Chapter 9

Nonstationarity: patching

There are many reasons for cutting data planes or image planes into overlapping pieces (patches), operating on the pieces, and then putting them back together again, as depicted in Figure 9.1. The earth's dip varies with lateral location and depth. The dip spectrum and spatial spectrum thus also varies. The dip itself is the essence of almost all earth mapping, and its spectrum plays an important role in the estimation any earth properties. In statistical estimation theory, the word to describe changing statistical properties is "**nonstationary**".

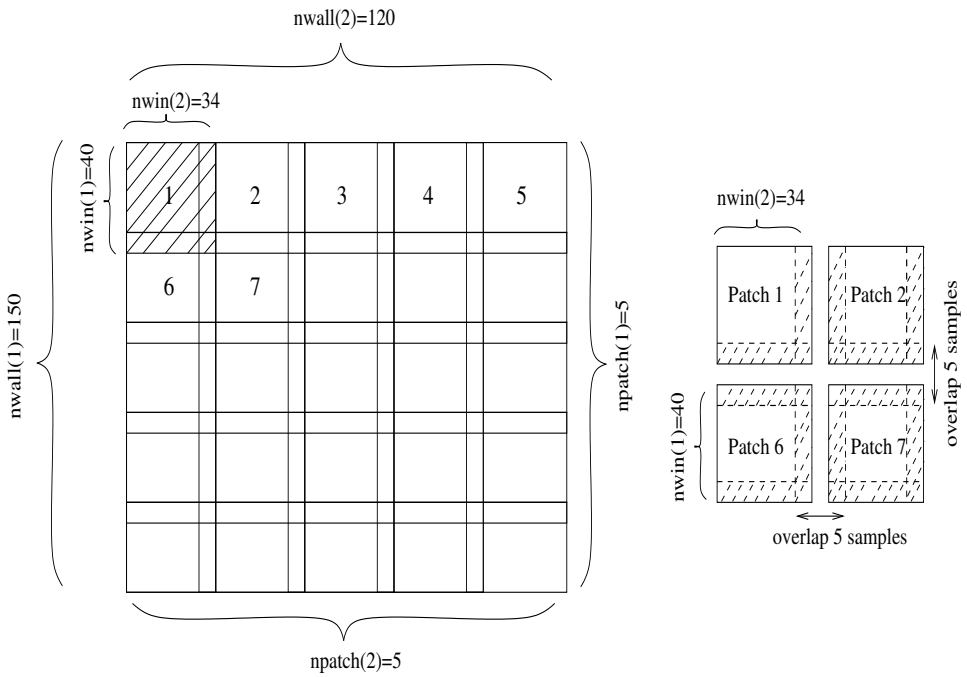

Figure 9.1: Decomposing a wall of information into windows (also called patches). Left is an example of a 2-D space input to module `patch`. Right shows a close-up of the output (top left corner). pch-antoine [NR]

We begin this chapter with basic patching concepts along with supporting utility code.

The language of this chapter, *patch, overlap, window, wall,* is two-dimensional, but it may as well be three-dimensional, *cube, subcube, brick,* or one-dimensional, *line, interval.* We sometimes use the language of windows on a wall. But since we usually want to have overlapping windows, better imagery would be to say we assemble a quilt from patches.

The codes are designed to work in any number of dimensions. After developing the infrastructure, we examine some two-dimensional, time- and space-variable applications: adaptive steep-dip rejection, noise reduction by prediction, and segregation of signals and noises.

9.1 PATCHING TECHNOLOGY

A plane of information, either data or an image, say `wall(nwall1, nwall2)`, will be divided up into an array of overlapping **window**s each window of size `(nwind1,nwind2)`. To choose the number of windows, you specify `(npatch1,npatch2)`. Overlap on the 2-axis is measured by the fraction `(nwind2*npatch2)/nwall2`. We turn to the language of F90 which allows us to discuss N-dimensional hypercubes almost as easily as two-dimensional spaces. We define an N-dimensional volume (like the wall) with the vector `nwall= (nwall1, nwall2, ...)`. We define subvolume size (like a 2-D window) with the vector `nwind=(nwind1, nwind2, ...)`. The number of subvolumes on each axis is `npatch=(npatch1, npatch2, ...)`. The operator `patch` on this page simply grabs one patch from the wall, or when used in adjoint form, it puts the patch back on the wall. The number of patches on the wall is `product(npatch)`. Getting and putting all the patches is shown later in module `patching` on page 215.

The i-th patch is denoted by the scalar counter `ipatch`. Typical patch extraction begins by taking `ipatch`, a fortran linear index, and converting it to a multidimensional subscript `jj` each component of which is less than `npatch`. The patches cover all edges and corners of the given data plane (actually the hypervolume) even where `nwall/npatch` is not an integer, even for axes whose length is not an integer number of the patch length. Where there are noninteger ratios, the spacing of patches is slightly uneven, but we'll see later that it is easy to reassemble seamlessly the full plane from the patches, so the unevenness does not matter. You might wish to review the utilities `line2cart` and `cart2line` on page 112 which convert between multidimensional array subscripts and the linear memory subscript before looking at the patch extraction-putback code:

```
module patch {                      # N-dimensional patch extract and putback
use cartesian
integer, private                 :: ipatch         # count till product(npatch)
integer, dimension (:), pointer :: npatch,  nwind,  nwall
               # overlap where    npatch * nwind > nwall
#% _init( npatch, nwall, nwind)
ipatch = 1
#% _lop ( wall, wind)
  integer, dimension( size( npatch)) :: ii, jj       # Ndim=size(npatch)
  integer                            :: i, j, shift
  call line2cart(  npatch, ipatch, jj )  # (j1,j2) = (1,1) to (npatch1,npatch2)
             # round jj to shift end of last patch flush against the far wall.
```

```
  where( npatch==1) { jj = 1   }
  elsewhere         { jj = 1.5 +( nwall - nwind)*( jj - 1.)/( npatch - 1.)}
  call cart2line( nwall,   jj, shift)    # jj is where the patch begins.
  shift -= 1                             #
  do i  = 1, size( wind) {               # sweep through a window
     call line2cart(  nwind, i, ii)      #   ii(i)   is (i1,i2) in window
     call cart2line(  nwall,    ii, j)   # j(ii)     linear subscript in window
     if( adj)
        wall( j + shift) += wind( i)
     else
        wind( i)         += wall( j + shift)
  }
#% _close
ipatch = ipatch + 1
}
```

The cartesian vector `jj` points to the beginning of a patch, where on the wall the (1,1,..) coordinate of the patch lies. Obviously this begins at the beginning edge of the wall. Then we pick `jj` so that the last patch on any axis has its last point exactly abutting the end of the axis. The formula for doing this would divide by zero for a wall with only one patch on it. This case arises legitimately where an axis has length one. Thus we handle the case `npatch=1` by abutting the patch to the beginning of the wall and forgetting about its end. As in any code mixing integers with floats, to guard against having a floating-point number, say 99.9999, rounding down to 99 instead of up to 100, the rule is to always add .5 to a floating point number the moment before converting it to an integer. Now we are ready to sweep a window to or from the wall. The number of points in a window is `size(wind)` or equivalently `product(nwind)`. Figure 9.2 shows an example with five nonoverlapping patches on the 1-axis and many overlapping patches on the 2-axis.

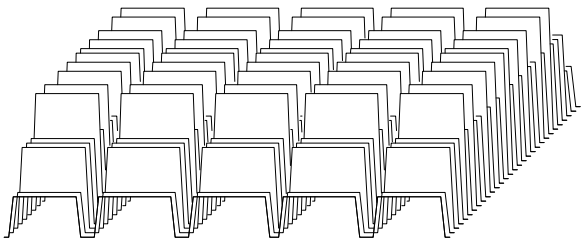

Figure 9.2: A plane of identical values after patches have been cut and then added back. Results are shown for `nwall=(100,30)`, `nwind=(17,6)`, `npatch=(5,11)`. For these parameters, there is gapping on the horizontal axis and overlap on the depth axis. pch-parcel90 [ER]

9.1.1 Weighting and reconstructing

The adjoint of extracting all the patches is adding them back. Because of the overlaps, the adjoint is not the inverse. In many applications, **inverse patching** is required; i.e. patching things back together seamlessly. This can be done with weighting functions. You can have any weighting function you wish and I will provide you the patching reconstruction operator

