of a signal or an image. Generally the spectrum changes slowly from place to place although it could change rapidly. Of all the assumptions we could make to restore missing data, one that people usually find easiest to agree with is that the spectrum should be the same in the missing data area as where the data is known. In practice we deal with neither the spectrum nor its autocorrelation but with a third object. This third object is easily transformed to the spectrum. This third object is the Prediction Error Filter (PEF) whose spectrum is inverse to the data spectrum. If the spectrum remains constant as we move from one area to another, then so does the PEF. We have seen how to define the PEF in one dimension. Now we need to see how to define it in higher dimensional space.

We will find that the half disk shape in Figure 1.1 has the desired whitening property and from that proof we will discern what other shapes are possible and impossible for a multidimensional PEF.[1] The conclusion will be that the semidisks in Figure 1.1 can be reflected in any axis or rotated by multiples of 90°. We begin the proof that a PEF whitens its input by reviewing the theorem that when energy is minimum, the **residual** is **orthogonal** to the fitting functions.

Typically we begin from a Physics book with equations written in the continuum, say

$$d(x,t) \quad = \quad \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} \alpha_{ij}\ c_{ij}(x,t) \tag{1.6}$$

Instead of continuous functions of x and t, we represent the functional dependence as components in the abstract vectors $\mathbf{d}$ and $\mathbf{c}$. Instead of a double summation to infinity, we write out three terms explicitly so instead of an equation, we have a regression like $\mathbf{d} \approx \sum_{i=1}^{3} a_i \mathbf{c}_i$. We define the residual $\mathbf{r}$ by

$$\mathbf{r} \quad = \quad a_1\mathbf{c}_1 + a_2\mathbf{c}_2 + a_3\mathbf{c}_3 - \mathbf{d} \tag{1.7}$$

The fitting functions are the column vectors $\mathbf{c}_1$, $\mathbf{c}_2$, and $\mathbf{c}_3$. Let us verify only that the dot product $\mathbf{r}\cdot\mathbf{c}_2$ vanishes showing that those two vectors are orthogonal. Energy minimum is found by

$$0 \quad = \quad \frac{\partial}{\partial a_2}\,\mathbf{r}\cdot\mathbf{r} \quad = \quad 2\,\mathbf{r}\cdot\frac{\partial \mathbf{r}}{\partial a_2} \quad = \quad 2\,\mathbf{r}\cdot\mathbf{c}_2 \tag{1.8}$$

which shows the residual is orthogonal to the fitting function. Now let us think about what this means if each column contains a lagged time series, a different lag for each column. In the prediction-error filtering application, the output is the residual and the input is time-shifted signal copies in the $\mathbf{c}$ vectors. The orthogonality says the crosscorrelation of the time series and the residual vanishes at lags where there are filter coefficients. Next we switch from time series to two-dimensional functions.

[1] I learned this theorem and its proof from John P. **Burg**. It is unpublished, as far as I know, except in my 1992 textbook, PVI.

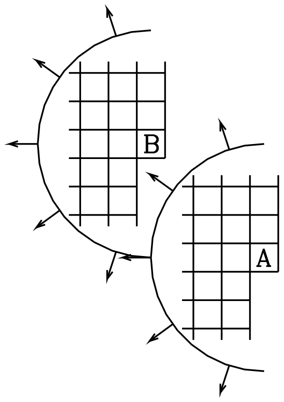


Figure 1.1: A 2-D whitening filter template, and itself lagged. At output locations "A" and "B" the filter coefficient is constrained to be "1". Viewing the semicircles as having infinite radius, the B filter is contained in the A filter. Since the output at A is orthogonal to all its inputs which include all inputs of B, the output at A is orthogonal to the output of B. md-whitepruf [NR]

Figure 1.1 shows the input plane with a 2-D filter on top of it at two possible locations. The filter shape is a semi disk which you should imagine being of infinitely large radius. Notice that semidisk A includes all the points in B. The output of disk A will be shown to be orthogonal to the output of disk B. Conventional least squares says the coefficients of the filter are designed so that the output of the filter is orthogonal to each of the inputs to that filter (except for the input under the "1" because any nonzero signal cannot be orthogonal to itself). Recall that if a given signal is orthogonal to each in a given group of signals, then the given signal is orthogonal to all linear combinations within that group. The output at B is a linear combination of members of its input group which is included in the input group of A which are already orthogonal to A. Therefore the output at B is orthogonal to the output at A. That means a particular lag of the output **autocorrelation** function vanishes. In summary,

residual	$\perp$	fitting function
output at A	$\perp$	each input to A
output at A	$\perp$	each input to B
output at A	$\perp$	linear combination of each input to B
output at A	$\perp$	output at B

(Where does this all break down if we use full disks instead of semi disks? What happens then is that full disk B includes the "1" in disk A. Since the "1" is not adjustable to make the residual orthogonal to the fitting function, there is no reason for it to be orthogonal, indeed, the output at A cannot be orthogonal to its input there, so it cannot be orthogonal to *all* of the inputs to B so it cannot be orthogonal to the output at B. Thus the autocorrelation function is not zero at the vector lag between A and B. I can prove for **interpolation-error filter**s in one dimension that the spectrum of the output is inverse to the spectrum of the input, but I will not show this proof because I have no plan to use interpolation-error filters in *any* dimensional space.)

Study Figure 1.1 to see for what lags all the elements of the B filter are wholly contained in the A filter. These are the lags where we have shown the output autocorrelation vanishing. Notice another set of lags where we have proven nothing (where B is moved to the right of A). Autocorrelations are centrosymmetric, which means that the value at any lag is the same as the value at the negative of that lag, even in 2-D and 3-D where the lag is a vector quantity. Above we have shown that a halfplane of autocorrelation values vanish. By the centrosymmetry, the other half must vanish too. Thus the autocorrelation of the PEF output is an impulse function, so its 2-D spectrum is white.

Now we can understand why the proper filter form is not a square with the "1" on the corner. For a spectrum to be white, *all* nonzero autocorrelation lags must be zero valued. If the filter were a quarter plane, then the symmetry of autocorrelations would only give us vanishing in another quarter, so there would be two remaining quarter planes where the autocorrelation was not zero.

Fundamentally, the white output theorem requires a one-dimensional ordering to the values in a plane or volume and the same filter must apply no matter where it is applied. Additionally, the filter must contain a halfplane of values so that symmetry gives the other half. It seems we could design whitening autoregression filters for 45° rotations, and we could also design them for hexagonal coordinate systems.

1.1.2 3-D prediction-error filter shape

Imagine a 3-D PEF and put a plane through the "1.0" which has an infinitesimal tilt so as to get all zeros on one side of the plane and all regression coefficients on the other. Subroutine `setfree()` (page 5) builds a mask of "0"s and "1"s to tell us where the constrained coefficients are and where are the free ones (the coefficients to be adjusted to minimize power output). We take the "1.0" to be the value `afree(lag1,lag2,lag3)=1`. The algorithm first sets all coefficients free, then constrains those in the half volume `i3<lag3`. (Since zero coefficients don't do anything useful, efficiency suggests taking `lag3=1`.) Next along the interface plane between the two half spaces it constrains those in the half plane before the "1.0" namely, `i3=lag3, i2<lag2`. Finally, in the plane that is half zero and half nonzero, along the line that divides the halfplanes we find the "1.0" and then set zeros along the halfline in one direction.

Figure 1.2 shows the volume of free coefficients of the **three-dimensional 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)
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.
niter = 0
do i1=1,a1{  do i2=1,a2{  do i3=1,a3{  if( afree(i1,i2,i3) > 0.)  niter=niter+1
```

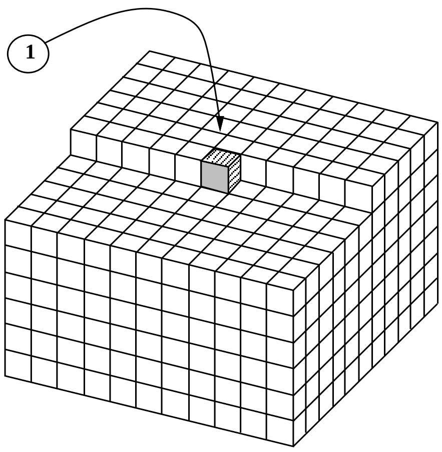


Figure 1.2: Three-dimensional prediction-error filter. md-3dpef [NR]

```
                                        }}}
   return; end
```

A reasonable arrangement for a small filter is `a1=5 a2=3 a3=2 lag1=1+a1/2=3 lag2=1+a2/2=2 lag3=1` for which the output of `setfree()` (page 5) is

```
0  0  1        1  1  1
0  0  1        1  1  1
0  2  1        1  1  1
0  1  1        1  1  1
0  1  1 ,      1  1  1
```

where `i1` runs vertical, `i2` runs horizontal in a block, `i3` jumps blocks to the right, and before printing, I reset to 2 the value `afre(lag1,lag2,lag3)=2`.

1.2 TWO-STAGE DATA INFILL

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 arise from recursive use of the prediction filter. Instability is easily avoided by using a second stage of linear least squares to solve directly for the missing data. (This is not as efficient as recursion, however.)

Here I show code to restore **missing data** in a 2-D data plane by a two-stage linear least squares process. In the first stage we fit a 2-D prediction-error filter (PEF) to the given plane. 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, we take the PEF as known and find missing elements in (or beyond) the plane to minimize the power out of the prediction-error filter. This two-stage method avoids the nonlinear problem we would otherwise face if we included the regression equations containing both free data values and free filter values. Presumably, after two stages of linear least squares we are close enough to the final solution that we could switch over to the full nonlinear setup.

The 3-D subroutine `setfree()` (page 5) works fine for 2-D problems. The 2-D codes here are easy to extend to 3-D. The 3-D versions are in the CD-ROM library, and are actually better to use for 2-D problems than the 2-D routines because I added input parameter validity verification in 3-D routines. (The filter should be shorter than the data. The `lag` value should be on the filter.)

1.2.1 Quarry blast 3-D seismic data infill

Figure 1.3 shows an example of two-stage data infill. The data was recorded at **Stanford University** with a 13×13 array of independent recorders. The figure shows 12 of the 13 lines each of length 13. Our main goal was to measure the ambient night-time noise. By morning about half the recorders had dead batteries but the other half recorded a wave from a quarry blast. The raw data was distracting to look at because of the many missing traces so I interpolated it using parameters for the small filter mentioned with subroutine `setfree()` (page 5).

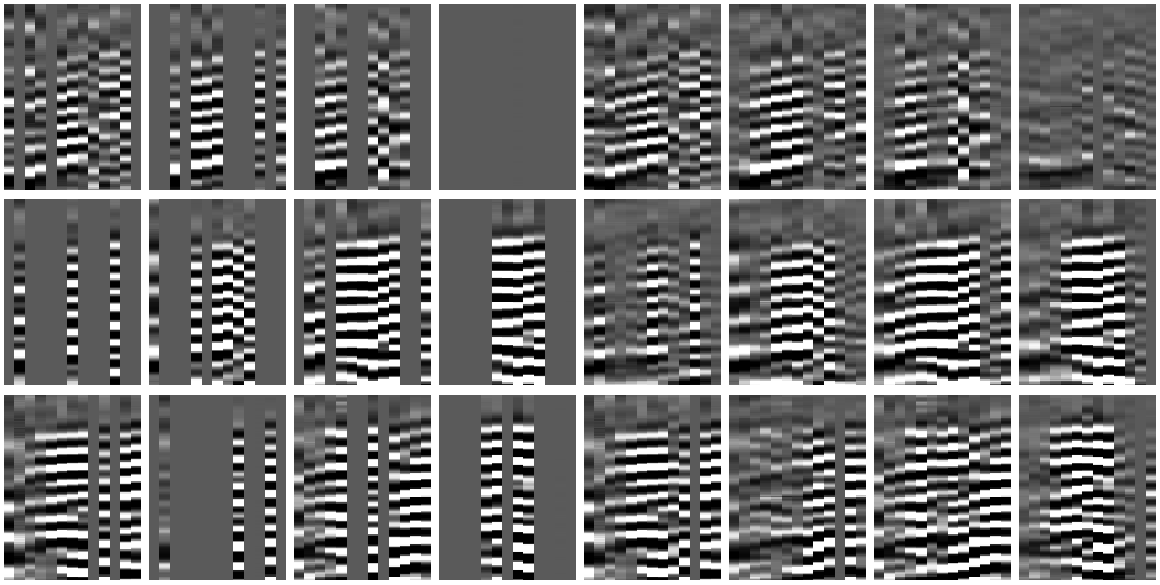


Figure 1.3: The left 12 panels are the inputs. The right 12 panels are outputs. md-passfill [NR]

1.2.2 Potential applications of two-stage infill

There are many applications of two-stage data infill that I have hardly begun to investigate.

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 Matthias **Schwab**.

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.

Obviously, the methodology and code in this chapter is easily extendable to four dimensions (prestack 3-D data).

1.2.3 Two-dimensional convolutions, transient and internal

In a later chapter we will break a time series into overlapping segments, separately analyze the segments, and put everything back together. To allow the segments to be short we will use a convolution with no end transients. Input data is not assumed zero beyond the interval that the data is given. The adjoint is where the crosscorrelation does not include hypothetical zero values off ends of the signal. This is important later where crosscorrelations change with time and they are handled as constant in short time windows. Care must be taken that zero signal values not be presumed off the ends of the short time windows otherwise the many ends of the many short segments can overwhelm the results. To keep the filter output from being affected by input data off its ends, the output signal must be shorter than the input. In practice, the input and output are allocated equal space. Since the output is shorter than the input, it can slide around in its allocated space, so its location is specified by an additional parameter called its `lag`. We coded this already in one dimension in subroutine `icaf1()` (page ??).

Two dimensional convolution is just like one-dimensional convolution except that it applies to two axes. The extension is easy, just another subscript and extra loops for it. Subroutine icaf2() does the job for internal convolution where the adjoint is the filter. It is what we need when estimating filters.

