Chapter 1

Spatial aliasing and scale invariance

Landforms are not especially predictable. Therefore, crude PEF approximations are often satisfactory. Wavefields are another matter. Consider the "shape" of the acoustic wavefronts at this moment in the room you are in. The acoustic wavefield has statistical order in many senses. If the 3-D volume is filled with waves emitted from a few point sources, then (with some simplifications) what could be a volume of information is actually a few 1-D signals. When we work with wavefronts we can hope for more dramatic, even astounding, results from estimating properly.

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

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. Where regions are noisy, there is no escaping the simple form of the Nyquist limitation. Where regions are good 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 offer us the opportunity to resample without aliasing. 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 shows such tiny windows of data.

1.1 INTERPOLATION BEYOND 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 that link an axis that is not aliased (time) to axes that often are (space).

1.1.1 Interlacing a filter

The filter below can be designed despite alternate missing traces. This filter destroys 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. For example, $d = e = -.5$ would almost destroy the plane wave and it is an aliased planewave for its higher frequencies.

$$\begin{array}{ccccccccc} a & \cdot & b & \cdot & c & \cdot & d & \cdot & e \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \end{array} \tag{1.1}$$

We could use module `pef` on page ?? to find the filter (1.1), if we set up the lag table `lag` appropriately. Then we could throw away alternate zeroed rows and columns (rescale the lag) to get the filter

$$\begin{array}{ccccc} a & b & c & d & e \\ \cdot & \cdot & 1 & \cdot & \cdot \end{array} \tag{1.2}$$

which could be used with subroutine `mis1()` on page ??, to find the interleaved data because both the filters (1.1) and (1.2) have the same dip characteristics.

Figure 1.1 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 **alias**ing," because

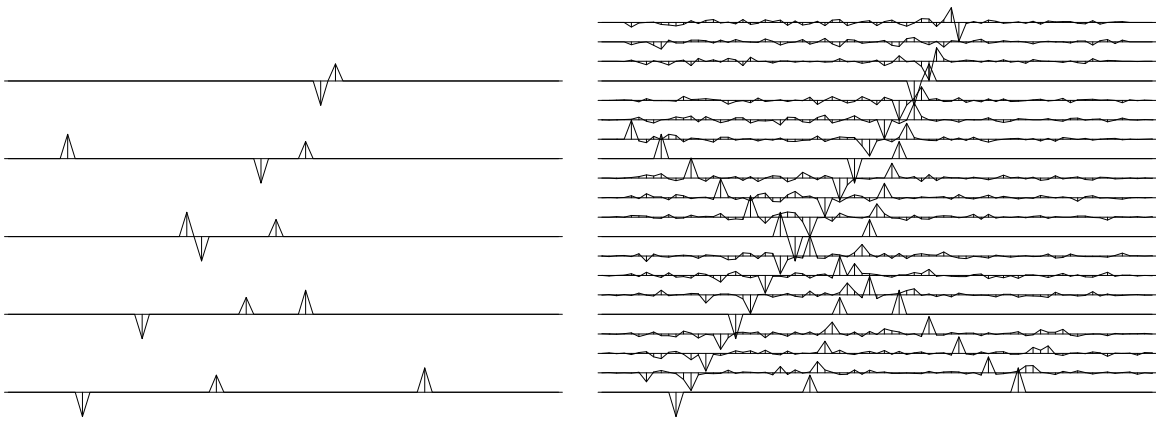

Figure 1.1: Left is five signals, each showing three arrivals. With the data shown on the left (and no more), the signals have been interpolated. Three new traces appear between each given trace, as shown on the right. lal-lace390 [ER]

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, in which the first stage estimates a PEF (inverse spectrum) of the known data, and the second uses the PEF to estimate the missing traces. Figure 1.1 comes from PVI which introduces the clever method described above. We will review how that was done and examine the F90 codes that generalize it to N-dimensions. Then we'll go on to more general methods that allow missing data in any location. Before the methods of this section are applied to field data for migration, data must be broken into many overlapping tiles of size about like those shown here and the results from each tile pieced together. That is described later in chapter 9.