$\tilde{\mathbf{I}}_p$ in

$$\tilde{\mathbf{d}} \quad = \quad [\mathbf{W}_{\text{wall}}\mathbf{P}'\mathbf{W}_{\text{wind}}\mathbf{P}]\mathbf{d} \quad = \quad \tilde{\mathbf{I}}_p\,\mathbf{d} \tag{9.1}$$

where $\mathbf{d}$ is your initial data, $\tilde{\mathbf{d}}$ is the reconstructed data, $\mathbf{P}$ is the patching operator, $\mathbf{P}'$ is adjoint patching (adding the patches). $\mathbf{W}_{\text{wind}}$ is your chosen weighting function in the window, and $\mathbf{W}_{\text{wall}}$ is the weighting function for the whole wall. You specify any $\mathbf{W}_{\text{wind}}$ you like, and module `mkwallwt` below builds the weighting function $\mathbf{W}_{\text{wall}}$ that you need to apply to your wall of reconstructed data, so it will undo the nasty effects of the overlap of windows and the shape of your window-weighting function. You do not need to change your window weighting function when you increase or decrease the amount of overlap between windows because $\mathbf{W}_{\text{wall}}$ takes care of it. The method is to use adjoint `patch` on page 212 to add the weights of each window onto the wall and finally to invert the sum wherever it is non-zero. (You lose data wherever the sum is zero).

```
module mkwallwt {               # N-dimensional wall weights
use patch
contains
  subroutine wallwtn( n, nwall, nwind, windwt, wallwt) {
    integer, dimension( :), pointer      :: n, nwall, nwind
    real,    dimension( :), intent( in)  :: windwt
    real,    dimension( :), intent( out) :: wallwt
    integer                              :: ipatch, stat
    wallwt = 0.
    call patch_init( n, nwall, nwind)
    do ipatch = 1, product( n) {
       stat = patch_lop( .true., .true., wallwt, windwt)
       call patch_close ()
    }
    where( wallwt /= 0.) wallwt = 1. / wallwt
  }
}
```

No matrices are needed to show that this method succeeds, because data values are never mixed with one another. An equation for any reconstructed data value $\tilde{d}$ as a function of the original value d and the weights w_i that hit d is $\tilde{d} = (\sum_i w_i d)/\sum_i w_i = d$. Thus, our process is simply a "partition of unity."

To demonstrate the program, I made a random weighting function to use in each window with positive random numbers. The general strategy allows us to use different weights in different windows. That flexibility adds clutter, however, so here we simply use the same weighting function in each window.

The operator $\tilde{\mathbf{I}}_p$ is called "**idempotent**." The word "idempotent" means "self-power," because for any N, $0^N = 0$ and $1^N = 1$, thus the numbers 0 and 1 share the property that raised to any power they remain themselves. Likewise, the patching reconstruction operator multiplies every data value by either one or zero. Figure 9.3 shows the result obtained when a plane of identical constant values $\mathbf{d}$ is passed into the patching reconstruction operator $\tilde{\mathbf{I}}_p$. The result is constant on the 2-axis, which confirms that there is adequate sampling on the 2-axis,

and although the weighting function is made of random numbers, all trace of random numbers has disappeared from the output. On the 1-axis the output is constant, except for being zero in gaps, because the windows do not overlap on the 1-axis.

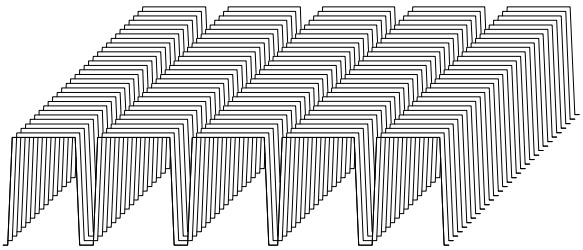

Figure 9.3: A plane of identical values passed through the idempotent patching reconstruction operator. Results are shown for the same parameters as Figure 9.2. pch-idempatch90 [ER]

Module `patching` assists in reusing the patching technique. It takes a linear operator **F**. as its argument and applies it in patches. Mathematically, this is $[\mathbf{W}_{\text{wall}}\mathbf{P}'\mathbf{W}_{\text{wind}}\mathbf{F}\mathbf{P}]\mathbf{d}$. It is assumed that the input and output sizes for the operator `oper` are equal.

```
module patching {                   # Apply existing operator in all patches
  use patch
  use mkwallwt
contains
  subroutine patchingn( oper, modl, data, npatch, nwall, nwind, windwt) {
    interface {
       integer function oper(adj, add, modl, data) {
          logical, intent (in) :: adj, add
          real, dimension( :)  :: modl, data
          }
       }
    real,    dimension( :), intent( in)  :: windwt, modl
    real,    dimension( :), intent( out) :: data
    integer, dimension( :), pointer      :: npatch, nwall, nwind
    real,    dimension( size( modl))     :: wallwt
    real,    dimension( size( windwt))   :: winmodl, windata
    integer                              :: i, stat1, stat2
    data = 0.
    call patch_init( npatch, nwall, nwind)
    do i = 1, product( npatch) {
       stat1 = patch_lop( .false., .false., modl, winmodl)
       stat2 = oper(      .false., .false., winmodl, windata)
       stat1 = patch_lop( .true. , .true. , data,  windwt * windata)
       call patch_close ()
       }
    call wallwtn( npatch, nwall, nwind, windwt, wallwt);  data = data * wallwt
    }
}
```

9.1.2 2-D filtering in patches

A way to do time- and space-variable filtering is to do invariant filtering within each patch. Typically, we apply a filter, say $\mathbf{F}_p$, in each patch. The composite operator, filtering in patches,

$\tilde{\mathbf{F}}$, is given by

$$\tilde{\mathbf{d}} \quad = \quad [\mathbf{W}_{\text{wall}}\mathbf{P}'\mathbf{W}_{\text{wind}}\mathbf{F}_p\mathbf{P}]\,\mathbf{d} \quad = \quad \tilde{\mathbf{F}}\,\mathbf{d} \tag{9.2}$$

I built a triangular weighting routine `tentn()` that tapers from the center of the patch of the filter's *outputs* towards the edges. Accomplishing this weighting is complicated by (1) the constraint that the filter must not move off the edge of the input patch and (2) the alignment of the input and the output. The layout for prediction-error filters is shown in Figure 9.4.

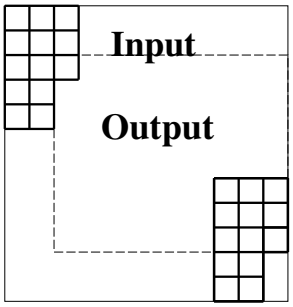

Figure 9.4: Domain of inputs and outputs of a two-dimensional prediction-error filter. pch-rabdomain [NR]

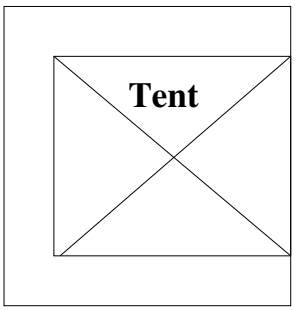

Figure 9.5: Placement of tent-like weighting function in the space of filter inputs and outputs. pch-rabtent [NR]

We need a weighting function that vanishes where the filter has no outputs. The amplitude of the weighting function is not very important because we have learned how to put signals back together properly for arbitrary weighting functions. We can use any pyramidal or tent-like shape that drops to zero outside the domain of the filter output. The job is done by subroutine `tentn()`. A new parameter needed by `tentn` is `a`, the coordinate of the beginning of the tent.

```
module tent {  # triangle tent weights in N dimensions
use cartesian
contains
subroutine tentn ( nwind, center, a, windwt) {
  integer, dimension (:), intent (in)  :: nwind, center, a
  real,    dimension (:), intent (out) :: windwt
  integer, dimension( size( nwind))    :: start, end, x
  real,    dimension( size( nwind))    :: mid, wid
  integer                              :: i
  start= 1+a-center;     end= nwind-center+1         # a is beginning of tent
  mid=  (end+start)/2.;  wid= (end-start+1.)/2.
  do i = 1, size( windwt) {
       call line2cart(  nwind, i, x)
       if( all( x >= start) .and. all( x <= end))
```

```
            windwt( i) = product( max( 0., 1. - abs((x-mid)/wid)))
        else
            windwt( i) = 0.
        }
    }
}
```

In applications where triangle weights are needed on the *inputs* (or where we can work on a patch without having interference with edges), we can get "triangle tent" weights from `tentn()` if we set filter dimensions and lags to unity, as shown in Figure 9.6.