```
# Internal Convolution, Adjoint is the Filter. 2-dimensional
#
subroutine icaf2( adj,add, lag1,lag2, xx,n1,n2,  aa,na1,na2,  yy )
integer           adj,add, lag1,lag2,    n1,n2,     na1,na2
real                                  xx(n1,n2), aa(na1,na2), yy(n1,n2)
integer                                  x1,x2,      a1, a2,     y1,y2
call adjnull(     adj, add,                      aa,na1*na2,  yy,n1*n2)
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) = yy( y1,y2) + aa( a1,a2) * xx( x1,x2)
        else
                        aa( a1,a2) = aa( a1,a2) + yy( y1,y2) * xx( x1,x2)
        }}  }}
return; end
```

When estimating missing data, the adjoint is the input, not the filter. For missing data I generally use transient end conditions as done in one dimension by tcai1() (page ??). Two-dimensional **transient convolution** with the input being adjoint is done by subroutine tcai2() (page 9).

```
# Transient Convolution Adjoint is the Input. 2-D.        real yy(m1+n1,m2+n2)
#
subroutine tcai2( adj,add, aa,m1,m2,  xx,n1,n2,  yy )
integer           adj,add,    m1,m2,     n1,n2
real                       aa(m1,m2), xx(n1,n2), yy( m1+n1 , m2+n2)
integer                       a1,a2,     x1,x2,            y1,y2
call adjnull(     adj,add,            xx,n1*n2,  yy,(m1+n1)*(m2+n2) )
do a2=1,m2 {  do x2=1,n2 {  y2 = x2 + a2 - 1
do a1=1,m1 {  do x1=1,n1 {  y1 = x1 + a1 - 1
        if( adj == 0 )
                        yy(y1,y2) = yy(y1,y2) + xx(x1,x2) * aa(a1,a2)
        else
                        xx(x1,x2) = xx(x1,x2) + yy(y1,y2) * aa(a1,a2)
        }}  }}
return; end
```

The library contains three-dimensional versions of tcai2() and icaf2() named tcai3() and icaf3(), and I think you will agree that the extensions are so obvious that they need not be printed here.

1.2.4 Finding the missing data

Finding missing data requires a filter. In this subsection we will assume the filter is known. In the next we will see how to find the filter. Except for using a 2-D convolution operator instead of a 1-D one, the theory is all the same in 2-D as it was in 1-D in chapter ??, equations (??) to (??) and there is no reason to repeat it. Subroutine mis2() (page 9) is a two-dimensional example of the one-dimensional subroutine mis1() (page ??). (In **PVI** my missing data solvers were based on internal convolution but experience showed it better to code missing data problems with transient end conditions like in subroutine tcai2() (page 9).)

```
# Given PEF=aa() and data=xx with zeros missing,  find xx = known + missing
#
subroutine mis2( xfre, xx, x1,x2, niter, aa,a1,a2)
integer                   x1,x2, niter,    a1,a2,  x12, r12
integer                   i1,i2,  iter
real     xfre(x1,x2),  xx( x1,x2),       aa(a1,a2)
temporary real                              rr( (x1+a1)*(x2+a2))
temporary real          dx( x1,x2),         dr( (x1+a1)*(x2+a2))
temporary real          sx( x1,x2),         sr( (x1+a1)*(x2+a2))
                       x12= x1*x2;          r12=(x1+a1)*(x2+a2)
        call tcai2( 0,0, aa,a1,a2,  xx,x1,x2,  rr)
do iter= 0, niter {
        call tcai2( 1,0, aa,a1,a2,  dx,x1,x2,  rr)
        do i1= 1, x1 {
        do i2= 1, x2 {  dx(i1,i2) = dx(i1,i2) * xfre(i1,i2)
                        }}
        call tcai2( 0,0, aa,a1,a2,  dx,x1,x2,  dr)
        call cgplus( iter,  x12, xx,dx,sx,
                            r12, rr,dr,sr )
        }
return; end
```

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 filter parameters. 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 (thus calling for accelerating numerical techniques). I find where gaps are small, they fill in quickly. Where the gaps are large, they don't, and more iterations are required.

1.2.5 Finding the prediction-error filter

The first stage of the least-squares estimation is computing the **prediction-error filter**. The second stage will be using it to find the missing data. The input data space contains a mixture of known data values and missing unknown ones. For the first stage of finding the filter, we generally have many more regression equations than we need so we can proceed by ignoring the regression equations that involve missing data values. We ignore them everywhere the missing inputs hit the filter, either by hitting the filter's "1" or by hitting the filter's adjustable values. To identify these less desireable regression equations we prepare a **mask** from two other masks. First we make a mask plane `dfre` the size of the data plane with mask values +1.0 where data is missing and 0.0 where data is known. Likewise we make a filter mask `afre` equal +1.0 where there could be nonzero filter coefficients and equal 0.0 where filter coefficients are zero. (This mask is almost the same as `afre` previously defined, but we also need to reject equations where the filter's "1" hits a missing data value. Such a regression equation might be linear but could involve both the unknown filter and also a missing data value under the filter's "1". Here we seek only regression equations for the filter.)

Subroutine `pef2()` is basically a two-dimensional version of the one-dimensional prediction-error subroutine `pef1()` (page ??) with the new feature of ignoring missing data. In one dimension we made a special subroutine `wicaf1()` (page ??) to compound diagonal weighting with convolution. In pef2() (page 11) wherever subroutine icaf2() (page 9) appears, we also see a call to subroutine diag() (page ??) which is a diagonal matrix multiply of the weighting function (which is zero or unity depending on if data is missing). Where the output mask 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.

Near the beginning of subroutine pef2() (page 11) we convolve the free masks using icaf2() just as a few lines later the filter and the data are convolved.