A PEF is like a differential equation. The more plane-wave solutions you expect, the more lags you need on the data. Returning to Figure 1.1, the filter must cover four traces (or more) to enable it to predict three plane waves. In this case, `na=(9,4)`. As usual, the spike on the 2-D PEF is at `center=(5,1)`. We see the filter is expanded by a factor of `jump=4`. The data size is `nd=(75,5)` and `gap=0`. Before looking at the code `lace` on this page for estimating the PEF, it might be helpful to recall the basic utilities `line2cart` and `cart2line` on page ?? for conversion between a multidimensional space and the helix filter lag. We need to sweep across the whole filter and "stretch" its lags on the 1-axis. We do not need to stretch its lags on the 2-axis because the data has not yet been interlaced by zero traces.

```
module lace {                                        # find PEF on interlaced data
  use createhelixmod
  use bound
  use pef
  use cartesian
contains
  function lace_pef( dd, jump, nd, center, gap, na)  result( aa) {
    type( filter)                       :: aa
    integer,                intent( in) :: jump
    integer, dimension(:), intent( in) :: nd, center, gap, na
    real,    dimension(:), pointer     :: dd          # input data
    integer, dimension(:), pointer     :: savelags    # holding place
    integer, dimension( size( nd))     :: ii
    integer                            :: ih, nh, lag0, lag
    aa = createhelix( nd, center, gap, na);  nh = size( aa%lag)
    savelags => aa%lag;   allocate( aa%lag( nh))      # prepare interlaced helix
    call cart2line(  na, center, lag0)
    do ih = 1, nh {                                     # Sweep thru the filter.
       call line2cart( na, ih+lag0, ii)
       ii = ii - center;            ii(1) = ii(1)*jump  # Interlace on 1-axis.
       call cart2line( nd,  ii+1, lag)
       aa%lag( ih) = lag - 1
       }
    call boundn( nd, nd, (/ na(1)*jump, na(2:) /), aa)  # Define  aa.mis
    call find_pef( dd, aa, nh*2)                        # Estimate aa coefs
    deallocate( aa%lag);   aa%lag => savelags           # Restore filter lags
  }
}
```

The line `ii(1)=ii(1)*jump` means we interlace the 1-axis but not the 2-axis because the data has not yet been interlaced with zero traces. For a 3-D filter `aa(na1,na2,na3)`, the

somewhat obtuse expression `(/na(1)*jump, na(2:)/)` is a three component vector containing `(na1*jump, na2, na3)`.

After the PEF has been found, we can get missing data in the usual way with with module `mis2` on page ??.

1.2 MULTISCALE, SELF-SIMILAR FITTING

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

Equations (1.3) and (1.4) compute the same thing by two different methods, $\mathbf{r} = \mathbf{Ya}$ and $\mathbf{r} = \mathbf{Ay}$. When it is viewed as fitting goals minimizing $||\mathbf{r}||$ and used along with suitable constraints, (1.3) leads to finding filters and **spectra**, while (1.4) leads to finding **missing data**.

$$\begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ \hline r_6 \\ r_7 \\ r_8 \\ r_9 \end{bmatrix} = \begin{bmatrix} y_2 & y_1 \\ y_3 & y_2 \\ y_4 & y_3 \\ y_5 & y_4 \\ y_6 & y_5 \\ \hline y_3 & y_1 \\ y_4 & y_2 \\ y_5 & y_3 \\ y_6 & y_4 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \qquad \text{or} \qquad \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{Y}_1 \\ \mathbf{Y}_2 \end{bmatrix} \mathbf{a} \tag{1.3}$$

$$\begin{bmatrix} r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ \hline r_6 \\ r_7 \\ r_8 \\ r_9 \end{bmatrix} = \begin{bmatrix} a_2 & a_1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & a_2 & a_1 & \cdot & \cdot & \cdot \\ \cdot & \cdot & a_2 & a_1 & \cdot & \cdot \\ \cdot & \cdot & \cdot & a_2 & a_1 & \cdot \\ \cdot & \cdot & \cdot & \cdot & a_2 & a_1 \\ \hline a_2 & \cdot & a_1 & \cdot & \cdot & \cdot \\ \cdot & a_2 & \cdot & a_1 & \cdot & \cdot \\ \cdot & \cdot & a_2 & \cdot & a_1 & \cdot \\ \cdot & \cdot & \cdot & a_2 & \cdot & a_1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \qquad \text{or} \qquad \begin{bmatrix} \mathbf{r}_1 \\ \mathbf{r}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{A}_1 \\ \mathbf{A}_2 \end{bmatrix} \mathbf{y} \tag{1.4}$$

A new concept embedded in (1.3) and (1.4) 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?"