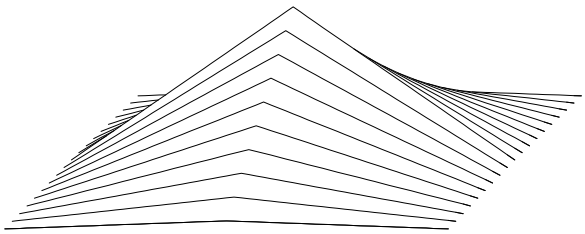

Figure 9.6: Window weights from `tentn()` with `nwind=(61,19), center=(31,1), a=(1,1)` . pch-windwt90 [ER]

Triangle weighting functions can sum to a constant if the spacing is such that the midpoint of one triangle is at the beginning of the next. I imagined in two dimensions that something similar would happen with shapes like Egyptian pyramids of Cheops, $2-|x-y|+|x+y|$. Instead, the equation $(1-|x|)(1-|y|)$ which has the tent-like shape shown in Figure 9.6 adds up to the constant flat top shown in Figure 9.7. (To add interest to Figure 9.7, I separated the windows by a little more than the precise matching distance.) In practice we may chose window shapes and overlaps for reasons other than the constancy of the sum of weights, because `mkwallwt` on page 214 accounts for that.

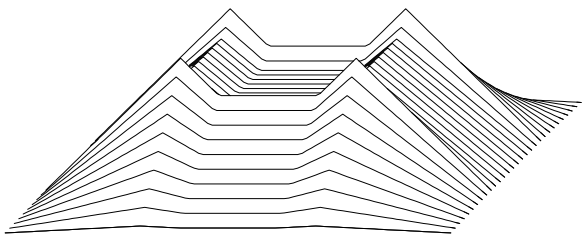

Figure 9.7: (Inverse) wall weights with `n1=100, w1=61, k1=2, n2=30, w2=19, k2=2` pch-wallwt90 [ER]

Finally is an example of filtering a plane of uniform constants with an impulse function. The impulse function is surrounded by zeros, so the filter output patches are smaller than the input patches back in Figure 9.3. Here in Figure 9.8, both axes need more window density.

9.1.3 Designing a separate filter for each patch

Recall the prediction-error filter subroutine `find_pef()` on page 157. Given a data plane, this subroutine finds a filter that tends to whiten the spectrum of that data plane. The output is white residual. Now suppose we have a data plane where the dip spectrum is changing from place to place. Here it is natural to apply subroutine `find_pef()` in local patches. This is

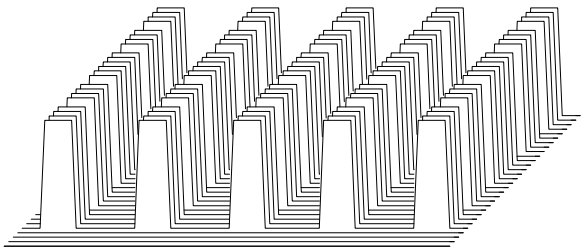

Figure 9.8: Filtering in patches with the same parameters as in Figures 9.2 and 9.3. Additionally, the filter parameters are `a1=11 a2=5 lag1=6 lag2=1` . Thus, windows are centered on the 1-axis and pushed back out the 2-axis. pch-cinloip90 [ER]

done by subroutine `find_lopef()`. The output of this subroutine is an array of helix-type filters, which can be used, for example, in a local convolution operator `loconvol` on this page.

```
module lopef {            # Local PEF estimation in patches.
  use patch               # Estimate a vector of filters, one for each patch.
  use misinput
  use pef
contains
  subroutine find_lopef( wall, aa, npatch, nwall, nwind, mask) {
    optional                                  :: mask
    integer,       dimension(:), pointer      :: npatch, nwall, nwind
    real,          dimension(:), intent( in) :: wall, mask
    type( filter), dimension(:)               :: aa
    real,          dimension(:), pointer      :: windata, winmask
    integer                                   :: i, stat
                        allocate( windata( product( nwind)))   # a patch
    if( present( mask)) allocate( winmask( product( nwind)))   # missing inputs
    call patch_init( npatch, nwall, nwind)
    do i = 1, product( npatch) {                               # do all patches
       stat = patch_lop( .false., .false., wall, windata)      # get a patch
       if( present( mask)) {
           stat = patch_lop( .false., .false., mask, winmask)
           call find_mask( (winmask /= 0.), aa (i))             # missing data
           }
       if( count(.not.aa(i)%mis) > size(aa(i)%lag))             # enuf eqns?
          call find_pef( windata, aa(i), niter=size( aa(i)%lag)) # find PEF
       else if( i > 1)
          aa(i)%flt = aa(i-1)%flt                               # use last PEF
       call patch_close()
       }
    deallocate( windata)
    if( present( mask))  deallocate( winmask)
    }
}
#       if( size(aa(i)%mis) - count(aa(i)%mis) > size(aa(i)%lag)) # enuf eqns?
```

We notice that when a patch has fewer regression equations than the filter has coefficients, then the filter is taken to be that of the previous patch.

```
# successive invocations apply successive filters from a vector.
#                     (will fail a dot product test?  Oft used with patching.)
```

```
module loconvol {
use helicon
integer, private  :: i
type( filter), dimension(:), pointer :: aa
#% _init( aa)
i = 0
#% _lop( xx, yy)
 integer stat1;   i = i + 1
 call helicon_init( aa( i))
 stat1 = helicon_lop( adj, .false., xx, yy)
}
```

9.1.4 Triangular patches

I have been running patching code for several years and my first general comment is that realistic applications often call for patches of different sizes and different shapes. (Tutorial, non-interactive Fortran code is poorly suited to this need.) Raw seismic data in particular seems more suited to triangular shapes. It is worth noting that the basic concepts in this chapter have ready extension to other shapes. For example, a rectangular shape could be duplicated into two identical patches; then data in one could be zeroed above the diagonal and in the other below; you would have to allow, of course, for overlap the size of the filter. Module `pef` on page 157 automatically ignores the zeroed portion of the triangle, and it is irrelevant what `mis2()` on page 132 does with a zeroed portion of data, if a triangular footprint of weights is designed to ignore its output.

EXERCISES:

1 Code the linear operator $\mathbf{W}_{\text{wall}}\mathbf{P}'\mathbf{W}_{\text{wind}}\mathbf{P}$ including its adjoint.

2 **Smoothing program.** Some familiar operations can be seen in a new light when done in patches. Patch the data. In each patch, find the mean value. Replace each value by the mean value. Reconstruct the wall.

3 **Smoothing while filling missing data.** This is like smoothing, but you set window weights to zero where there is no data. Because there will be a different set of weights in each window, you will need to make a simple generalization to `mkwallwt` on page 214.

4 **Gain control.** Divide the data into patches. Compute the square root of the sum of the squares of all the data in each patch. Divide all values in that patch by this amount. Reassemble patches.

9.2 STEEP-DIP DECON

Normally, when an autoregression filter (PEF) predicts a value at a point it uses values at earlier points. In practice, a gap may also be set between the predicted value and the earlier values.

What is not normally done is to supplement the fitting signals on nearby traces. That is what we do here. We allow the prediction of a signal to include nearby signals at earlier times. The times accepted in the goal are inside a triangle of velocity less than about the water velocity. The new information allowed in the prediction is extremely valuable for water-velocity events. Wavefronts are especially predictable when we can view them along the wavefront (compared to perpendicular or at some other angle from the wavefront). It is even better on land, where noises move more slowly at irregular velocities, and are more likely to be aliased.

Using `lopef` on page 218, the overall process proceeds independently in each of many overlapping windows. The most important practical aspect is the filter masks, described next.

9.2.1 Dip rejecting known-velocity waves

Consider the two-dimensional filter

$$\begin{array}{ccc} & +1 & \\ -1 & 0 & -1 \\ & +1 & \end{array} \tag{9.3}$$

When this this filter is applied to a field profile with 4 ms time sampling and 6 m trace spacing, it should perfectly extinguish 1.5 km/s water-velocity noises. Likewise, the filter

$$\begin{array}{ccc} & +1 & \\ & 0 & \\ & 0 & \\ -1 & 0 & -1 \\ & 0 & \\ & 0 & \\ & +1 & \end{array} \tag{9.4}$$

should perfectly extinguish water noise when the trace spacing is 18 m. Such noise is, of course, spatially aliased for all temporal frequencies above 1/3 of Nyquist, but that does not matter. The filter extinguishes them perfectly anyway. Inevitably, the filter cannot both extinguish the noise and leave the signal untouched where the alias of one is equal to the other. So we expect the signal to be altered where it matches aliased noise. This simple filter does worse than that. On horizontal layers, for example, signal wavelets become filtered by $(1,0,0,-2,0,0,1)$. If the noise is overwhelming, this signal distortion is a small price to pay for eliminating it. If the noise is tiny, however, the distortion is unforgivable. In the real world, data-adaptive deconvolution is usually a good compromise.