```
# Find prediction-error filter avoiding missing data.
#
subroutine pef2( data,     rr,n1,n2, niter, lag1,lag2,afre,        aa,a1,a2 )
integer                       n1,n2, niter, lag1,lag2,                  a1,a2
integer                       i1,i2, a12, n12,    iter
real           data(n1*n2),rr(n1*n2),                afre(a1,a2), aa(a1,a2)
temporary real dfre(n1*n2), da( a1,a2), dr( n1*n2), ww( n1*n2)
temporary real              sa( a1,a2), sr( n1*n2), wr( n1*n2)
              a12= a1*a2;             n12= n1*n2
call null( rr, n12);   call null( aa, a12);                 aa(lag1,lag2) = 1.
do i1=1,n12 { if( data(i1)==0.0)  dfre(i1) = 1.
              else                dfre(i1) = 0.}          afre(lag1,lag2) = 1.
call         icaf2( 0, 0, lag1,lag2, dfre,n1,n2, afre,a1,a2, ww )
                                                          afre(lag1,lag2) = 0.
do i1= 1, n12
        if( ww(i1) > 0.0) { ww(i1) = 0.0 }      # free data hits nonzero filter.
        else              { ww(i1) = 1.0 }
call         icaf2( 0, 0, lag1,lag2, data,n1,n2,  aa,a1,a2, rr )
call         diag(  0, 0,        ww, n12,              rr, wr )
do iter= 0, niter {
        call diag(  1, 0,        ww, n12,              dr, wr )
        call icaf2( 1, 0, lag1,lag2, data,n1,n2,  da,a1,a2, dr )
        do i2= 1, a2
        do i1= 1, a1 {  da(i1,i2) = da(i1,i2) * afre(i1,i2)}
        call icaf2( 0, 0, lag1,lag2, data,n1,n2,  da,a1,a2, dr )
        call diag(  0, 0,        ww, n12,              dr, dr )
        call cgplus( iter,  a12 , aa,da,sa,
                            n12 , wr,dr,sr )
        }
call         icaf2( 0, 0, lag1,lag2, data,n1,n2,  aa,a1,a2, rr )
return; end
```

The codes here do not address the difficulty that maybe too much data is missing so that all weights are zero. We could add stabilization by supplementing the data volume with a synthetic "training dataset" or by a "prior filter". With things as they are, if there is not enough data to specify a prediction-error filter, you will encounter the error exit from cgplus() (page ??).

The synthetic data in Figure 1.4 is a superposition of two plane waves of different directions each with a random (but low passed) waveform. After punching a hole in the data, we find the lost data is pleasingly restored, though a bit weak near the side boundary. I do not know whether the imperfect restoration results from the side boundary behavior of the operator or from an insufficient number of missing-data iterations.

I think you will agree that the 2-D subroutines pef2() and mis2() are easy to generalize to 3-D and it would be wasteful to print them here. They are in the CD-ROM library named pef3() and mis3().

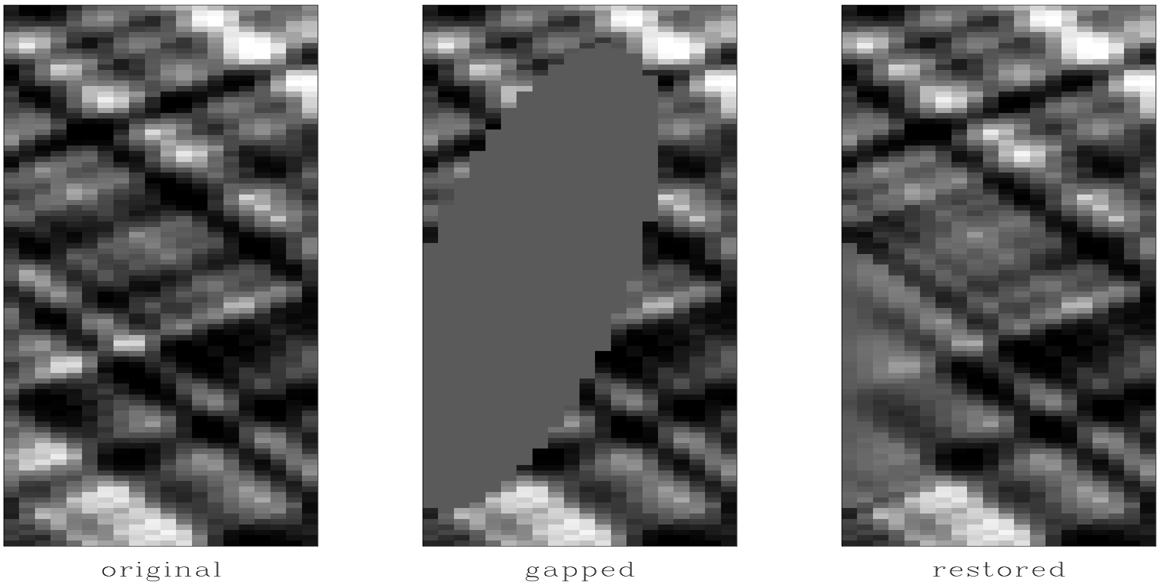


Figure 1.4: Original data (left), with a zeroed hole (center), and as restored (right). md-hole [NR]

Figure 1.5 was made with a $3 \times 3 \times 3$ filter. Since making that figure I switched my missing data solvers from **internal convolution** to **transient convolution**.

1.2.6 Imposing prior knowledge of symmetry

Reversing a signal in time does not change its autocorrelation. In the analysis of stationary time series, it is well known (**FGDP**) that the filter for predicting forward in time should be the same as that for "predicting" backward in time (except for time reversal). With short data samples, however, a different filter may be found for predicting forward than for backward. Rather than average the two filters directly, the better procedure is to find the filter that minimizes the sum of power in two residuals, one a filtering of the original signal, and the other a filtering of a time-reversed signal as in equation (1.9) where the top half of the equations represent prediction error predicting forward in time and the second half prediction backward.

$$\left[\begin{array}{c} r_1 \\ r_2 \\ r_3 \\ r_4 \\ \hline r_5 \\ r_6 \\ r_7 \\ r_8 \end{array}\right] \quad = \quad \left[\begin{array}{ccc} y_3 & y_2 & y_1 \\ y_4 & y_3 & y_2 \\ y_5 & y_4 & y_3 \\ y_6 & y_5 & y_4 \\ \hline y_6 & y_5 & y_4 \\ y_5 & y_4 & y_3 \\ y_4 & y_3 & y_2 \\ y_3 & y_2 & y_1 \end{array}\right] \left[\begin{array}{c} 1 \\ a_1 \\ a_2 \end{array}\right] \tag{1.9}$$

In an (x, y) space, you might also expect rotational symmetry. In an (x, t) space,

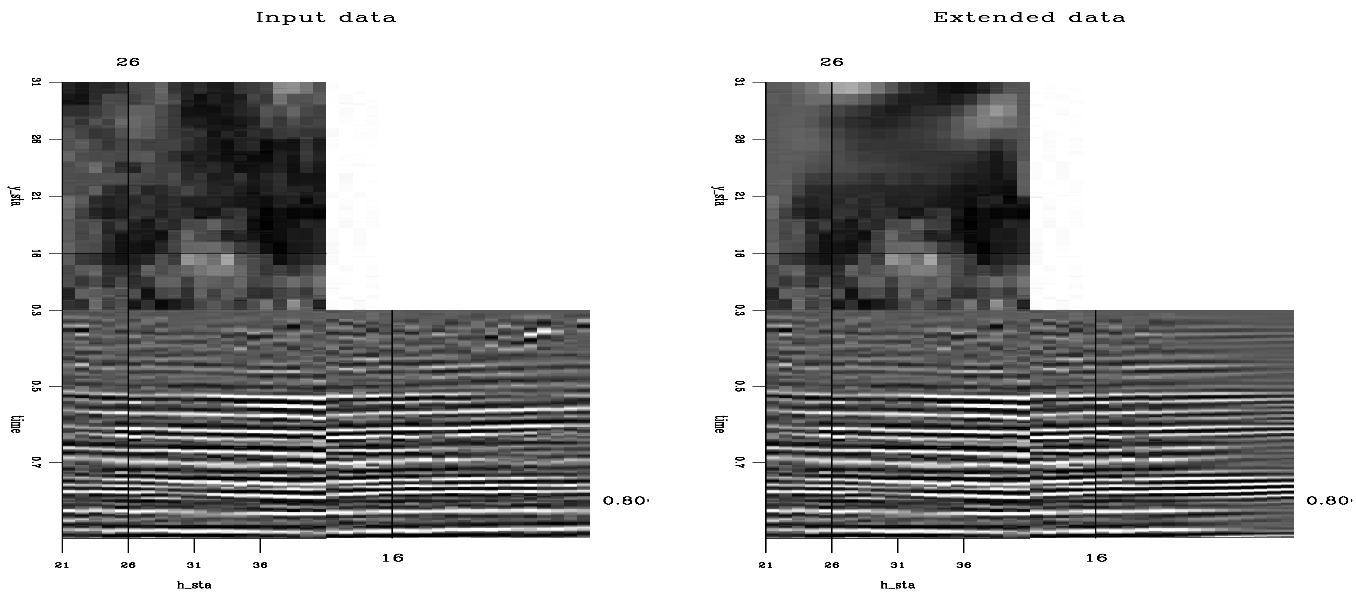


Figure 1.5: Original cube on left. The back half of the cube was chopped off. Right shows the reconstruction. md-fullcube [NR]

dips are preserved if t-axis reversal is accompanied by x-axis reversal. Many of my research codes include these symmetries, but I excluded them here. Nowhere did I prove the symmetry presumption made a large difference but in Fortran it always adds noticible clutter, expanding the residual to a two component *residual array*, which as usual needs to have three copies allocated.

Where a data sample grows exponentially towards the boundary, I expect extrapolated data would diverge too. This is prevented however by basing missing data estimation on *transient* convolution (instead of *internal* convolution). With **transient convolution** extrapolations are forced to zero beyond a known specified distance.

EXERCISES:

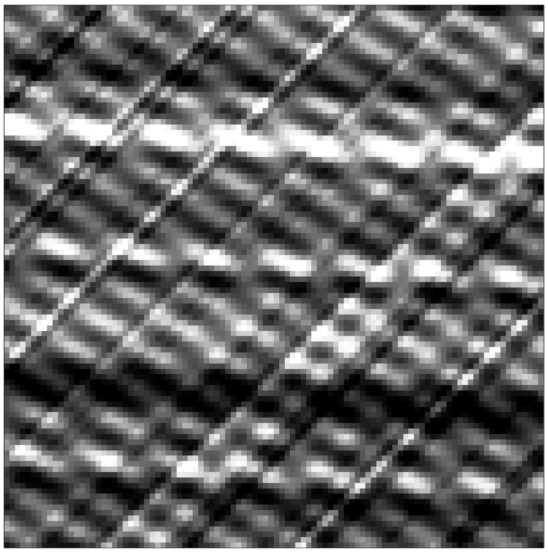
1 A special case of 2-D filtering is when the input and output are 2-D planes but the filter itself is 1-D (because either `a2=1` or `a1=1`). What makes such deconvolution different than typical 1-D deconvolution?

2 Two filters, $\mathbf{a}$ and $\mathbf{b}$, are convolved on a data plane $\mathbf{x}$ to get an output $\mathbf{y} = \mathbf{a} * \mathbf{b} * \mathbf{x}$ or $\mathbf{y} = \mathbf{b} * \mathbf{a} * \mathbf{x}$. Linearize to express $\Delta\mathbf{y}$ as a function of $\Delta\mathbf{a}$ and $\Delta\mathbf{b}$. Take the $\mathbf{a}$ filter to be a 1-D filter along the 1-axis and the $\mathbf{b}$ filter to be a 1-D filter along the 2-axis. Using your favorite 2-D filter subroutine, write a linear operator subroutine for relating $\Delta\mathbf{y}$ to the pair $(\Delta\mathbf{a}, \Delta\mathbf{b})$. Include the adjoint.

3 What is the similarity and difference of the solution to the previous exercise to 1-D deconvolution on the 1-axis followed by 1-D deconvolution on the 2-axis?

1.3 TWO PEFS FOR CROSSING-TRACK DATA

Waveforms are commonly recorded on crossing lines such as shown in Figure 1.6. Obviously we can fill in between lines by minimizing the power out of a Laplacian operator but it is better to minimize the power out of a prediction-error filter (PEF). In practice the problem is, where do we get the PEF? Figure 1.6 shows an example of sparse tracks except that it is not realistic in the upper left corner (where it will be used for testing) in a quarter circular disk where the data covers the model densely. Such a dense region is ideal for determining the PEF. Indeed, subroutine `pef2()` (page 11) cannot determine a 2-D PEF from the sparse data lines because any place you put the filter (unless there are enough adjacent data lines) unknown filter coefficients will multiply missing data so every regression equation is nonlinear and hence abandoned by the algorithm.



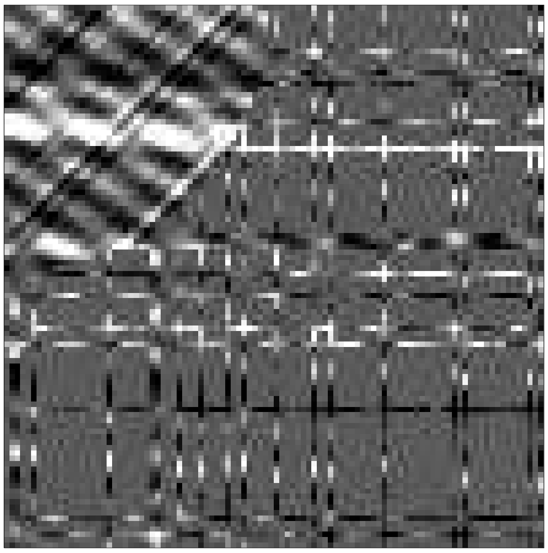


Figure 1.6: Synthetic wavefield (left) and as observed over survey lines (right). The wavefield is a superposition of waves from three directions. md-duelin [NR]

The test data set shown in Figure 1.6 is a superposition of three functions that I call plane waves. One plane wave looks like low-frequency horizontal layers. Notice that the various layers vary in strength with depth. The second wave is dipping about 30° down to the right and its waveform is perfectly sinusoidal. The third wave dips down 45° to the left and its waveform is bandpassed random noise like the horizontal beds. These waves will be handled differently by different processing schemes, so I hope you can identify all three. If you have difficulty, view the figure at a grazing angle from various directions.

Before we use the dense data region and before we leap to the nonlinear regression, there are a few tricks that get us started in the right direction. (When we do try a nonlinear approach it will be helpful and often essential to have a good starting guess.) The idea is simply this: If it is too hard to estimate a 2-D PEF, instead we estimate a 1-D PEF, or better yet, we estimate a 1-D PEF for east-west tracks and a second 1-D PEF for north-south tracks. Let $\mathbf{A}$ be the east-west PE operator and $\mathbf{B}$ be the north-south operator and let the signal or image be $\mathbf{h} = h(x, y)$. The regressions we will fit are

$$\begin{array}{lcl} \mathbf{0} & \approx & \mathbf{T}(\mathbf{h} - \mathbf{d}) \\ \mathbf{0} & \approx & \mathbf{A}\ \mathbf{h} \\ \mathbf{0} & \approx & \mathbf{B}\ \mathbf{h} \end{array} \tag{1.10}$$

where $\mathbf{d}$ is the data and $\mathbf{T}$ is a point-to-point mapping which I often call the tracking operator. This operator copies values from the desired image space $\mathbf{h} = h(x, y)$ to the data space. The data space could be organized in a variety of ways. For theoretical examples it is easiest to let the data space be a grid that overlies the image space. At each location the tracking operator multiplies the image by either a one or a zero depending on whether data is recorded at that location. This operation is often called masking and it can be thought of as an identity matrix with some of its ones missing. In practice, the data space is generally organized as tracks. Naturally the tracking operator subroutine is organized differently for the two kinds of data organizations. In the synthetic example here the tracking operator $\mathbf{T}$ is the masking done by subroutine `mask2()` (page 15).

```
# Find data plane dd from image plane hh by masking.
#
subroutine mask2( adj,add,  hh,n1,n2,  dd       , dknown )
integer i1,i2,    adj,add,     n1,n2
real                        hh(n1,n2), dd(n1,n2), dknown(n1,n2)
call adjnull(     adj,add,  hh,n1*n2,  dd,n1*n2)
do i2=1, n2
do i1=1, n1
        if( adj == 0)
                dd(i1,i2) =  dd(i1,i2) + hh(i1,i2) * dknown(i1,i2)
        else
                hh(i1,i2) =  hh(i1,i2) + dd(i1,i2) * dknown(i1,i2)
return; end
```

The inverse to the masking operator using the regression (1.10) is a straightforward application of conjugate gradients leading to subroutine `mask2i()` (page 15).