1.2.1 Examples of scale-invariant filtering

When we consider all functions with vanishing gradient, we notice that 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, as long as there is no viscosity and as 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. The resonant frequency of a filter shifts if we stretch the time axis. The difference equations

$$y_t - \alpha y_{t-1} = 0 \tag{1.5}$$
$$y_t - \alpha^2 y_{t-2} = 0 \tag{1.6}$$

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 fitting goal.

Another aspect to scale-invariance work is the presence of "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. The 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 that such damping is important in observations. It is easy to manufacture $t^{-\gamma}$ damping in Fourier space; $\exp[(-i\omega)^{\gamma+1}]$ is used. Presumably, difference equations can make reasonable approximations over a reasonable frequency range.)

1.2.2 Scale-invariance introduces more fitting equations

The fitting goals (1.3) and (1.4) have about double the usual number of fitting equations. Scale-invariance *introduces extra equations*. If the range of scale-invariance is wide, there will be more equations. Now we begin to see the big picture.

1. Refining a model mesh improves accuracy.
2. Refining a model 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 equations.
7. Scale-invariance introduces more equations.

An example of these concepts is shown in Figure 1.2. Additionally, when we have a PEF, often

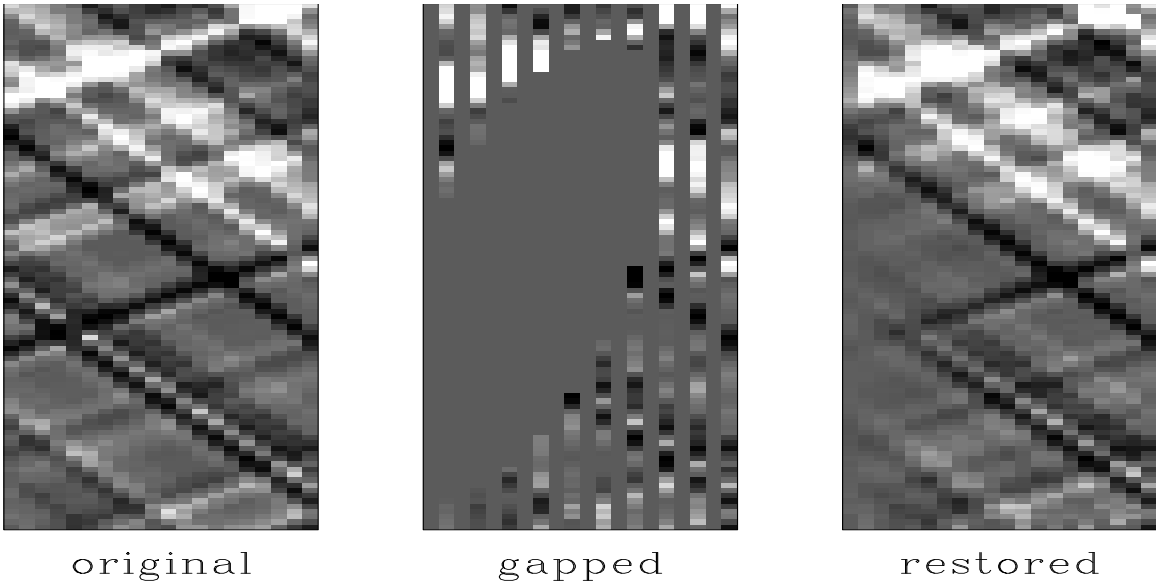

Figure 1.2: Overcoming aliasing with multiscale fitting. lal-mshole90 [ER]

we still cannot find missing data because conjugate-direction iterations do not converge fast enough (to fill large holes). Multiscale convolutions should converge quicker because they are like mesh-refinement, which is quick. An example of these concepts is shown in Figure 1.3.