The two-dimensional deconvolutions filters we explore here look like this:

$$
\begin{array}{ccccccccc}
x & x & x & x & x & x & x & x & x \\
x & x & x & x & x & x & x & x & x \\
. & x & x & x & x & x & x & x & . \\
. & x & x & x & x & x & x & x & . \\
. & x & x & x & x & x & x & x & . \\
. & . & x & x & x & x & x & . & . \\
. & . & x & x & x & x & x & . & . \\
. & . & x & x & x & x & x & . & . \\
. & . & . & x & x & x & . & . & . \\
. & . & . & x & x & x & . & . & . \\
. & . & . & x & x & x & . & . & . \\
. & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . \\
. & . & . & . & 1 & . & . & . & .
\end{array}
\tag{9.5}
$$

where each . denotes a zero and each x denotes a (different) adjustable filter coefficient that is chosen to minimize the power out.

You can easily imagine variations on this shape, such as a diamond instead of a triangle. I invite you to experiment with the various shapes that suggest themselves.

9.2.2 Tests of steep-dip decon on field data

Low-velocity noises on shot records are often not fully suppressed by stacking because the noises are spatially aliased. Routine field arrays are not perfect and the noise is often extremely strong. An interesting, recently-arrived data set worth testing is shown in Figure 9.9.

I scanned the forty **Yilmaz** and **Cumro** shot profiles for strong low-velocity noises and I selected six examples. To each I applied an AGC that is a slow function of time and space (triangle smoothing windows with triangle half-widths of 200 time points and 4 channels). Because my process simultaneously does both low-velocity rejection and deconvolution, I prepared more traditional 1-D deconvolutions for comparison. This is done in windows of 250 time points and 25 channels, the same filter being used for each of the 25 channels in the window. In practice, of course, considerably more thought would be given to optimal window sizes as a function of the regional nature of the data. The windows were overlapped by about 50%. The same windows are used on the steep-dip deconvolution.

It turned out to be much easier than expected and on the first try I got good results on all all six field profiles tested. I have not yet tweaked the many adjustable parameters. As you inspect these deconvolved profiles from different areas of the world with different recording methods, land and marine, think about how the stacks should be improved by the deconvolution. Stanford Exploration Project report 77 (SEP-77) shows the full suite of results. Figure 9.10 is a sample of them.

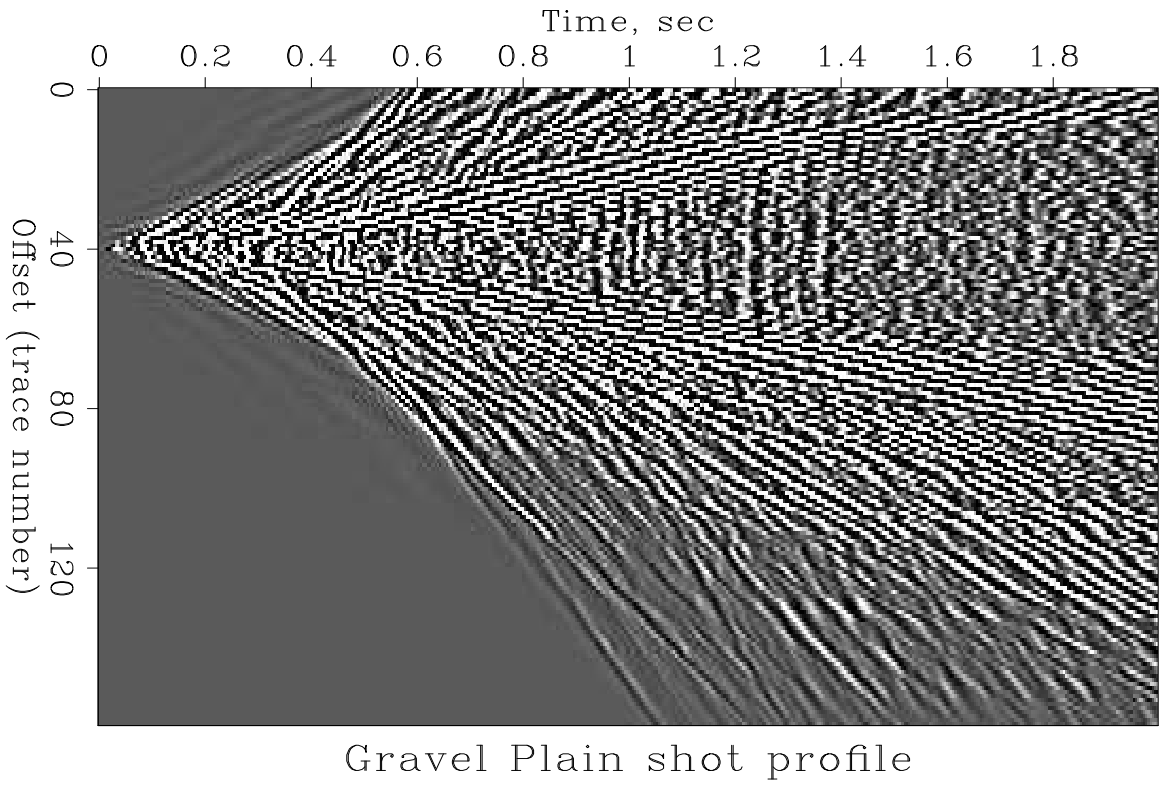

Figure 9.9: Gravel plain ground roll (Middle East) Worth testing. pch-gravel2D [ER]

Unexpectedly, results showed that 1-D deconvolution also suppresses low-velocity noises. An explanation can be that these noises are often either low-frequency or quasimonochromatic.

As a minor matter, fundamentally, my code cannot work ideally along the side boundaries because there is no output (so I replaced it by the variance scaled input). With a little extra coding, better outputs could be produced along the sides if we used spatially one-sided filters like

$$\begin{matrix} x & x & x & x & x \\ . & x & x & x & x \\ . & x & x & x & x \\ . & . & x & x & x \\ . & . & x & x & x \\ . & . & . & x & x \\ . & . & . & x & x \\ . & . & . & . & . \\ . & . & . & . & . \\ . & . & . & . & 1 \end{matrix} \tag{9.6}$$

These would be applied on one side of the shot and the opposite orientation would be applied on the other side. With many kinds of data sets, such as off-end marine recording in which a ship tows a hydrophone streamer, the above filter might be better in the interior too.