```
# Build image from two PEFs and data subset plane.
#
subroutine mask2i( dkno,    dd,n1,n2, niter, aa,a1,a2, bb,b1,b2, hh )
integer n12, iter,             n1,n2, niter,    a1,a2,    b1,b2,       nr,ra,rb
real          dkno(n1,n2), dd( n1,n2),       aa(a1,a2),bb(b1,b2),hh(n1,n2)
temporary real             rr( n1*n2*3)
temporary real dh( n1*n2), dr( n1*n2*3)
temporary real sh( n1*n2), sr( n1*n2*3)
               n12=n1*n2;             nr = n12*3;     ra= 1+n12;  rb= ra+n12
call null( hh, n12);    call null( rr, nr )
call             negcopy( n12, dd, rr )
do iter= 0, niter {
     call mask2(1,0,dh,n1,n2,rr,dkno); call icai2(1,1,1,1,aa,a1,a2,dh,n1,n2,rr(ra))
                                       call icai2(1,1,1,1,bb,b1,b2,dh,n1,n2,rr(rb))
     call mask2(0,0,dh,n1,n2,dr,dkno); call icai2(0,0,1,1,aa,a1,a2,dh,n1,n2,dr(ra))
                                       call icai2(0,0,1,1,bb,b1,b2,dh,n1,n2,dr(rb))
     call cgplus( iter,  n12, hh, dh,  sh,
                         nr , rr, dr,  sr )
     }
return; end
```

Figure 1.7 shows the result of using a single one-dimensional PEF along either the vertical or the horizontal axis. To make the figure I used subroutine `mask2i()` (page 15). To prepare its input, I used subroutine `pef2()` (page 11) to find the 1-D PEFs. Although `pef2()` (page 11) is designed to find 2-D PEFs it is easily invoked to find a 1-D PEF. For one figure panel I set `(a1,a2)` equal to `(7,1)` and for the other panel I it set equal to `(1,7)`. In this figure the second PEF `bb(,)` required by the subroutine is taken zero.

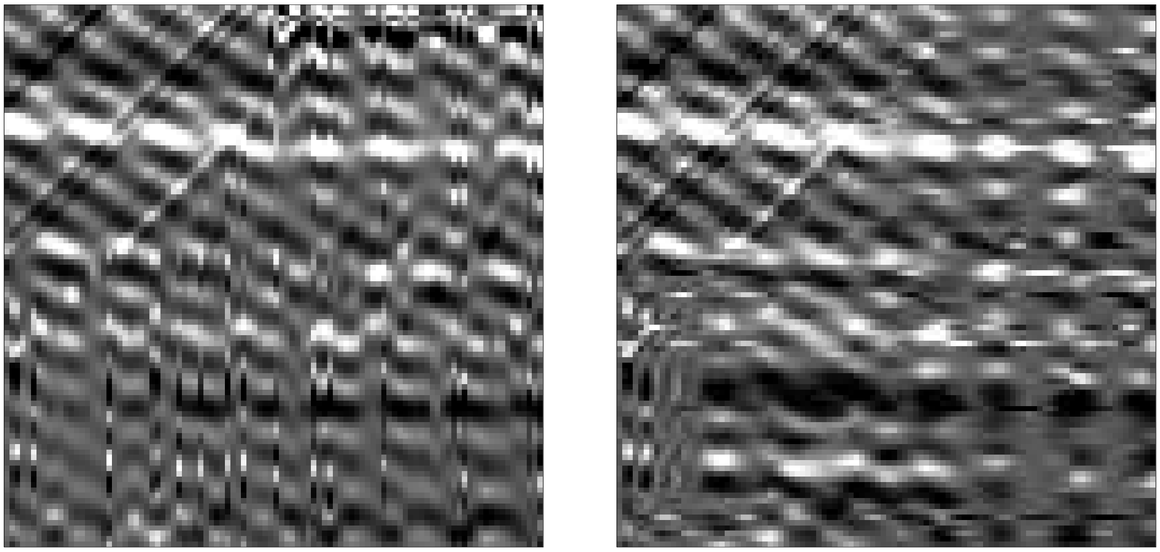
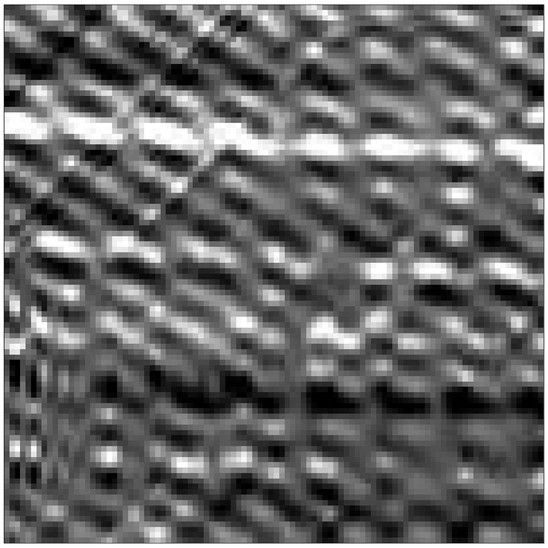


Figure 1.7: Interpolation by 1-D PEF along the vertical axis (left) and along the horizontal axis (right). md-dueleither [NR]

The same solver subroutine `mask2i()` was used for two simultaneous 1-D PEFs. For that it was loaded with `aa(7,1)` and `bb(1,7)`. Results are in Figure 1.8. Again I used the same solver for a 2-D PEF loading with `aa(7,4)` (the filter's "1" being at (4,1)) and with `bb(1,1)` being again a zero filter. To get this 2-D PEF, I used `pef2()` (page 11) and I was dependent on the cheating corner of dense data. The purpose for cheating here is to establish motivation for the more difficult task of doing the nonlinear estimation on data lines where cheating would be impossible. Figure 1.8 compares the use of a pair of 1-D PEFs versus a single 2-D PEF. Studying Figure 1.8 we conclude (what theory predicts) that

- Waves predictable with a pair of 1-D filters:
  - Horizontal (or vertical) plane wave with random waveform
  - Dipping plane wave with a sinusoidal waveform
- Waves predictable with a single 2-D filter:
  - both of the above
  - Dipping plane wave with a random waveform



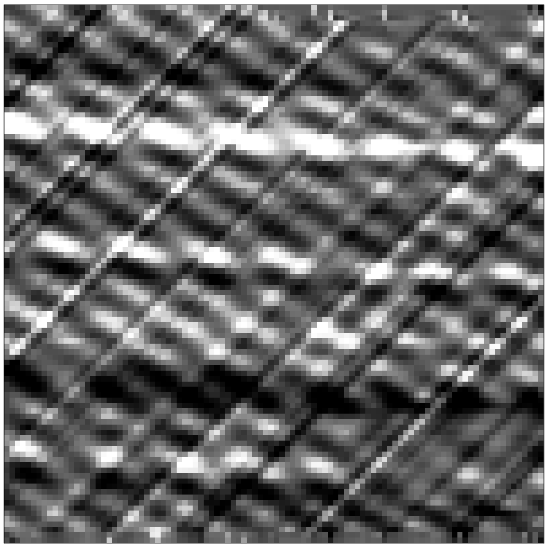


Figure 1.8: Data infilled by a pair of 1-D PEFs (left) and by a single 2-D PEF (right). md-duelversus [NR]

1.3.1 Inverse masking versus missing data fitting

The idea of inverse masking seems like the idea of missing-data fitting but their conceptual bases differ. Missing data estimation arises from the single regression

$$\mathbf{0} \approx \mathbf{A}\mathbf{h} \tag{1.11}$$

$$\mathbf{0} \approx \mathbf{A}[\mathbf{J}\mathbf{h} + (\mathbf{I}-\mathbf{J})\mathbf{h}] \tag{1.12}$$

$$\mathbf{0} \approx \mathbf{A}(\mathbf{J}\mathbf{h} + \mathbf{K}\mathbf{d}) \tag{1.13}$$

where $\mathbf{J}$ is the usual "missing-data operator" also called the data's free mask, $\mathbf{I}-\mathbf{J} = \mathbf{K}$ is the data-known mask, and where solving proceeds by iteration, implying powers of $\mathbf{AJ}$ and its adjoint. On the other hand, **masking inversion** has two or three regression equations like (1.10), say

$$\mathbf{0} \approx \mathbf{K}(\mathbf{h}-\mathbf{d}) \tag{1.14}$$

$$\mathbf{0} \approx \mathbf{A}\ \mathbf{h} \tag{1.15}$$

Solving by iteration implies powers of the column operator $\begin{bmatrix} \mathbf{K} \\ \mathbf{A} \end{bmatrix}$ and its adjoint.

The missing-data formulation assures us that the solution matches the known data exactly at all stages during the iteration. The inverse-mask formulation allows mismatch. Therefore, the mask formulation seems inferior. In practice, however, the mask formulation has many useful generalizations. When field data is stored as tracks, these tracks can cross, and when they do, the data could be inconsistent with itself. This often happens. It is not an embarrassment but an important problem deserving a reformulation (which we tackle next). Another generalization is binning where the tracking operator throws many data values into the same bin. Where data values in a bin differ, the solution (bin value) obviously cannot be consistent with each data value. Yet another generalization of the masking operator is where data is not on a regular mesh but needs to be interpolated from it.

1.4 SATELLITE ALTIMETRY SPARSE TRACKS

Earth satellites orbiting over the poles go round us in north flying tracks and south flying tracks. Orbits would be north-south lines, but the earth's rotation causes them to appear to drift westward (with the sun) so the north going runs leave northwest going tracks and the southward runs leave southwest going tracks. Figure 1.9 shows an interesting topography striated in two non-orthogonal directions (much like superposed plane waves;-) and some tracks. For my convenience I have plotted northwest tracks in a vertical direction and southwest tracks in a horizontal direction. In reality the track types are not orthogonal, so my displays will have a shearing distortion (not affecting the analysis). The satellites considered here measure **sea surface** altitude. You might think the sea surface should be at the same altitude everywhere but the gravitational attraction of the mountains beneath the sea causes the altitude of sea level to vary from place to place. It would be an arduous task to survey all the oceans to find their depth and map it, or to map the strength of gravity on all the ocean surfaces, but either map looks much like each other and both look like the satellite altimeter map, the differences being mainly in calibration and spatial filtering.

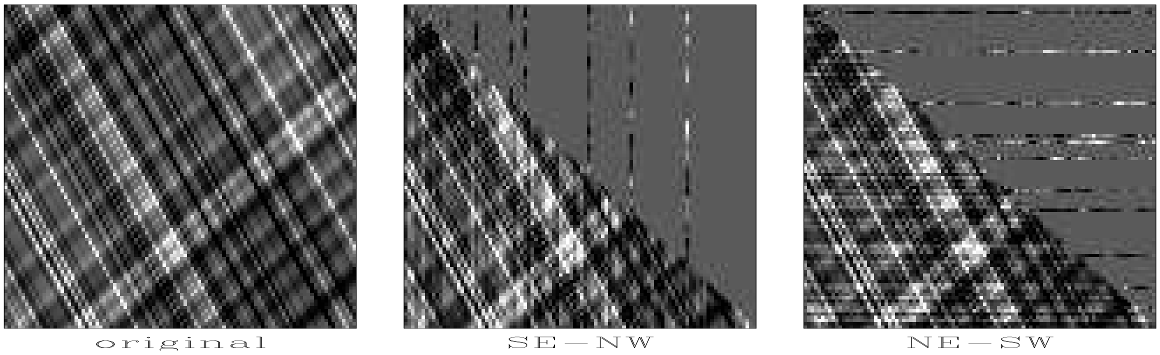


Figure 1.9: Synthetic topography (left), northward tracks (center), and southward tracks (right). The south pole is the lower left hand corner and thirty degrees south latitude is the diagonal. md-synsat [NR]

There is great track density below 30° south latitude but it is much sparser above. Most satellite tracks are missing north of 30° south latitude because knowledge of the distribution of gravity is required to guide a ballistic missile to a precise target, so they are secret. The tracks north of 30° south latitude come from other satellites.

Notice that some tracks are brighter and some are dimmer. This mimics a calibration problem that causes each track to have a different mean, or more generally, a very low frequency, along-the-track noise.

Although the geometry of Figure 1.9 is somewhat unusual, it is an attractive example for study for several reasons. First, it exhibits the familiar problem that survey lines are never as dense as we wish them to be. Second, it is not unusual for crossing lines to have inconsistency. Third, having dense coverage near sparse coverage offers us a good chance to study the transition from a region of high information density to a region of low density and ambiguity (spatial aliasing).

The obvious way to fill in between the tracks is with some kind of smoothness criterion. For example, we could have the solution fit the track data exactly and between the tracks we could have the solution satisfy Laplace's equation. In Figure 1.10, I chose instead to minimize the *energy* in the output of the **Laplace operator** (which amounts to satisfying the *squared* Laplace equation). The problem with either of these approaches is the implicit presumption of isotropy, i.e., that the interpolation be independent of orientation.

1.4.1 A deeper model than anisotropy

From the data south of 30° latitude in Figure 1.9 we see the orientation of the two **striations** (along with the third striation of along-the-track noise). But there are two orientations in the topography, not just one, so the problem is not so simple that we can merely find a skewing and stretching of coordinates that would make the image appear isotropic. The approach I advocate is to seek a partial differential equation that the image will be a solution of. In reality, a complicated image will satisfy no simple differential equation, so instead I seek a filter with minimum power out. I first find this filter in the region of good spatial coverage. Then I carry it north to the area of sparse coverage where I find the solution by the principle of minimum power out. The result is shown on the right in Figure 1.10. The result is overwhelmingly better than the isotropic approach. The result is almost perfect but the solutions might still be a little weaker in the middle of large track gaps because I limited the number of conjugate-gradient iterations to 20. (The figure computes in a few seconds.)

Figure 1.10: Adjoint reconstruction (left), reconstruction by Laplacian (center), and reconstruction by 2-D PEF (right). md-seasat [NR]

You might notice that the two striations in the topography have different textures, one being string like and the other being rod like. This means the underlying "waves" making up the striations have a different spectrum. The method used here works equally well (or badly) if the two spectra are the same. The fundamental assumption is one of superposition, an assumption that will have limited validity depending on the geological mechanism that built the topography.

1.4.2 Distribution of sparsity

We rarely have data where the transition from good coverage to poor is as abrupt as here but we generally have data that terminates even more abruptly at the edges of a survey. There is also the possibility of coverage that is uniformly sparse so we would have no good region in which to learn the data statistics. In principle we should be simultaneously estimating the solution and its multidimensional prediction-error filter. Such simultaneous estimation is nonlinear. Thus there are well-known dangers, but the problem itself is not ill-conceived or impossible to approach, as the one-dimensional example of figure ?? shows. Although real problems are nonlinear, it is often realistic to approach them textbook style, as a sequence of linearized approximations. Sometimes ingenious tricks can be brought to bear as in Figure 1.16.

Current practice in the seismic exploration industry meets the requirements perfectly of a nonlinear problem that is near to linear because the high costs limit surveys to avoid aliasing at central frequencies while allowing aliasing at the highest frequencies (which define the resolution).

1.4.3 Seasat modeling and solution formulation

The problem formulation is:

$$\mathbf{0} \approx \mathbf{A}_n(\mathbf{Th} - \mathbf{d}) \tag{1.16}$$

$$\mathbf{0} \approx \mathbf{A}_h \, \epsilon \, \mathbf{h} \tag{1.17}$$

where $\mathbf{A}_n$ is a noise whitener, $\mathbf{A}_h$ is a topography whitener (PEF), $\mathbf{d}$ is data, $\mathbf{h}$ is altitude, $\mathbf{T}$ is the operator that makes data tracks from an altitude map, and ϵ is the usual damping parameter. Figure 1.10 uses two different values of $\mathbf{A}_h$, one a Laplacian and one a PEF.

Subroutine `icai2()` (page 20) is yet another 2-D convolution program, a straightforward generalization of the 1-D subroutine `icaf1()` (page ??) but with the adjoint being the input instead of the filter. We need it to apply the spatial PEF to the topography without overflowing the edges.

```
# Internal Convolution, Adjoint is the Input.  2-dimensional
#
subroutine icai2( adj,add, lag1,lag2, aa,na1,na2,  xx,n1,n2,  yy )
integer           adj,add, lag1,lag2,    na1,na2,     n1,n2
```

```
real                                  aa(na1,na2), xx(n1,n2), yy(n1,n2)
integer                                   a1, a2,     x1,x2,     y1,y2
call adjnull(     adj,add,                          xx,n1*n2,  yy,n1*n2)
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) = yy( y1,y2) + xx( x1,x2) * aa( a1,a2)
        else
                        xx( x1,x2) = xx( x1,x2) + yy( y1,y2) * aa( a1,a2)
        }}  }}
return; end
```

Subroutine `track()` (page 21) shows how tracks are defined from topography, and it also does the adjoint reconstruction of topography from tracks. The array `dknow(,,)` is 1 where data is recorded and 0 where it is not. The subroutine also carries along the $\epsilon \mathbf{I}$ operator with subroutine `ident()` (page 21).