1.2.3 Coding the multiscale filter operator

Equation (1.3) 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 scaling of a filter axis. 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`.

The multiscale helix filter is defined in module `mshelix` on the current page, analogous to the single-scale module `helix` on page ??. A new function `onescale` extracts our usual helix filter of one particular scale from the multiscale filter.

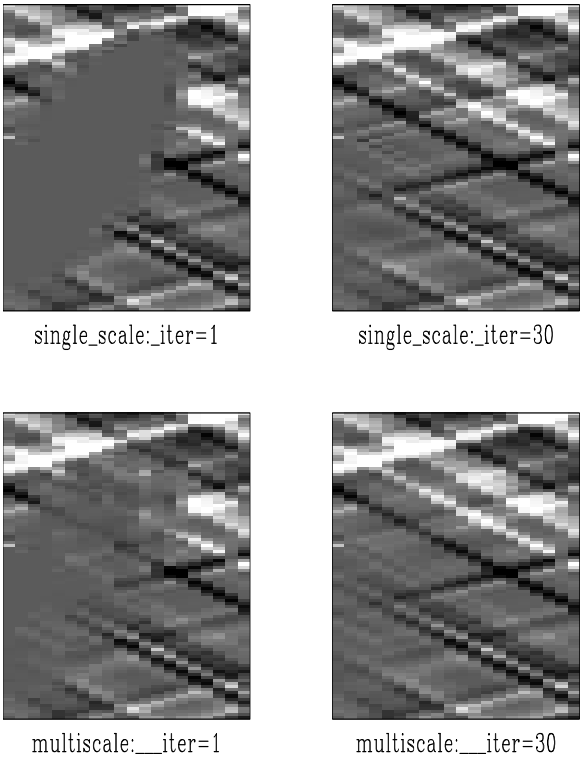

Figure 1.3: Large holes are filled faster with multiscale operators. lal-msiter90 [ER]

```
module mshelix {                                  # multiscale helix filter type
  use helix
  type msfilter {
    real,    dimension( :),    pointer :: flt   # (nh) filter coefficients
    integer, dimension( :, :), pointer :: lag   # (nh,ns) filter (lags,scales)
    logical, dimension( :, :), pointer :: mis   # (nd,ns) boundary conditions
  }
contains
  subroutine msallocate( msaa, nh, ns) {
    type( msfilter)   :: msaa
    integer           :: nh, ns
    allocate( msaa%flt( nh), msaa%lag( nh, ns))
    msaa%flt = 0.;  nullify( msaa%mis)
    }
  subroutine msdeallocate( msaa) {
    type( msfilter) :: msaa
    deallocate( msaa%flt, msaa%lag)
    if( associated( msaa%mis)) deallocate( msaa%mis)
    }
  subroutine onescale( i, msaa, aa) {      # Extract single-scale filter.
    integer, intent (in) :: i
    type( filter)        :: aa
    type( msfilter)      :: msaa
    aa%flt => msaa%flt
    aa%lag => msaa%lag( :, i)
    if( associated( msaa%mis))
        aa%mis => msaa%mis( :, i)
    else
        nullify( aa%mis)
```

```
    }
}
```

We create a multscale helix with module `createmshelixmod` on the current page. An expanded scale helix filter is like an ordinary helix filter except that the lags are scaled according to a `jump`.

```
module createmshelixmod {         # Create multiscale helix filter lags and mis
use mshelix
use createhelixmod
use bound
contains
  function createmshelix( nd, center, gap, jump, na)  result( msaa) {
    type( msfilter)                   :: msaa    # needed by mshelicon.
    integer, dimension(:), intent(in) :: nd, na  # data and filter axes
    integer, dimension(:), intent(in) :: center  # normally (na1/2,na2/2,...,1)
    integer, dimension(:), intent(in) :: gap     # normally ( 0,   0,  0,...,0)
    integer, dimension(:), intent(in) :: jump    # jump(ns) stretch scales
    type( filter)                     :: aa
    integer                           :: is, ns, nh, n123
    aa = createhelix( nd, center, gap, na)
    ns = size( jump);  nh = size( aa%lag);  n123 = product( nd)
    call msallocate( msaa, nh, ns)
    do is = 1, ns
        msaa%lag(:,is) = aa%lag(:)*jump(is)      # set lags for expanded scale
    call deallocatehelix( aa)
    allocate( msaa%mis( n123, ns))
    do is = 1, ns {                              # for all scales
     call onescale( is, msaa, aa);  nullify( aa%mis)    # extract a filter
     call boundn( nd, nd, na*jump(is), aa)              # set up its boundaries
     msaa%mis( :, is) = aa%mis;  deallocate( aa%mis)    # save them
    }
  }
}
```