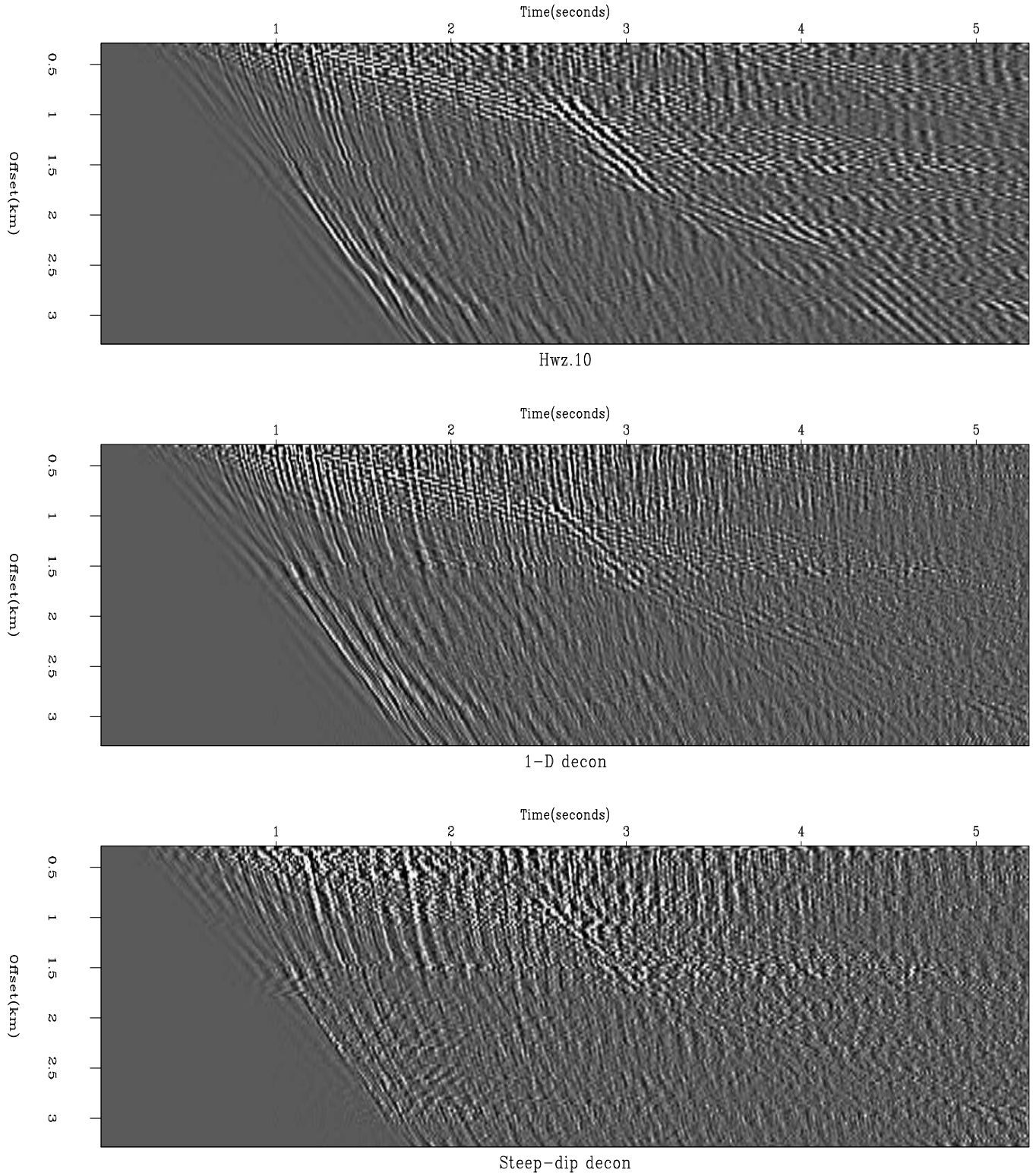

Figure 9.10: Top is a North African vibrator shot profile (Y&C #10) after AGC. Middle is gapped 1-D decon. Bottom is steep-dip decon. pch-wz.1090 [ER,M]

9.2.3 Are field arrays really needed?

Field arrays cancel random noise but their main function, I believe, is to cancel low-velocity coherent noises, something we now see is handled effectively by steep-dip deconvolution. While I do not advocate abandoning field arrays, it is pleasing to notice that with the arrival of steep-dip deconvolution, we are no longer so dependent on field arrays and perhaps coherent noises can be controlled where field arrays are impractical, as in certain 3-D geometries. A recently arrived 3-D shot profile from the sand dunes in the Middle East is Figure 9.11. The strong hyperbolas are **ground roll** seen in a line that does not include the shot. The open question here is, how should we formulate the problem of ground-roll removal in 3-D?

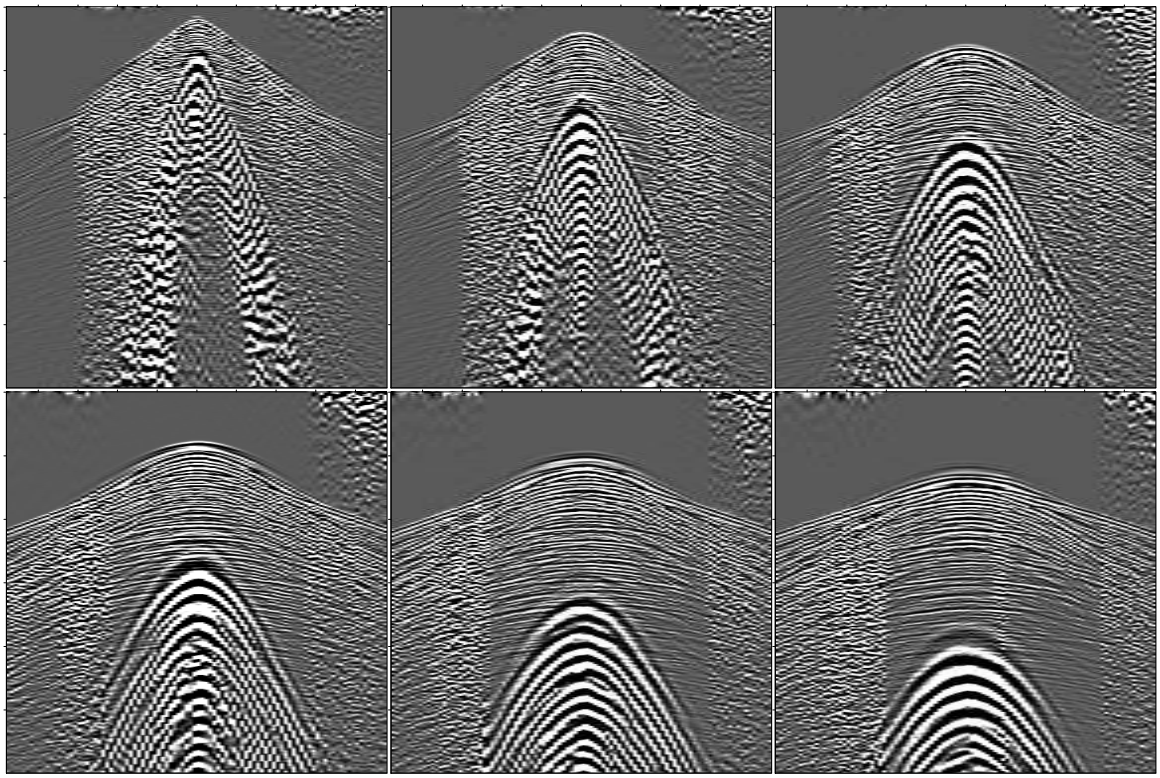

Figure 9.11: Sand dunes. One shot, six parallel receiver lines. pch-dune3D [ER,M]

9.2.4 Which coefficients are really needed?

Steep-dip decon is a heavy consumer of computer time. Many small optimizations could be done, but more importantly, I feel there are some deeper issues that warrant further investigation. The first question is, how many filter coefficients should there be and where should they be? We would like to keep the number of nonzero filter coefficients to a minimum because it would speed the computation, but more importantly I fear the filter output might be defective in some insidious way (perhaps missing primaries) when too many filter coefficients are used. Perhaps if 1-D decon were done sequentially with steep-dip decon the number of

free parameters (and hence the amount of computer time) could be dropped even further. I looked at some of the filters and they scatter wildly with the Nyquist frequency (particularly those coefficients on the trace with the "1" constraint). This suggests using a damping term on the filter coefficients, after which perhaps the magnitude of a filter coefficient will be a better measure of whether this practice is really helpful. Also, it would, of course, be fun to get some complete data sets (rather than a single shot profile) to see the difference in the final stack.

9.3 INVERSION AND NOISE REMOVAL

Here we relate the basic theoretical statement of geophysical inverse theory to the basic theoretical statement of separation of signals from noises.

A common form of linearized **geophysical inverse theory** is

$$\mathbf{0} \approx \mathbf{W}(\mathbf{Lm}-\mathbf{d}) \tag{9.7}$$
$$\mathbf{0} \approx \epsilon\mathbf{Am} \tag{9.8}$$

We choose the operator $\mathbf{L}=\mathbf{I}$ to be an identity and we rename the model $\mathbf{m}$ to be signal $\mathbf{s}$. Define noise by the decomposition of data into signal plus noise, so $\mathbf{n}=\mathbf{d}-\mathbf{s}$. Finally, let us rename the weighting (and filtering) operations $\mathbf{W}=\mathbf{N}$ on the noise and $\mathbf{A}=\mathbf{S}$ on the signal. Thus the usual model fitting becomes a fitting for signal-noise separation:

$$0 \approx \mathbf{N}(-\mathbf{n}) = \mathbf{N}(\mathbf{s}-\mathbf{d}) \tag{9.9}$$
$$0 \approx \epsilon\mathbf{Ss} \tag{9.10}$$

9.4 SIGNAL-NOISE DECOMPOSITION BY DIP

Choose noise $\mathbf{n}$ to be energy that has no spatial correlation and signal $\mathbf{s}$ to be energy with spatial correlation consistent with one, two, or possibly a few plane-wave segments. (Another view of noise is that a huge number of plane waves is required to define the wavefield; in other words, with **Fourier analysis** you can make anything, signal or noise.) We know that a first-order differential equation can absorb (kill) a single plane wave, a second-order equation can absorb one or two plane waves, etc. In practice, we will choose the order of the wavefield and minimize power to absorb all we can, and call that the signal. $\mathbf{S}$ is the operator that absorbs (by prediction error) the plane waves and $\mathbf{N}$ absorbs noises and $\epsilon > 0$ is a small scalar to be chosen. The difference between $\mathbf{S}$ and $\mathbf{N}$ is the spatial order of the filters. Because we regard the noise as spatially uncorrelated, $\mathbf{N}$ has coefficients only on the time axis. Coefficients for $\mathbf{S}$ are distributed over time and space. They have one space level, plus another level for each plane-wave segment slope that we deem to be locally present. In the examples here the number of slopes is taken to be two. Where a data field seems to require more than two slopes, it usually means the "patch" could be made smaller.