```
# make satellite altimeter tracks (and filtered topography) from topography hh.
#
subroutine track( adj,add, eps, hh,n1,n2,  dd         , dkno )
integer i1,i2,i3, adj,add,         n1,n2
real                        eps, hh(n1,n2), dd(n1,n2,3), dkno(n1,n2,2)
call adjnull(     adj,add,      hh,n1*n2,  dd,n1*n2*3)
do i3=1, 2
do i2=1, n2
do i1=1, n1
        if( adj == 0)
                dd(i1,i2,i3) =  dd(i1,i2,i3) + hh(i1,i2)    * dkno(i1,i2,i3)
        else
                hh(i1,i2)    =  hh(i1,i2)    + dd(i1,i2,i3) * dkno(i1,i2,i3)
call ident( adj, 1, eps, n1*n2, hh,           dd( 1, 1, 3) )
return; end
```

```
subroutine ident( adj, add, epsilon, n, pp,     qq    )
integer i,        adj, add,          n
real                       epsilon,     pp(n), qq(n)  # equivalence (pp,qq) OK
if( adj == 0 ) {
        if( add == 0 ) { do i=1,n {  qq(i) =         epsilon * pp(i) } }
        else           { do i=1,n {  qq(i) = qq(i) + epsilon * pp(i) } }
        }
else {  if( add == 0 ) { do i=1,n {  pp(i) =         epsilon * qq(i) } }
        else           { do i=1,n {  pp(i) = pp(i) + epsilon * qq(i) } }
        }
return; end
```

1.4.4 Along-track noise and crooked tracks

Subroutine `trakwit()` (page 22) applies a weighting operator $\mathbf{A}_n$ to each track. This **weighting operator** is a little unusual because it is not a scaling function but an operator. Recall the unknown mean values or low-frequency noise in the data tracks. Because of this noise there would be huge residuals if we tried to fit the solution altitude to the crossing altimeter tracks where they do not match at the point of crossing. The proper method is to filter the *residual* to eliminate the low frequency noise in the *data.* Many people filter *observed* data to eliminate noise in it, but when we come to compare such data to *modeled* data we need to filter it with the same filter. Filtering the residual gets both data sets at the same time, with an assuredly identical filter.

The simplest filter that eliminates zero frequency is the first order finite difference operator $(1, -1)$. The next simplest is an operator with a narrower spectral notch around zero frequency, namely $(1, -1/3, -1/3, -1/3)$, etc. I found both worked about equally well in this case. These are rough prior guesses of the filters that whiten the *data*. A more general solution is to form the *residuals* and then design a whitener for them. A still more general solution is to find the residuals and the PEF at the same time.

```
# Weight (filter) the track residuals.  Also adjoint.
#
subroutine trakwit( adj, aa,a1,a2,  rr, n1,n2,    wr )
integer i1,i2,nd,   adj,    a1,a2,      n1,n2,               n12, rn, rs, rh
real                      aa(a1,a2), rr( n1*n2*3), wr(n1*n2*3)
temporary real del(n1)
          nd= 5;        del( 1)=    1.
do i1 = 2, nd
                        del(i1) = - 1./( nd - 1.)
rn = 1;  n12 =  n1 * n2
rs = 1 + n12
rh = 1 + n12 + n12
call icai2( adj,0, 1,1, del, nd,1, rr(rn), n1,n2, wr(rn))
call icai2( adj,0, 1,1, del, 1,nd, rr(rs), n1,n2, wr(rs))
call icai2( adj,0, 1,1, aa,a1,a2,  rr(rh), n1,n2, wr(rh))
return; end
```

The idea behind subroutine `track()` applies also to crooked survey lines. With crooked lines, however, subroutine `trakwit()` would need modification to apply the data noise whitening filter along the actual track and not simply along the coordinate axis.

1.4.5 Seasat optimization

To prepare to solve the regression (1.17) we begin by restating it as the operator that takes a model perturbation $\Delta\mathbf{h}$ and produces the weighted residual perturbation $\Delta\mathbf{r}$.

$$\begin{bmatrix} \\ \Delta\mathbf{r} \\ \\ \end{bmatrix} \quad = \quad \begin{bmatrix} \mathbf{A}_n & \cdot & \cdot \\ \cdot & \mathbf{A}_n & \cdot \\ \cdot & \cdot & \mathbf{A}_h \end{bmatrix} \begin{bmatrix} \mathbf{N} \\ \mathbf{S} \\ \epsilon\mathbf{I} \end{bmatrix} [\Delta\mathbf{h}] \tag{1.18}$$

where we have also split the track operator $\mathbf{T}$ into two parts, $\mathbf{N}$ making the northwest tracks and $\mathbf{S}$ making the southwest tracks.

To find the best fitting topography $\mathbf{h} = h(x, y)$ we prepare the solver subroutine `seatopo()` (page 23). Begin by loading the residual vector with the negative of the data $\mathbf{r} = -\mathbf{d}$ and load the solution vector with zeros $\mathbf{h} = \mathbf{0}$. Then proceed as we did with earlier weighted regressions like `pef1()` (page ??) or `pef2()` (page 11). First we revise the data to weighted data by a single application of subroutine `trakwit()` (page 22). Then begin the usual conjugate-gradient iteration containing first the adjoint of operator (1.18) and then the forward operator. The only thing new here is that the *filter weighting* cannot have the output and input share the same computer memory as is allowed by the more usual *scalar weighting* routines `ident()` (page 21) and `diag()` (page ??) so an extra array `dr2` must be declared. (In a better computer language than Fortran (such as **C++**) there would be no need to list separate programs for each application of weighted least squares—one routine should meet all needs.)

```
# Build topography from satellite tracks and PEF of prior topo.
#
subroutine seatopo( dkno,  dd,n1,n2, niter, eps, aa,a1,a2,  hh )
integer n12, r123, iter,      n1,n2, niter,         a1,a2
real        dkno(n1,n2,2), dd(n1,n2,2)    , eps, aa(a1,a2), hh(n1,n2)
temporary real             rr( n1*n2*3)
temporary real dh( n1*n2), dr( n1*n2*3), dr2(n1*n2*3)
temporary real sh( n1*n2), sr( n1*n2*3)
temporary real             wr( n1*n2*3)
               n12=n1*n2;               r123=n1*n2*3

call null( hh, n12);    call null( rr, r123)
call         negcopy( n12 * 2, dd, rr          )
call         trakwit( 0, aa,a1,a2,          rr, n1,n2,  wr )
do iter= 0, niter {
        call trakwit( 1, aa,a1,a2,          dr, n1,n2,  wr )
        call   track( 1,0, eps,  dh,n1,n2,  dr,              dkno )
        call   track( 0,0, eps,  dh,n1,n2,  dr,              dkno )
        call trakwit( 0, aa,a1,a2,          dr, n1,n2,  dr2 )
        call cgplus( iter,  n12,  hh, dh,  sh,
                            r123, wr, dr2, sr )
        }
return; end
```

Finally, subroutine `seastat()` (page 23) builds a topographic map from satellite data tracks using either of two different statistical assumptions. The first assumption is that the 2-D prediction-error filter (PEF) for the topography is Laplace's operator. The alternate assumption is that a previous map is available which can be used to compute the PEF using `pef2()` (page 11). The previous map is presumed to be poor where the tracks are missing. To put this presumption into practice, the map is set to zero where there are no data tracks so `pef2()` will handle such regions as missing data. Then we define the template of free parameters for the filter with subroutine `setfree()` (page 5) and we call subroutine `pef2()` (page 11) to get the PEF. With the inputs now all prepared, we invoke subroutine `seatopo()` (page 23).

```
# Find the topo map from data and either a previous topo map or a PEF.
#
subroutine seastat( dd,n1,n2, niter, a1,a2, hh, laplace)
integer i1,i2,i3,      n1,n2, niter, a1,a2,     laplace,     lag1,lag2, iter
real                dd(n1,n2,2),            hh(n1,n2), eps
temporary real    dkno(n1,n2,2), afre(a1,a2)
temporary real     res(n1,n2  ),   aa(a1,a2)
eps= 1.;        lag1=1+a1/2;    lag2=1
do i3=1, 2
do i2=1,n2
do i1=1,n1  {   if( dd(i1,i2,i3) == 0.)   dkno(i1,i2,i3) = 0.0
                else                      dkno(i1,i2,i3) = 1.0
                };           call null( aa, a1*a2)
if( laplace > 0 ) {                     aa(lag1-1,2) =  .25
                aa(lag1+0,1) = .25;     aa(lag1  ,2) = -1.0;    aa(lag1,3)= .25
                                        aa(lag1+1,2) =  .25                     }
else {  do i1=1,n1
```

```
        do i2=1,n2   { if( dkno(i1,i2,1) == 0. |
                           dkno(i1,i2,2) == 0.  )   hh(i1,i2) = 0.  }
        call setfree(                       lag1,lag2, 1, afre,    a1,a2,1, iter)
        call pef2( hh, res, n1,n2,   iter, lag1,lag2,    afre, aa,a1,a2 )
        }
call seatopo( dkno, dd,n1,n2, niter, eps, aa,a1,a2,  hh )
return; end
```

1.5 SEABEAM: SEEING THE BOTTOM OF THE SEA

Figure 1.11 shows a day's worth of data[2] collected at sea by SeaBeam, an apparatus for measuring water depth both directly under a ship, and somewhat off to the sides of the ship's track. The data is measurements of depth $h(x,y)$ at miscellaneous locations in the (x,y)-plane. The locations are scattered about according to various aspects

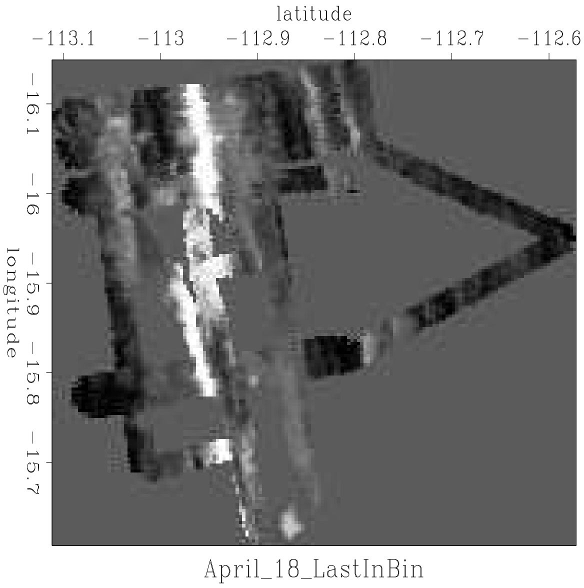


Figure 1.11: Depth of the ocean under ship tracks. md-last18 [NR]

of the ship's navigation and the geometry of the SeaBeam sonic antenna. Figure 1.11 was made by setting out a uniform mesh (160×160) in the (x,y)-plane, and copying data into the nearest mesh point. Data noise covariance could be estimated where tracks cross over themselves. More interesting are the empty mesh locations where no data is recorded. Here I left empty locations with a background value equal the mean depth $\bar{h}$. After you study the water-depth map for a while, you come to realize that at many locations, your human intuition suggests some obvious extensions of the **ocean floor ridges** (which are white) and we seek mathematical methods to guess what humans can guess, namely, a background depth $\bar{h}(x,y)$ more plausible than $\bar{h}$. (This amounts to finding and using the *model covariance*.)

[2] I'd like to thank Alistair Harding for this interesting data set named April 18.

The first thing we will need is a subroutine for two-dimensional linear interpolation. Starting from the 1-D linear interpolation subroutine `lint1()` (page ??) I prepared a 2-D counterpart in the obvious manner and named it `lint2()` (page 25).

```
# Linear interpolation 2-D
#
subroutine lint2( adj, add, o1,d1,o2,d2, xy,        mm,m1,m2,  dd, nd)
integer i, ix,iy, adj, add,                     id,     m1,m2,      nd
real    f, fx,fy, gx,gy,    o1,d1,o2,d2, xy(2,nd), mm(m1,m2), dd( nd)
call adjnull(     adj, add,                         mm,m1*m2,  dd, nd)
do id=1,nd {
        f=(xy(1,id)-o1)/d1; i=f; ix=1+i; if( 1<=ix & ix<m1){  fx=f-i; gx=1.-fx
        f=(xy(2,id)-o2)/d2; i=f; iy=1+i; if( 1<=iy & iy<m2){  fy=f-i; gy=1.-fy
                if( adj == 0)
                        dd(id) = dd(id) + gx * gy * mm(ix   ,iy   ) +
                                          fx * gy * mm(ix+1,iy   ) +
                                          gx * fy * mm(ix   ,iy+1) +
                                          fx * fy * mm(ix+1,iy+1)
                else {
                        mm(ix   ,iy   ) = mm(ix   ,iy   ) + gx * gy * dd(id)
                        mm(ix+1,iy   ) = mm(ix+1,iy   ) + fx * gy * dd(id)
                        mm(ix   ,iy+1) = mm(ix,   iy+1) + gx * fy * dd(id)
                        mm(ix+1,iy+1) = mm(ix+1,iy+1) + fx * fy * dd(id)
                        }
                }}
        }
return; end
```

Recall that this linear-interpolation operator carries us from the uniform mesh to the irregularly distributed data. What we need to do is solve the inverse problem which goes the other direction.

The simplest thing to do is minimize the power out of the Laplacian operator, which is in Figure 1.12.

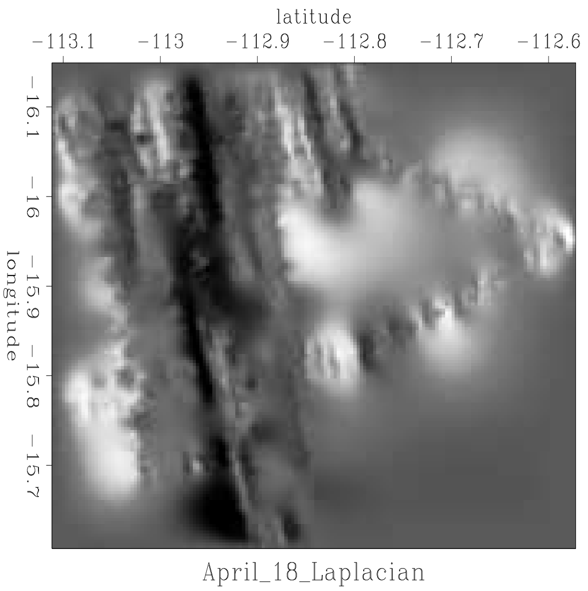


Figure 1.12: Depth of the ocean under ship tracks with missing mesh values filled by minimizing energy of Laplacian filtered mesh. md-lap18 [NR]

The problem you see with the Laplacian operator as an interpolator is that it smears things uniformly in all directions. From Figure 1.12 you see we need an anisotropic interpolation oriented along the regional trends. We should find a better filter which we could do by applying subroutine `pef2()` (page 11) to the image of Figure 1.11. After crudely throwing values in bins (either keep only the last value in the bin, or divide by the number in the bin) we could do two stages of linear least squares beginning with designing a 2-D PEF on the binned data and then using that PEF to find the missing data in the manner of Figure 1.4 or Figure 1.10. (I should do this.)

A nonlinear approach to filling in the missing data is suggested by Figures ??–?? where the PEF and the missing data are estimated simultaneously. The nonlinear approach has the disadvantage of being new, but the advantage that it allows for completely arbitrary data positioning whereas the two-stage linear approach requires a sufficient amount of the data to be on a uniform mesh without too many empty mesh locations.

For the nonlinear approach we follow the logic of the one-dimensional estimation. The mathematics, regression, and method of solution we need is the one-dimensional approach of equations (??) and (??) Expressing this theory in the present notation we have

$$\mathbf{0} \;\approx\; \mathbf{L}\Delta\mathbf{h} + (\mathbf{L}\mathbf{h} - \mathbf{d}) \tag{1.19}$$

$$\mathbf{0} \;\approx\; \mathbf{A}\;\Delta\mathbf{h} + \mathbf{H}\mathbf{K}\;\Delta\mathbf{a} + (\mathbf{A}\mathbf{h} \text{ or } \mathbf{H}\mathbf{a}) \tag{1.20}$$

where $\mathbf{d}$ is the data $d(x,y)$, $\mathbf{L}$ is linear interpolation, $\mathbf{h}$ is the modeled altitude $h(x,y)$, $\mathbf{H}$ is 2-D convolution with altitude, and $\mathbf{K}$ identity matrix but with a zero at the filter's "1".

Proceeding entirely by analogy from one dimension to two subroutine `levint1()` (page ??) leads us to subroutine `levint2()` (page 26).