First we examine code for estimating a prediction-error filter that is applicable at many scales. We simply invoke the usual filter operator `hconest` on page ?? for each scale.

```
module mshconest {      # multi-scale helix convolution, adjoint is the filter.
  use mshelix
  use hconest
  use helix
  integer, private                    :: nx, ns
  real,    dimension(:),     pointer :: x
  type( msfilter)                     :: msaa
#% _init( x, msaa)
   nx = size( x);   ns = size( msaa%lag, 2)
#% _lop( a(:), y(nx,ns))
    integer  :: is, stat1
    type (filter) :: aa
    do is = 1, ns {
        call onescale (is, msaa, aa)
        call hconest_init( x, aa)
```

```
        stat1 = hconest_lop( adj, .true., a, y(:,is))
    }
}
```

The **multiscale prediction-error filter** finding subroutine is nearly identical to the usual subroutine `find_pef()` on page ??. (That routine cleverly ignores missing data while estimating a PEF.) To easily extend `pef` to multiscale filters we replace its call to the ordinary helix filter module `hconest` on page ?? by a call to `mshconest`.

```
module mspef {         # Find multi-scale prediction-error filter (helix magic)
  use mshconest
  use cgstep_mod
  use solver_mod
contains
  subroutine find_pef( yy, aa, niter) {
    integer,             intent( in)             :: niter
    real, dimension( :), pointer                 :: yy
    type( msfilter)                              :: aa
    integer                                      :: is
    real, dimension( size( yy), size( aa%lag, 2)) :: dd
    do is = 1, size( dd, 2)
       dd( :,is) = -yy
    call mshconest_init( yy, aa)
    call solver( mshconest_lop, cgstep, aa%flt, pack( dd, .true.),
                 niter, x0= aa%flt)
    call cgstep_close()
    }
}
```

The purpose of `pack(dd,.true.)` is to produce the one-dimensional array expected by `solver()` on page ??.

Similar code applies to the operator in (1.4) which is needed for missing data problems. This is like `mshconest` on the facing page except the adjoint is not the filter but the input.

```
module mshelicon {                                  # Multi-scale convolution
  use mshelix
  use helicon
  integer           :: nx, ns
  type( msfilter)   :: msaa
#%  _init (nx, ns, msaa)
#%  _lop ( xx( nx),  yy( nx, ns))
  integer :: is, stat1
  type (filter) :: aa
  do is = 1, ns {
     call onescale( is, msaa, aa)
     call helicon_init( aa)
     stat1 = helicon_lop( adj, .true., xx, yy(:,is))
     }
}
```

The multiscale missing-data module `msmis2` is just like the usual missing-data module `mis2` on page ?? except that the filtering is done with the multiscale filter `mshelicon` on this page.

```
module msmis2 {                       # multi-scale missing data interpolation
  use mshelicon
  use cgstep_mod
  use solver_mod
contains
  subroutine mis1( niter, nx, ns, xx, aa, known) {
    integer,                intent( in)     :: niter, nx, ns
    logical, dimension( :), intent( in)     :: known
    type( msfilter),        intent( in)     :: aa
    real,    dimension( :), intent( in out) :: xx
    real,    dimension( nx*ns)              :: dd
    dd = 0.
    call mshelicon_init( nx,ns, aa)
    call solver( mshelicon_lop, cgstep, niter= niter, x = xx, dat = dd,
                  known = known, x0 = xx)
    call cgstep_close()
  }
}
```

1.3 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.

Index