It would be nice if we could forget about the goal (9.10) but without it the goal (9.9), would simply set the signal $\mathbf{s}$ equal to the data $\mathbf{d}$. Choosing the value of ϵ will determine in some

way the amount of data energy partitioned into each. The last thing we will do is choose the value of ϵ, and if we do not find a theory for it, we will experiment.

The operators $\mathbf{S}$ and $\mathbf{N}$ can be thought of as "leveling" operators. The method of least-squares sees mainly big things, and spectral zeros in $\mathbf{S}$ and $\mathbf{N}$ tend to cancel spectral lines and plane waves in $\mathbf{s}$ and $\mathbf{n}$. (Here we assume that power levels remain fairly level in time. Were power levels to fluctuate in time, the operators $\mathbf{S}$ and $\mathbf{N}$ should be designed to level them out too.)

None of this is new or exciting in one dimension, but I find it exciting in more dimensions. In seismology, quasisinusoidal signals and noises are quite rare, whereas local plane waves are abundant. Just as a short one-dimensional filter can absorb a sinusoid of any frequency, a compact two-dimensional filter can absorb a wavefront of any dip.

To review basic concepts, suppose we are in the one-dimensional frequency domain. Then the solution to the fitting goals (9.10) and (9.9) amounts to minimizing a quadratic form by setting to zero its derivative, say

$$0 \quad = \quad \frac{\partial}{\partial \mathbf{s}'} \left((\mathbf{s}' - \mathbf{d}') \mathbf{N}' \mathbf{N} (\mathbf{s} - \mathbf{d}) + \epsilon^2 \mathbf{s}' \mathbf{S}' \mathbf{S} \mathbf{s} \right) \tag{9.11}$$

which gives the answer

$$\mathbf{s} \quad = \quad \left(\frac{\mathbf{N}'\mathbf{N}}{\mathbf{N}'\mathbf{N} + \epsilon^2 \mathbf{S}'\mathbf{S}} \right) \mathbf{d} \tag{9.12}$$

$$\mathbf{n} \quad = \quad \mathbf{d} - \mathbf{s} \quad = \quad \left(\frac{\epsilon^2 \mathbf{S}'\mathbf{S}}{\mathbf{N}'\mathbf{N} + \epsilon^2 \mathbf{S}'\mathbf{S}} \right) \mathbf{d} \tag{9.13}$$

To make this really concrete, consider its meaning in one dimension, where signal is white $\mathbf{S}'\mathbf{S} = 1$ and noise has the frequency ω_0, which is killable with the multiplier $\mathbf{N}'\mathbf{N} = (\omega - \omega_0)^2$. Now we recognize that equation (9.12) is a notch filter and equation (9.13) is a narrow-band filter.

The analytic solutions in equations (9.12) and (9.13) are valid in 2-D **Fourier space** or dip space too. I prefer to compute them in the time and space domain to give me tighter control on window boundaries, but the Fourier solutions give insight and offer a computational speed advantage.

Let us express the fitting goal in the form needed in computation.

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \quad \approx \quad \begin{bmatrix} \mathbf{N} \\ \epsilon \mathbf{S} \end{bmatrix} \mathbf{s} + \begin{bmatrix} -\mathbf{N}\mathbf{d} \\ \mathbf{0} \end{bmatrix} \tag{9.14}$$

```
module signoi {
  use helicon
  use polydiv
  use solver_mod
  use cgstep_mod
  integer, private                    :: i
  integer                             :: niter, nd
```

```
  real                                    :: eps
  type( filter), dimension( :), pointer :: nn, ss
  real, dimension (nd), allocatable     :: dd
#% _init (nn, ss, niter, nd, eps)
i = 0
#% _lop (data, sign)
    integer stat1; i = i + 1
    call helicon_init (nn (i))
    call polydiv_init (nd, ss (i))
    stat1 = helicon_lop (.false., .false., data, dd)
    call solver_prec (helicon_lop, cgstep, niter= niter, x=sign, dat = dd,
                 prec= polydiv_lop, nprec= nd, eps= eps)
    call cgstep_close()
#% _close
    call polydiv_close()
}
```

As with the missing-data subroutines, the potential number of iterations is large, because the dimensionality of the space of unknowns is much larger than the number of iterations we would find acceptable. Thus, sometimes changing the number of iterations `niter` can create a larger change than changing `epsilon`. Experience shows that helix preconditioning saves the day.

9.4.1 Signal/noise decomposition examples

Figure 9.12 demonstrates the signal/noise decomposition concept on synthetic data. The signal and noise have similar frequency spectra but different dip spectra.

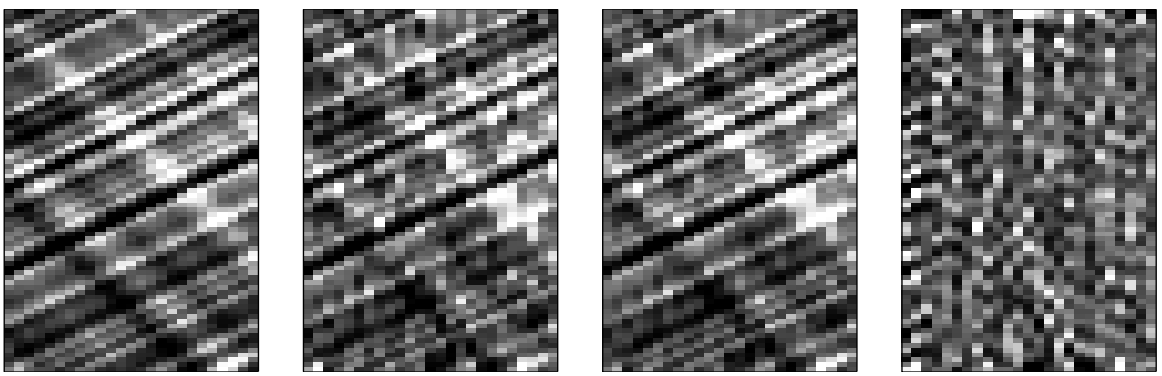

Figure 9.12: The input signal is on the left. Next is that signal with noise added. Next, for my favorite value of `epsilon=1.`, is the estimated signal and the estimated noise. pch-signoi90 [ER]

Before I discovered helix preconditioning, Ray Abma found that different results were obtained when the fitting goal was cast in terms of **n** instead of **s**. Theoretically it should not make any difference. Now I believe that with preconditioning, or even without it, if there are

enough iterations, the solution should be independent of whether the fitting goal is cast with either **n** or **s**.

Figure 9.13 shows the result of experimenting with the choice of ϵ. As expected, increasing ϵ weakens **s** and increases **n**. When ϵ is too small, the noise is small and the signal is almost the original data. When ϵ is too large, the signal is small and coherent events are pushed into the noise. (Figure 9.13 rescales both signal and noise images for the clearest display.)

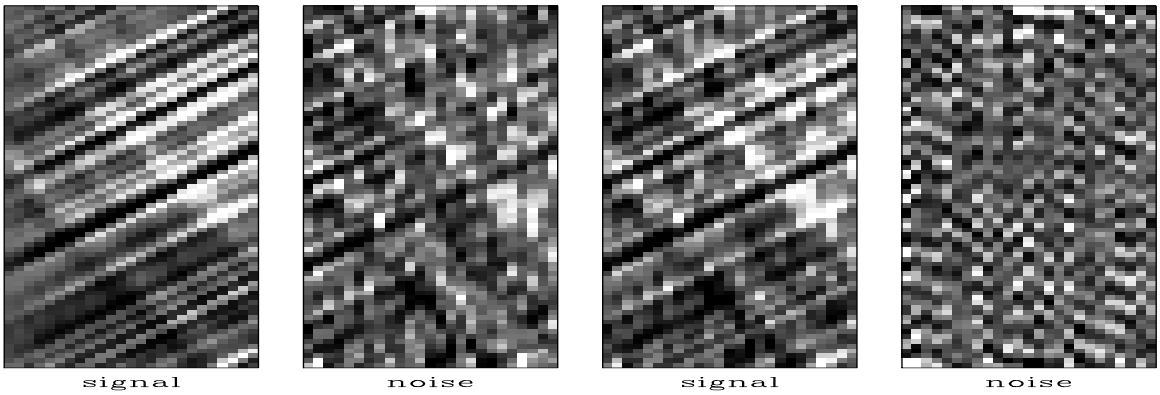

Figure 9.13: Left is an estimated signal-noise pair where `epsilon=4` has improved the appearance of the estimated signal but some coherent events have been pushed into the noise. Right is a signal-noise pair where `epsilon=.25`, has improved the appearance of the estimated noise but the estimated signal looks no better than original data. pch-signeps90 [ER]

Notice that the leveling operators **S** and **N** were both estimated from the original signal and noise mixture $\mathbf{d} = \mathbf{s} + \mathbf{n}$ shown in Figure 9.12. Presumably we could do even better if we were to reestimate **S** and **N** from the estimates **s** and **n** in Figure 9.13.