```
# levint2 -- LEVeler INverse INTerpolation in 2-D while finding PEF.
#
subroutine levint2( afre,aa,a1,a2, xy,      dd,nd, o1,d1,o2,d2, mm,m1,m2, niter,warmup)
integer iter, i,            a1,a2,      nd,                         m1,m2, niter,warmup
real        afre(a1*a2), aa(a1,a2),xy(2,nd),dd(nd),o1,d1,o2,d2, mm(m1,m2)
integer nx, xa, xm, nr, rd,rm, m12, a12, am, warmup
temporary real  xx(m1*m2+a1*a2),   rr( nd+(m1+a1)*(m2+a2))
temporary real  dx(m1*m2+a1*a2),   dr( nd+(m1+a1)*(m2+a2))
temporary real  sx(m1*m2+a1*a2),   sr( nd+(m1+a1)*(m2+a2))
                nx=m1*m2+a1*a2;    nr= nd+(m1+a1)*(m2+a2);   m12=m1*m2;   a12=a1*a2
        xm=1;   xa=1+m1*m2;        rd=1;  rm=1+nd
        call copy( m12, mm, xx(xm));  call copy( a12, aa, xx(xa))
do iter= 0, niter {
        call negcopy(     nd,          dd,              rr(rd))    #       -d
        call lint2( 0, 1, o1,d1,o2,d2,xy,    mm,m1,m2, rr(rd),nd) # r_d = -d+L m
        call tcai2( 0, 0, aa,a1,a2,          mm,m1,m2, rr(rm))    # r_m =    A m

        call lint2( 1, 0, o1,d1,o2,d2,xy,dx(xm),m1,m2, rr(rd),nd) #    L'r_d
        call tcai2( 1, 1, aa,a1,a2,      dx(xm),m1,m2, rr(rm))    # Dm=L'r_d+A'r_m
        call tcaf2( 1, 0, mm,m1,m2,      dx(xa),a1,a2, rr(rm))    # Da=M'r_m

        if( iter <= warmup) do i=1,a12{ dx(m12+i)= 0. }           # Da = 0
        else                do i=1,a12{ dx(m12+i)= dx(m12+i)*afre(i)}
```