9.4.2 Noise removal on Shearer's data

Professor Peter **Shearer**[1] gathered the earthquakes from the IDA network, an array of about 25 widely distributed gravimeters, donated by Cecil Green, and Shearer selected most of the shallow-depth earthquakes of magnitude greater than about 6 over the 1981-91 time interval, and sorted them by epicentral distance into bins 1° wide and stacked them. He generously shared his edited data with me and I have been restacking it, compensating for amplitude in various ways, and planning time and filtering compensations.

Figure 9.14 shows a test of noise subtraction by multidip narrow-pass filtering on the **Shearer-IDA stack**. As with prediction there is a general reduction of the noise. Unlike with prediction, weak events are preserved and noise is subtracted from them too.

[1] I received the data for this stack from Peter Shearer at the **Cecil and Ida Green** Institute of Geophysics and Planetary Physics of the Scripps Oceanographic Institute. I also received his permission to redistribute it to friends and colleagues. Should you have occasion to copy it please reference (Shearer, 1991a) (Shearer, 1991b) it properly. Examples of earlier versions of these stacks are found in the references. Professor Shearer may be willing to supply newer and better stacks. His electronic mail address is `shearer@mahi.ucsd.edu`.

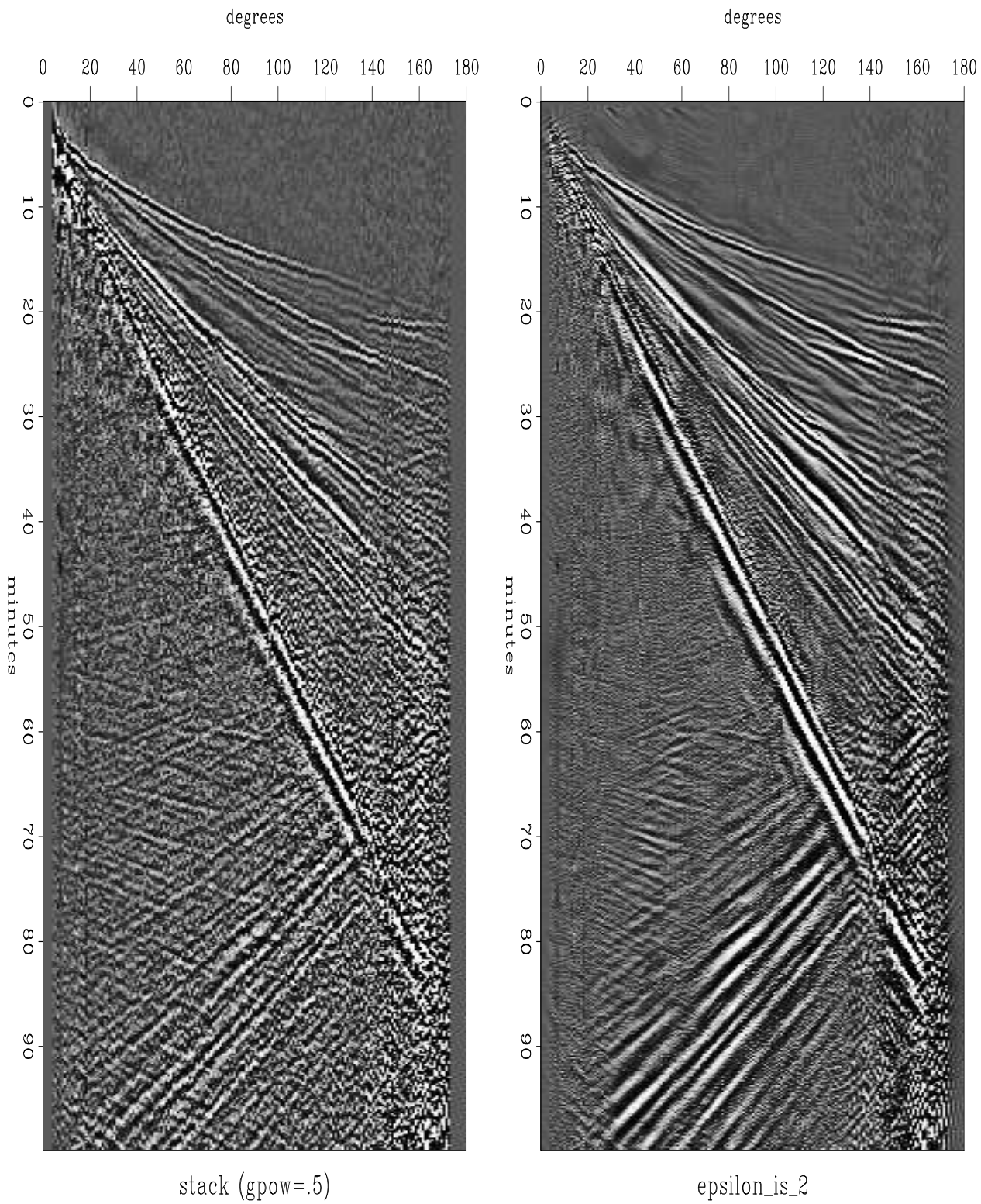

Figure 9.14: Stack of Shearer's IDA data (left). Multidip filtered (right). It is pleasing that the noise is reduced while weak events are preserved. pch-sneps+290 [CR,M]

Besides the difference in theory, the separation filters are much smaller because their size is determined by the concept that "two dips will fit anything locally" (`a2=3`), versus the prediction filters "needing a sizeable window to do statistical averaging." The same aspect ratio `a1/a2` is kept and the page is now divided into 11 vertical patches and 24 horizontal patches (whereas previously the page was divided in 3×4 patches). In both cases the patches overlap about 50%. In both cases I chose to have about ten times as many equations as unknowns on each axis in the estimation. The ten degrees of freedom could be distributed differently along the two axes, but I saw no reason to do so.

9.4.3 The human eye as a dip filter

Although the filter seems to be performing as anticipated, no new events are apparent. I believe the reason that we see no new events is that the competition is too tough. We are competing with the human eye, which through aeons of survival has become is a highly skilled filter. Does this mean that there is no need for filter theory and filter subroutines because the eye can do it equally well? It would seem so. Why then pursue the subject matter of this book?

The answer is 3-D. The human eye is not a perfect filter. It has a limited (though impressive) dynamic range. A nonlinear display (such as wiggle traces) can prevent it from averaging. The eye is particularly good at dip filtering, because the paper can be looked at from a range of grazing angles and averaging window sizes miraculously adjust to the circumstances. The eye can be overwhelmed by too much data. The real problem with the human eye is that the retina is only two-dimensional. The world contains many three-dimensional data volumes. I don't mean the simple kind of 3-D where the contents of the room are nicely mapped onto your 2-D retina. I mean the kind of 3-D found inside a bowl of soup or inside a rock. A rock can be sliced and sliced and sliced again and each slice is a picture. The totality of these slices is a movie. The eye has a limited ability to deal with **movies** by optical persistence, an averaging of all pictures shown in about 1/10 second interval. Further, the eye can follow a moving object and perform the same averaging. I have learned, however, that the eye really cannot follow two objects at two different speeds and average them both over time. Now think of the third dimension in Figure 9.14. It is the dimension that I summed over to make the figure. It is the 1° range bin. If we were viewing the many earthquakes in each bin, we would no longer be able to see the out-of-plane information which is the in-plane information in Figure 9.14.

To view genuinely 3-D information we must see a movie, or we must compress the 3-D to 2-D. There are only a small number of ways to compress 3-D to 2-D. One is to select planes from the volume. One is to sum the volume over one of its axes, and the other is a compromise, a filtering over the axis we wish to abandon before subsampling on it. That filtering is a local smoothing. If the local smoothing has motion (out of plane dip) of various velocities (various dips), then the desired process of smoothing the out of plane direction is what we did in the in-plane direction in Figure 9.14. But Figure 9.14 amounts to more than that. It amounts to a kind of simultaneous smoothing in the *two* most coherent directions whereas in 3-D your eye can smooth in only *one* direction when you turn your head along with the motion.