```
        call lint2( 0, 0, o1,d1,o2,d2,xy,dx(xm),m1,m2, dr(rd),nd) # Dr_d=L Dm
        call tcai2( 0, 0, aa,a1,a2,      dx(xm),m1,m2, dr(rm))    #      A Dm
        call tcaf2( 0, 1, mm,m1,m2,      dx(xa),a1,a2, dr(rm))    # Dr_m=A Dm+M Da
        call cgplus( iter, nx, xx, dx, sx,
                           nr, rr, dr, sr )
        call copy( m12, xx(xm), mm)
        call copy( a12, xx(xa), aa)
        }
return; end
```

It turns out that good results take rather many iterations. Using 1.5 hours on a fast work station gave the result shown in Figure 1.13. I used a 160×160 mesh with a 5×3 filter. As with one-dimensional examples, I did a small number (20) of initial iterations with a **Laplacian** for the data roughener. Figure 1.13 shows the final result.

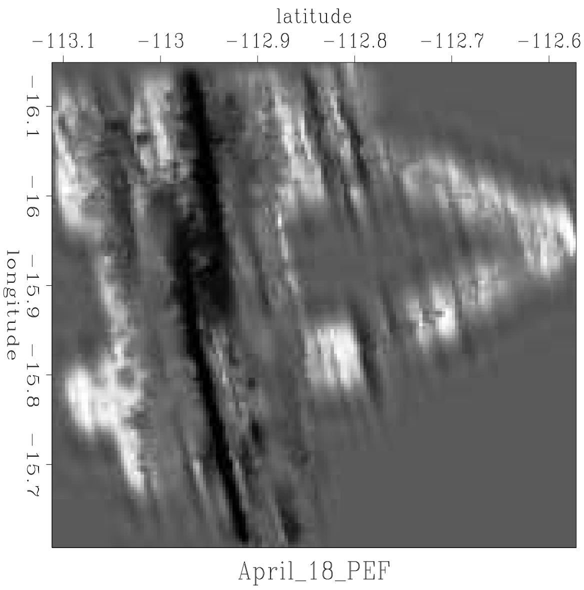


Figure 1.13: Filling missing locations at the same time as finding the PEF. md-lint18 [NR]

Figure 1.14 compares Figures 1.11-1.13.

In an earlier version of the program I ran the same number of Laplace iterations as PEF iterations. This produced an unexpected result that was forshadowed in one dimension. When the PEF iterations were started after a very large number of Laplacian iterations, the solution did not move. Evidently a local minimum was found which could not be escaped.

How can we speed calculations of this type? I suggest the following:

1. Begin on a very coarse mesh. Use any method, either a sophisticated one like `levint2()` (page 26), or an unsophisticated binning, followed by `pef2()` (page 11) to find the filter and then `mis2()` (page 9) to fill in empty bins.

2. Make a finer mesh, say twice as many pixels in both dimensions. Create an initial guess on this mesh by **pixel zoom** from the coarse mesh. Pixel zoom

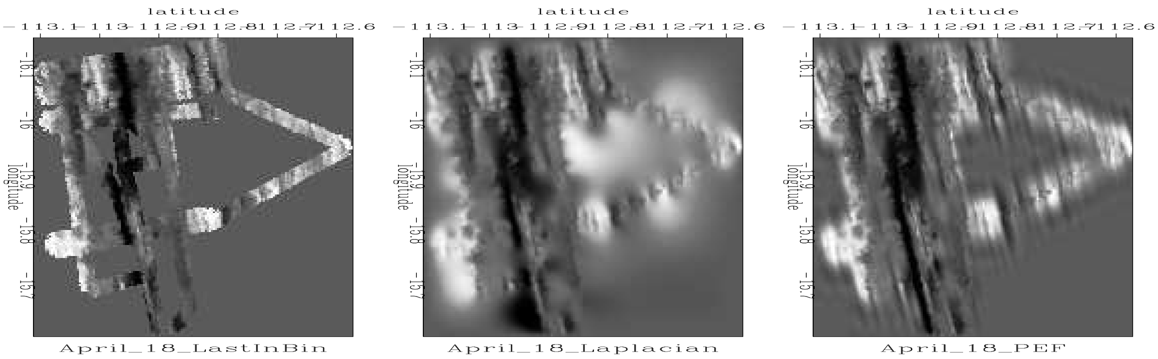


Figure 1.14: Comparison of "last in bin" (left), Laplacian (middle), PEF (left). md-compare [NR]

means that one coarse pixel is copied to its corresponding 2×2 area of pixels on the finer mesh.

3. Begin with the *same* filter on the finer mesh that you learned from the coarser mesh.
4. Update the filter and the model on the finer mesh either by two-stage least squares or by the more sophisticated `levint2()` (page 26).
5. Repeat from stage 2.

I recently revised the `levint2()` (page 26) subroutine to make it possible to restart from a given solution. Mainly this was extracting the initialization of the model mesh (I used zero in these examples) and the initial filter, the Laplacian, and the template of free variables in the filter which in this case is a PEF where the "1" aligned on a non-central point of the Laplacian operator. For the examples here, these initializations are now done before calling `levint2()` (page 26).

1.6 INTRODUCTION TO ALIASING

In its simplest form, the **Nyquist** condition says that we can have no frequencies higher than two points per wavelength. In migration, this is a strong constraint on data collection. It seems there is no escape. Yet, in applications dealing with a CMP gather we see data with spatial frequencies that exceed Nyquist and we are not bothered, because after **NMO**, these frequencies are OK. Nevertheless, such data is troubling because it breaks many of our conventional programs, such as downward continuation with finite differences or with Fourier transforms. (No one uses focusing for **stacking**.) Since NMO overcomes the limitation imposed by the simple statement of the Nyquist condition, we revise the condition to say that the real limitation is on the spectral bandwidth, not on the maximum frequency. Mr. Nyquist does not tell us where that bandwidth must be located. Further, it seems that precious bandwidth *need not be contiguous.* The signal's spectral band can be split into pieces and those pieces positioned in different places. Fundamentally, the issue is whether the total bandwidth exceeds Nyquist. Noncontiguous bandwidth arises naturally with two-dimensional data where there are several plane waves present. There the familiar spatial Nyquist limitation oversimplifies real life because the plane waves link time and space.

The plane wave model links an axis that is not aliased (time) with axes (space) that often are.

1.6.1 Aliasing a plane wave chirp

Spatial aliasing is often described as being like temporal aliasing but on another axis. Mathematically, aliasing on the two axes is independent, however, the plane-wave model links them in our day-to-day experience. The plane wave model links an axis (time) that is not aliased with axes (space) that might be.

A function generally useful with seismic vibrator sources is the "gated-**chirp** function". Within a specified time gate this function is a sinusoid with a time-variable frequency. Here we will take this signal to be defined by

$$f(\tau) \quad = \quad \begin{cases} \sin(\omega_{\max}\tau^2/2\tau_{\max}) & 0 \le \tau \le \tau_{\max} \\ 0 & \text{otherwise} \end{cases} \tag{1.21}$$

The phase ϕ and instantaneous frequency ω of this chirp signal is

$$\phi \quad = \quad 0.5\,\omega_{\max}\,\tau^2/\tau_{\max} \tag{1.22}$$

$$\omega \quad = \quad \frac{d\phi}{d\tau} \quad = \quad \omega_{\max}\,\tau/\tau_{\max} \tag{1.23}$$

which shows that as τ ranges from 0 to $\tau_{\max}$, the frequency ω ranges from 0 to $\omega_{\max}$.

Now consider the chirp signal seen on a wave, say $f(t-px)$. For constant x we see a chirp signal, possibly shifted on the time axis. For constant t we see a chirp function along the x axis, but the $-p$ causes the chirp to be reversed and stretched. Figure 1.15 shows the chirp wave function $f(t-px)$ for $p = 0.45$ and for $\omega_{\max} = 2\pi$. Since $\omega_{\max} = 2\pi$ is double the Nyquist frequency, you see aliasing on the time axis in Figure 1.15. In other words, along the left edge, the frequency first increases with time until it reaches the Nyquist frequency and then, as frequency goes beyond Nyquist, the apparent frequency decreases. Something else happens on the space axis which is best seen along the bottom edge of the plot. There the frequency sweeps steadily from zero to Nyquist.

Thus, we see that the continuum is different from the sampled world. In a continuum, the function $f(t-px)$ at constant t looks like the same function at constant

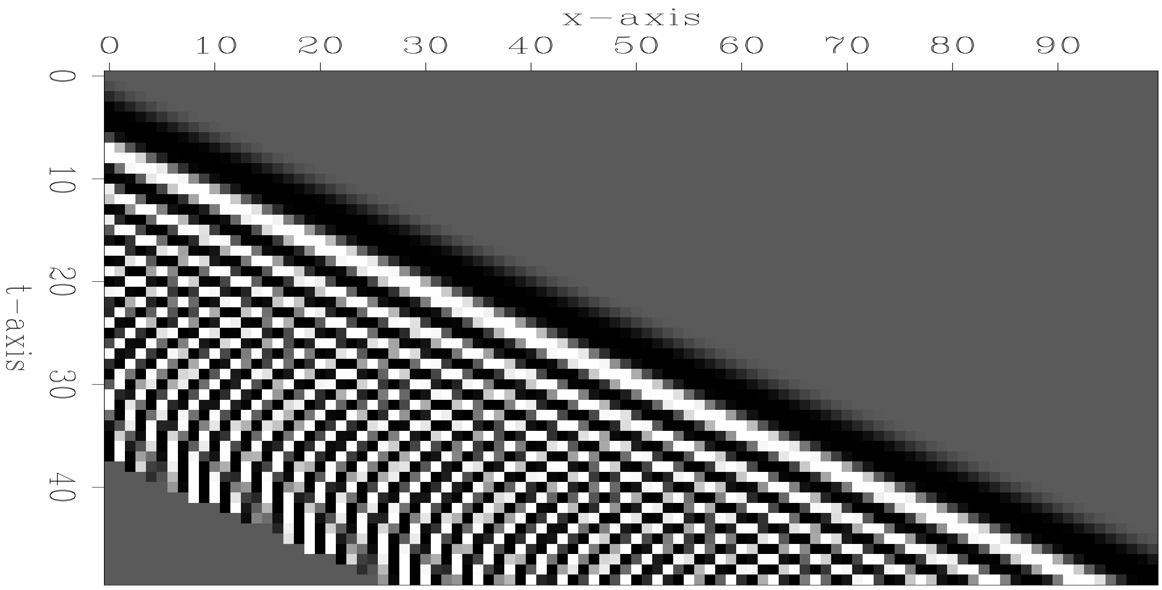


Figure 1.15: A coarsely sampled chirp function on a wavefront. View figure at a grazing angle from various directions. md-alias2d [NR]

x except for shifting, reversal, and stretching. In the sampled world we see different functions on each axis. The sampled world matches the continuum only at low frequencies.

Real life would be more like Figure 1.15 if we would interchange the time and space axis because it is easy to sample densely enough in time, or to high-cut filter the data before sampling it but it is hard to get enough observing locations in space. The result of insufficient density of stations is that seismic data is often aliased in space. Spatial aliasing is when a spatial frequency looks lower than it would if the data were more densely sampled in space.

1.6.2 Relation of missing data to inversion

We take *data space* to be a uniform mesh on which some values are given and some are missing. We rarely have missing values on a time axis, but commonly have missing values on a space axis, i.e., missing signals. Missing signals (traces) happen occasionally for miscellaneous reasons, and they happen systematically because of **alias**ing and **truncation**. The aliasing arises for economic reasons—saving instrumentation by running receivers far apart. Truncation arises at the ends of any survey, which, like any human activity, must be finite. Beyond the survey lies more hypothetical data. The traces we will find for the **missing data** are not as good as real observations, but they are closer to reality than supposing unmeasured data is zero valued. Making an image with a single application of an adjoint modeling operator amounts to assuming that data vanishes beyond its given locations. **Migration** is an example of an economically important process that makes this assumption. Dealing with missing data is a step beyond this. In **inversion**, restoring **missing data** reduces the need for arbitrary model filtering.

1.6.3 The world is filled with local plane waves.

In your ears now are sounds from various directions. From moment to moment the directions change. Momentarily, a single direction (or two) dominates. Your ears sample only two points in x-space. Earthquake data is a little better. Exploration data is much better and sometimes seems to satisfy the Nyquist requirement, especially when we forget that the world is 3-D.

We often characterize data from any region of (t, x)-space as "good" or "noisy" when we really mean it contains "few" or "many" plane-wave events in that region. For noisy regions there is no escaping the simple form of the Nyquist limitation. For good regions we may escape it. Real data typically contains both kinds of regions. Undersampled data with a broad distribution of plane waves is nearly hopeless. Undersampled data with a sparse distribution of plane waves is prospective. Consider data containing a spherical wave. The angular bandwidth in a plane-wave decomposition appears huge *until we restrict attention to a small region* of the data. (Actually a spherical wave contains very little information compared to an arbitrary wave field.) It can be very helpful in reducing the local angular bandwidth if we can deal effectively with tiny pieces of data. If we can deal with tiny pieces of data, then we can adapt to rapid spatial and temporal variations. This chapter will show such tiny windows of data.

1.7 INTERPOLATION BEYOND ALIASING

I have long marveled at the ability of humans to interpolate seismic data containing mixtures of dips where spatial frequencies exceed the Nyquist limits. These limits are hard limits on migration programs. Costly field-data-acquisition activities are designed with these limits in mind. I feared this human skill of going beyond the limit was deeply nonlinear and beyond reliable programming. Now, however, I have obtained results comparable in quality to those of S. **Spitz**, and I am doing so in a way that seems reliable—using two-stage, linear least squares.

Here we attack the problem of missing data arising in a regular pattern such as alternate traces being missing. This defeats earlier methods because every output requires missing data as inputs.

1.7.1 Traditional 2-D interpolation before aliasing

A traditional method of data interpolation on a regular mesh is a four-step procedure: (1) set zero values at the points to be interpolated; (2) **Fourier transform**; (3) set to zero the high frequencies; and (4) inverse transform. This is a fine method and is suitable for many applications in both one dimension and higher dimensions. However, this method fails to take advantage of our prior knowledge that seismic data has abundant fragments of plane waves which link an axis that is not aliased (time) to axes that might be (space).

1.7.2 Interlacing a filter

The filter below, however, can be designed despite alternate missing traces because it can destroy plane waves. If the plane wave should happen to pass halfway between the "d" and the "e", those two points could interpolate the halfway point, at least for well sampled temporal frequencies, and the time axis should always be well sampled.

$$\begin{array}{ccccccccc} a & \cdot & b & \cdot & c & \cdot & d & \cdot & e \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \end{array} \tag{1.24}$$

We could find the filter (1.24) using subroutine `pef2()` (page 11) if we set up the constraint array `afre(*,*)` appropriately. Then we could throw away alternate zeroed rows and columns to get the filter

$$\begin{array}{ccccc} a & b & c & d & e \\ \cdot & \cdot & 1 & \cdot & \cdot \end{array} \tag{1.25}$$

which could be given to subroutine `mis2()` (page 9) because both the filters (1.24) and (1.25) have the same dip characteristics.

Figure 1.16 shows three plane waves recorded on five channels and the interpolated data. Both the original data and the interpolated data can be described as "beyond

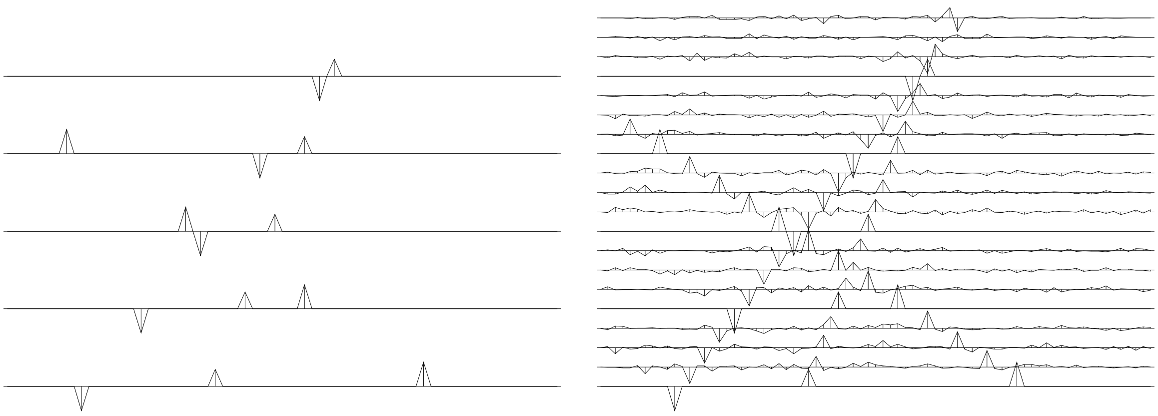


Figure 1.16: Left is five signals, each showing three arrivals. Using the data shown on the left (and no more), the signals have been interpolated. Three new traces appear between each given trace as shown on the right. md-lace3 [NR]

aliasing" because on the input data the signal shifts exceed the signal duration. The calculation requires only a few seconds of a "two-stage least-squares" method, where the first stage estimates an inverse **covariance matrix** of the known data, and the second uses it to estimate the missing traces. Actually, a **2-D prediction-error filter** is estimated, and the inverse covariance matrix, which amounts to the PE filter times its adjoint, is not needed explicitly. More examples of this type may be found in PVI.

Before the method of this section can be applied to field data for migration, remember that the data must be broken into many overlapping tiles of size about like those shown here and the results from each tile pieced together.

Figure 1.16 comes from PVI which introduces the clever method above. In this book we use more general methods that allow missing data in any location (but these more general methods often multiply by zero where multiplication could be avoided altogether). Next we see how the general methods of this book can overcome aliasing as in Figure 1.16.

1.8 MULTISCALE SELF-SIMILAR REGRESSIONS

Large objects often resemble small objects. Here we express this idea using *axis scaling* and we apply it to the basic theory of prediction-error filter (PEF) fitting and missing data estimation.

Equations (1.26) and (1.27) compute the same thing by two different methods, $\mathbf{r} = \mathbf{Ya}$ and $\mathbf{r} = \mathbf{Ay}$. Viewed as regressions minimizing $||\mathbf{r}||$ and along with suitable constraints, (1.26) leads to finding filters and **spectra** while (1.27) leads to finding **missing data**.

$$
\left[\begin{array}{c} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ \hline r_6 \\ r_7 \\ r_8 \\ r_9 \end{array}\right]
\quad = \quad
\left[\begin{array}{cc} y_2 & y_1 \\ y_3 & y_2 \\ y_4 & y_3 \\ y_5 & y_4 \\ y_6 & y_5 \\ \hline y_3 & y_1 \\ y_4 & y_2 \\ y_5 & y_3 \\ y_6 & y_4 \end{array}\right]
\left[\begin{array}{c} a_1 \\ a_2 \end{array}\right]
\qquad \text{or} \qquad
\left[\begin{array}{c} \mathbf{r}_1 \\ \mathbf{r}_2 \end{array}\right]
\;=\;
\left[\begin{array}{c} \mathbf{Y}_1 \\ \mathbf{Y}_2 \end{array}\right] \mathbf{a}
\tag{1.26}
$$

$$
\begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ \hline r_6 \\ r_7 \\ r_8 \\ r_9 \end{bmatrix}
\quad = \quad
\begin{bmatrix}
a_2 & a_1 & \cdot & \cdot & \cdot & \cdot \\
\cdot & a_2 & a_1 & \cdot & \cdot & \cdot \\
\cdot & \cdot & a_2 & a_1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & a_2 & a_1 & \cdot \\
\cdot & \cdot & \cdot & \cdot & a_2 & a_1 \\ \hline
a_2 & \cdot & a_1 & \cdot & \cdot & \cdot \\
\cdot & a_2 & \cdot & a_1 & \cdot & \cdot \\
\cdot & \cdot & a_2 & \cdot & a_1 & \cdot \\
\cdot & \cdot & \cdot & a_2 & \cdot & a_1
\end{bmatrix}
\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix}
\qquad \text{or} \qquad
\begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \end{bmatrix}
=
\begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{bmatrix} \mathbf{y}
\tag{1.27}
$$

A new concept embedded in (1.26) and (1.27) is that one filter can be applicable for different **stretching**s of the filter's time axis. One wonders, "Of all classes of filters, what subset remains appropriate for stretchings of the axes?"

1.8.1 Examples of scale invariant filtering

For all functions with vanishing gradient, we notice the gradient vanishes whether it is represented as $(1, -1)/\Delta x$ or as $(1, 0, -1)/2\Delta x$. Likewise for the Laplacian, in one dimension or more. Likewise for the wave equation, so long as there is no viscosity and so long as the time axis and space axes are stretched by the same amount. The notion of "dip filter" seems to have no formal definition but the idea that the spectrum should depend mainly on slope in Fourier space implies a filter that is scale invariant. I expect the most fruitful applications to be with **dip filter**s.

Resonance or **viscosity** or damping easily spoils scale invariance, however. The resonant frequency of a filter shifts if we stretch the time axis. The difference equations

$$
y_t - \alpha y_{t-1} \quad = \quad 0 \tag{1.28}
$$

$$
y_t - \alpha^2 y_{t-2} \quad = \quad 0 \tag{1.29}
$$

both have the same solution $y_t = y_0 \alpha^{-t}$. One difference equation has the filter $(1, -\alpha)$ while the other has the filter $(1, 0, -\alpha^2)$ and α is not equal to α^2. Although these operators differ, when $\alpha \approx 1$ they might provide the same general utility, say as a roughening operator in a regression.

Another aspect to scale invariance work is "parasitic" solutions which exist but are not desired. For example, another solution to $y_t - y_{t-2} = 0$ is the one that oscillates at the Nyquist frequency.

(Viscosity does not necessarily introduce an inherent length and thereby spoil scale invariance. Approximate frequency independence of sound absorption per wavelength typical in real rocks is a consequence of physical inhomogeneity at all scales. See for example **Kjartansson**'s **constant Q** viscosity described in **IEI**. Kjartansson teaches that the decaying solutions $t^{-\gamma}$ are scale invariant. There is no "decay time" for the function $t^{-\gamma}$. Differential equations of finite order and difference equations of finite order cannot produce $t^{-\gamma}$ damping yet we know such damping is important in observations. It is easy to manufacture $t^{-\gamma}$ damping in Fourier space using $\exp[(-i\omega)^{\gamma+1}]$. Presumably difference equations can make reasonable approximations over a reasonable frequency range.)

1.8.2 Scale-invariance introduces more regression equations

The regressions (1.26) and (1.27) have about double the usual number of regression equations. Scale-invariance *introduces extra regression equations.* If the range of scale-invariance is wide, there will be more equations. Now we begin to see the view to the end of the tunnel:

1. Refining a computational mesh improves accuracy.
2. Refining a data mesh makes empty bins.
3. Empty bins spoil analysis.
4. If there are not too many empty bins we can find a PEF.
5. With a PEF we can fill the empty bins.
6. To get the PEF and to fill bins we need enough regression equations.
7. Scale-invariance introduces more regression equations.

An example of these concepts are shown in Figure 1.17.

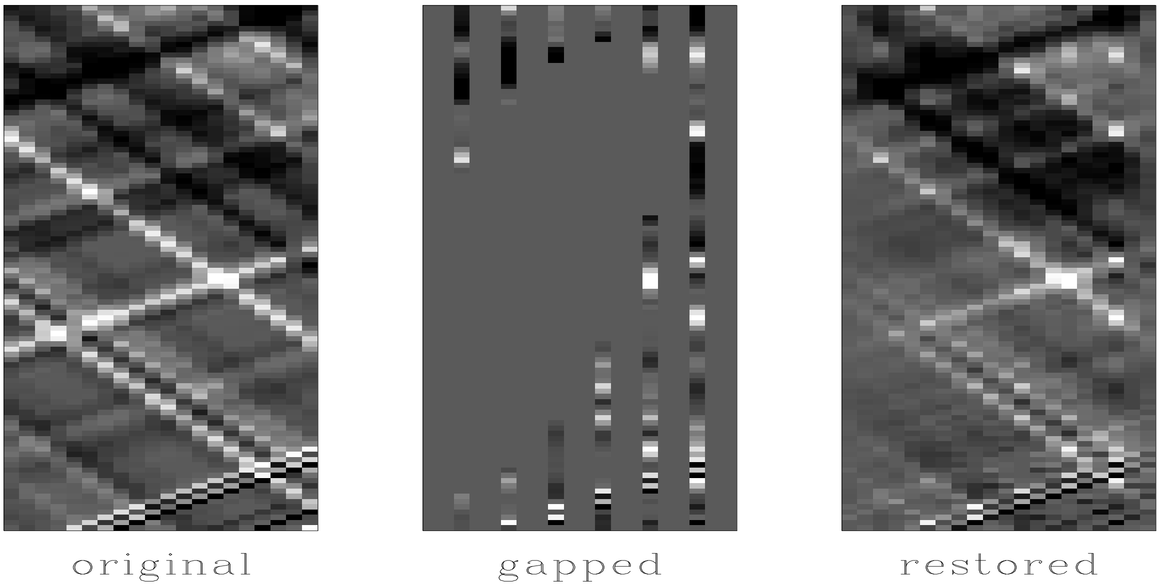


Figure 1.17: Overcomeing aliasing with multiscale regression. md-mshole [NR]

Additionally, when we have a PEF, often we still cannot find missing data because conjugate-gradient iterations do not converge fast enough (to fill large holes). Multi scale convolutions should converge quicker because they are like mesh refinement which is quick. An example of these concepts are shown in Figure 1.18.

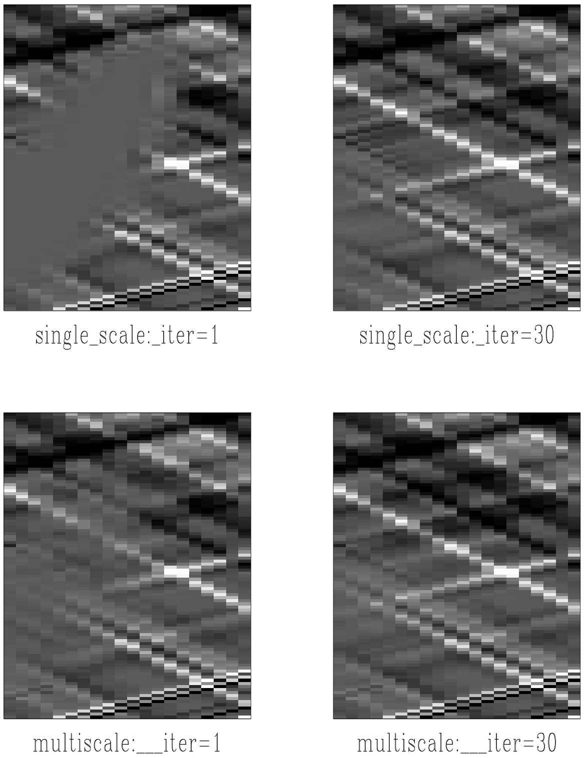


Figure 1.18: Large holes are filled faster with multiscale operators. md-msiter [NR]

1.8.3 Coding the multiscale filter operator

Many applications could need slightly different equations, namely

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \quad = \quad \begin{bmatrix} w_1\mathbf{I} & \mathbf{0} \\ \mathbf{0} & w_2\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{Y}_1 \\ \mathbf{Y}_2 \end{bmatrix} \mathbf{a} \; + \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \end{bmatrix} \tag{1.30}$$

and

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \quad = \quad \begin{bmatrix} w_1\mathbf{I} & \mathbf{0} \\ \mathbf{0} & w_2\mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{bmatrix} \mathbf{y} \; + \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \end{bmatrix} \tag{1.31}$$

where w_1 and w_2 are weighting scale factors. This added complication can be easily overcome by putting the scaling factor in the central loops in the program, which I did. It would be more efficient, however, to take the scale factor out of the inner loop and apply it to the residual directly. I didn't do this because it would clutter the program and an optimizing compiler might take care of it anyway.

Equation (1.26) shows an example where the first output signal is the ordinary one and the second output signal used a filter interlaced with zeros. We prepare subroutines that allow for more output signals, each with its own filter interlace parameter given in the table `jump(ns)`. Each entry in the jump table corresponds to a particular filter axis scaling. The number of output signals is `ns` and the number of zeros interlaced between filter points for the j-th signal is `jump(j)-1`. Also, there is the weighting scale factor for each scaling, `wt(ns)`.

First we examine code for estimating a prediction-error filter that is applicable at many scales. PEF estimation is generally done with *internal* filtering where the output is shorter than the input. For viewing (and for coding in Fortran) it is convenient for the filter outputs of different scales to be loosely packed (zeros between) so output space is conformable frames. The coding is much like that in `icaf2()` (page 9), the new ingredient being the loop over scales and the jumping along axes. (Till now I am ignoring the `lag1,lag2` parameters that allow positioning the smaller output in its larger input frame.)

```
# Multi-Scale Internal Convolution, Adjoint is the Filter.  2-dimensional.
#
subroutine msicaf2( adj,add,jump, wt,ns, yy,ny1,ny2, aa,na1,na2, rr)
integer     is,     adj,add,jump(ns),ns,    ny1,ny2,    na1,na2
integer                                      y1, y2,     a1, a2, r1,r2, nr1,nr2
real                              wt(ns),yy(ny1,ny2),aa(na1,na2),rr(ny1,ny2,ns)
call adjnull(       adj,add,                          aa,na1*na2, rr,ny1*ny2*ns)
do is = 1, ns {
        nr1= ny1 - (na1-1)* jump(is);   if( nr1<1) call erexit('msic2: 1-axis')
        nr2= ny2 - (na2-1)* jump(is);   if( nr2<1) call erexit('msic2: 2-axis')
        do a2=1,na2 {  do r2=1,nr2 {  y2 = r2 - (a2-na2) * jump(is)
        do a1=1,na1 {  do r1=1,nr1 {  y1 = r1 - (a1-na1) * jump(is)
            if( adj == 0 )
                rr(r1,r2,is) = rr(r1,r2,is) + yy(y1,y2) * aa(a1,a2   ) * wt(is)
            else
                aa(a1,a2   ) = aa(a1,a2   ) + yy(y1,y2) * rr(r1,r2,is) * wt(is)
            }} }}
        }
return; end
```

The multiscale prediction-error filter finding subroutine is nearly identical to the usual missing data subroutine `pef2()` (page 11). (That routine cleverly ignores missing data while estimating a PEF.) We can easily extend `pef2()` to multiscale filters by replacing its call to the ordinary 2-D filter subroutine by a call to `msicaf2()` (page 37). Of course it also needs to know that multiscale filtering has a larger output space (more panels) than single scale filtering does.

```
# Find multiscale prediction-error filter avoiding missing data.
#
subroutine mspef2(jump,   wt,ns, data,n1,n2,  afre,       aa,a1,a2)
integer             jump(ns),  ns,      n1,n2,                a1,a2,  nr
integer    a12, am, n12,                i1,i2,  iter, niter
real                       wt(ns),data(n1*n2), afre(a1,a2),aa(a1,a2)
temporary real   da(a1,a2), dr(n1*n2*ns),rr(n1*n2*ns),   ww(n1*n2*ns)
temporary real   sa(a1,a2), sr(n1*n2*ns),wr(n1*n2*ns), dfre(n1*n2)
am=1+a1/2;      a12= a1*a2; nr= n1*n2*ns;               n12= n1*n2;
call null( rr, nr);   call null( aa, a12);   aa(am,1) =1.;
call setfree( am,1,1, afre, a1,a2,1, niter)
do i1=1,n12 { if( data(i1)==0.0)  dfre(i1) = 1.
              else                dfre(i1) = 0. }                 afre(am,1)=1.
call         msicaf2( 0,0, jump,wt,ns, dfre,n1,n2,afre,a1,a2, ww);afre(am,1)=0.
do i1= 1, nr
        if( ww(i1) > 0.0) { ww(i1) = 0.0 }            # free data hits filter.
```

```
        else              { ww(i1) = 1.0 }
call        msicaf2( 0,0, jump,wt,ns, data,n1,n2, aa,a1,a2, rr)
call        diag(    0,0,        ww, nr,                   rr,    wr )
do iter= 0, 3*niter {
        call diag(    1,0,        ww, nr,                   dr,    wr )
        call msicaf2( 1,0, jump,wt,ns, data,n1,n2, da,a1,a2, dr)
        do i2= 1, a2
        do i1= 1, a1 {  da(i1,i2) = da(i1,i2) * afre(i1,i2)}
        call msicaf2( 0,0, jump,wt,ns, data,n1,n2, da,a1,a2, dr)
        call diag(    0,0,        ww, nr,                   dr,    dr )
        call cgplus( iter,  a12 , aa,da,sa,
                            nr  , wr,dr,sr )
        }
call        msicaf2( 0,0, jump,wt,ns, data,n1,n2,  aa,a1,a2, rr)
return; end
```

Likewise, I coded up the operator in (1.27) but with transient boundary conditions since those boundaries seem better for missing data estimation. With transient ends, the output size `nr` is harder to estimate since each output has a size that grows with the size of its expanded filter. Here I packed the outputs with no space between them, but I wonder if loose packing for the worst case might not be better. The 2-D multiscale convolution subroutine with transient boundaries, with tight-packed outputs, and with adjoint being the input is `mstcai2()` (page 38).

```
# Multi-Scale Transient Convolution, Adjoint is the Input.  2-dimensional.
#
subroutine mstcai2( adj,add, jump,   wt,ns, aa,a1,a2,  yy,y1,y2,  rr,nr)
integer    is, rs,  adj,add, jump(ns),  ns,    a1,a2,     y1,y2,     nr
real                                 wt(ns),aa(a1,a2), yy(y1,y2), rr(nr)
integer ir,ir1,ir2,r1,r2,    ja1,ja2,         ia1,ia2,   iy1,iy2
call adjnull(       adj,add,                           yy,y1*y2,  rr,nr)
rs = 0
do is= 1, ns {
        r1= y1 + (1+(a1-1)*jump(is)) - 1
        r2= y2 + (1+(a2-1)*jump(is)) - 1
        do ia2=1,a2 { ja2=1+(ia2-1)*jump(is);  do iy2=1,y2 { ir2= iy2+ja2-1
        do ia1=1,a1 { ja1=1+(ia1-1)*jump(is);  do iy1=1,y1 { ir1= iy1+ja1-1
            ir = rs + ir1 + r1*(ir2-1)
            if( adj == 0 )
                rr(ir     ) = rr(ir     ) + yy(iy1,iy2) * aa(ia1,ia2) * wt(is)
            else
                yy(iy1,iy2) = yy(iy1,iy2) + rr(ir     ) * aa(ia1,ia2) * wt(is)
            }} }}
        rs = rs + r1 * r2
        }
return; end
#
subroutine mstc2alloc( jump,ns,  a1,a2, y1,y2, nr )
integer         is,ns, jump(ns), a1,a2, y1,y2, nr , rs
rs = 0
do is= 1, ns
        rs = rs +  (y1 + (1+(a1-1)*jump(is)) - 1 ) * _
                   (y2 + (1+(a2-1)*jump(is)) - 1 )
nr = rs
return; end
```

The multiscale missing data subroutine `msmis2()` (page 38) is just like the usual missing data subroutine `mis2()` (page 9) the filtering is done with a multiscale subroutine and the size of the residual must be precomputed externally.

```
# Given PEF=aa() and data=xx with zeros missing,  find xx = known + missing
#
```

```
subroutine msmis2( jump,wt,ns, xfre, xx, x1,x2, niter, aa,a1,a2, nr)
integer                    ns,            x1,x2, niter,    a1,a2, nr
integer                                   i1,i2,  iter
real  xfre(x1,x2), jump(ns),wt(ns),  xx( x1,x2),       aa(a1,a2)
temporary real                                               rr( nr)
temporary real                       dx( x1,x2),             dr( nr)
temporary real                       sx( x1,x2),             sr( nr)
        call mstcai2( 0,0, jump,wt,ns, aa,a1,a2,  xx,x1,x2,  rr,nr)
do iter= 0, niter {
        call mstcai2( 1,0, jump,wt,ns, aa,a1,a2,  dx,x1,x2,  rr,nr)
        do i1= 1, x1 {
        do i2= 1, x2 {  dx(i1,i2) = dx(i1,i2) * xfre(i1,i2)
                        }}
        call mstcai2( 0,0, jump,wt,ns, aa,a1,a2,  dx,x1,x2,  dr,nr)
        call cgplus( iter,  x1*x2, xx,dx,sx,
                            nr   , rr,dr,sr )
        }
return; end
```

REFERENCES

Canales, L.L., 1984, Random noise reduction: 54th Ann. Internat. Mtg., Soc. Explor. Geophys., Expanded Abstracts, 525-527.

Rothman, D., 1985, Nonlinear inversion, statistical mechanics, and residual statics estimation: Geophysics, **50**, 2784-2798

Spitz, S., 1991, Seismic trace interpolation in the F-X domain: Geophysics, **56**, 785-794.