> If the purpose of data processing is to collapse 3-D data volumes to 2-D where they are comprehensible to the human eye, then perhaps data-slope adaptive, low-pass filtering in the out-of-plane direction is the best process we can invent.

My purpose in filtering the earthquake stacks is to form a guiding "pilot trace" to the analysis of the traces *within* the bin. Within each bin, each trace needs small time shifts and perhaps a small temporal filter to best compensate it to . . . to what? to the pilot trace, which in these figures was simply a stack of traces in the bin. Now that we have filtered in the range direction, however, the next stack can be made with a better quality pilot.

9.5 SPACE-VARIABLE DECONVOLUTION

Filters sometimes change with time and space. We sometimes observe signals whose spectrum changes with position. A filter that changes with position is called nonstationary. We need an extension of our usual convolution operator `hconest` on page 152. Conceptually, little needs to be changed besides changing `aa(ia)` to `aa(ia,iy)`. But there is a practical problem. Fomel and I have made the decision to clutter up the code somewhat to save a great deal of memory. This should be important to people interested in solving multidimensional problems with big data sets.

Normally, the number of filter coefficients is many fewer than the number of data points, but here we have very many more. Indeed, there are `na` times more. Variable filters require `na` times more memory than the data itself. To make the nonstationary helix code more practical, we now require the filters to be constant in patches. The data type for nonstationary filters (which are constant in patches) is introduced in module `nhelix`, which is a simple modification of module `helix` on page 95.

```
module nhelix {                              # Define nonstationary filter type
  use helix
  type nfilter {                                   #  nd  is data length.
    logical,       dimension(:), pointer :: mis  # (nd) boundary conditions
    integer,       dimension(:), pointer :: pch  # (nd) patches
    type( filter), dimension(:), pointer :: hlx  # (np) filters
  }
contains
  subroutine nallocate( aa, nh, pch) {         # allocate a filter
    type( nfilter)                          :: aa
    integer, dimension(:), intent( in)      :: nh, pch
    integer                                 :: ip, np, nd
    np = size( nh); allocate( aa%hlx( np))
    do ip = 1, np
       call allocatehelix( aa%hlx( ip), nh( ip))
    nd = size( pch); allocate( aa%pch( nd))
    aa%pch = pch
    nullify( aa%mis)                                 # set null pointer for mis.
  }
  subroutine ndeallocate( aa) {                      # destroy a filter
    type( nfilter) :: aa
```

```
    integer        :: ip
    do ip = 1, size( aa%hlx)
            call deallocatehelix( aa%hlx( ip))
    deallocate( aa%hlx, aa%pch)
    if( associated( aa%mis))                    # if logicals were allocated
        deallocate( aa%mis)                     # free them
  }
}
```

What is new is the integer valued vector `pch(nd)` the size of the one-dimensional (helix) output data space. Every filter output point is to be assigned to a patch. All filters of a given patch number will be the same filter. Nonstationary helixes are created with `createnhelixmod`, which is a simple modification of module `createhelixmod` on page 113.

```
module createnhelixmod {      # Create non-stationary helix filter lags and mis
use createhelixmod
use nhelix
use nbound
contains
  function createnhelix( nd, center, gap, na, pch) result (nsaa) {
    type( nfilter)                      :: nsaa   # needed by nhelicon
    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)   :: pch    # (prod(nd)) patching
    type( filter)                       :: aa
    integer                             :: n123, np, ip
    integer, dimension(:), allocatable :: nh
    aa = createhelix( nd, center, gap, na)
    n123 = product( nd);  np = maxval(pch)
    allocate (nh (np)); nh = size (aa%lag)
    call nallocate( nsaa, nh, pch)
    do ip = 1, np
       nsaa%hlx( ip)%lag = aa%lag
    call deallocatehelix (aa)
    call nboundn(1, nd, na, nsaa)
    }
}
```

Notice that the user must define the `pch(product(nd))` vector before creating a nonstationary helix. For a simple 1-D time-variable filter, presumably `pch` would be something like $(1,1,2,2,3,3,\cdots)$. For multidimensional patching we need to think a little more.

Finally, we are ready for the convolution operator. The operator `nhconest` on this page allows for a different filter in each patch.

```
module nhconest {           # Nonstationary convolution, adjoint is the filter.
use nhelix
  real, dimension(:), pointer                :: x
  type( nfilter)                             :: aa
  integer                                    :: nhmax
  integer, private                           :: np
```

```
#% _init( x, aa, nhmax)
    np = size( aa%hlx)
#% _lop( a( nhmax, np), y(:))
    integer                          :: ia, ix, iy, ip
    integer, dimension(:), pointer :: lag
    do iy = 1, size( y) {  if( aa%mis( iy)) cycle
        ip = aa%pch( iy); lag => aa%hlx( ip)%lag
        do ia = 1, size( lag) {
                ix = iy - lag( ia);   if( ix < 1) cycle
                     if( adj)    a( ia, ip) +=  y( iy) * x( ix)
                     else        y( iy) +=  a( ia, ip) * x( ix)
                 }
        }
}
```

A filter output `y(iy)` has its filter from the patch `ip=aa%pch(iy)`. The line `t=a(ip,:)` extracts the filter for the `ip`th patch. If you are confused (as I am) about the difference between `aa` and `a`, maybe now is the time to have a look at beyond Loptran to the Fortran version.[2]

Because of the massive increase in the number of filter coefficients, allowing these many filters takes us from overdetermined to very undetermined. We can estimate all these filter coefficients by the usual deconvolution fitting goal (6.18)

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{YKa} + \mathbf{r}_0 \tag{9.15}$$

but we need to supplement it with some damping goals, say

$$\begin{aligned} \mathbf{0} &\approx \mathbf{YKa} + \mathbf{r}_0 \\ \mathbf{0} &\approx \epsilon\, \mathbf{Ra} \end{aligned} \tag{9.16}$$

where $\mathbf{R}$ is a roughening operator to be chosen.

Experience with missing data in Chapter 3 shows that when the roughening operator $\mathbf{R}$ is a differential operator, the number of iterations can be large. We can speed the calculation immensely by "preconditioning". Define a new variable $\mathbf{m}$ by $\mathbf{a} = \mathbf{R}^{-1}\mathbf{m}$ and insert it into (9.16) to get the equivalent preconditioned system of goals.

$$\mathbf{0} \quad \approx \quad \mathbf{YKR}^{-1}\mathbf{m} \tag{9.17}$$

$$\mathbf{0} \quad \approx \quad \epsilon\, \mathbf{m} \tag{9.18}$$

The fitting (9.17) uses the operator $\mathbf{YKR}^{-1}$. For $\mathbf{Y}$ we can use subroutine `nhconest()` on the preceding page; for the smoothing operator $\mathbf{R}^{-1}$ we can use nonstationary polynomial division with operator `npolydiv()`:

```
module npolydiv {                          # Helix polynomial division
use nhelix
integer                          :: nd
type( nfilter)                   :: aa
```

[2] http://sepwww.stanford.edu/sep/prof/gee/Lib/

```
real, dimension (nd), allocatable :: tt
#%  _init ( nd, aa)
#%  _lop  ( xx, yy)
integer                          :: ia, ix, iy, ip
integer, dimension(:), pointer :: lag
real,    dimension(:), pointer :: flt
tt = 0.
if( adj) {
        tt = yy
        do iy= nd, 1, -1 { ip = aa%pch( iy)
                lag => aa%hlx( ip)%lag; flt => aa%hlx( ip)%flt
                do ia = 1, size( lag) {
                        ix = iy - lag( ia);     if( ix < 1)  cycle
                        tt( ix) -=  flt( ia) * tt( iy)
                        }
                }
        xx += tt
 } else {
        tt = xx
        do iy= 1, nd { ip = aa%pch( iy)
                lag => aa%hlx( ip)%lag; flt => aa%hlx( ip)%flt
                do ia = 1, size( lag) {
                        ix = iy - lag( ia);      if( ix < 1)  cycle
                        tt( iy) -=  flt( ia) * tt( ix)
                        }
                }
        yy += tt
        }
}
```

Now we have all the pieces we need. As we previously estimated stationary filters with the module `pef` on page 157, now we can estimate nonstationary PEFs with the module `npef` on the current page. The steps are hardly any different.

```
module npef {                                       # Estimate non-stationary PEF
  use nhconest
  use npolydiv2
  use cgstep_mod
  use solver_mod
contains
  subroutine find_pef( dd, aa, rr, niter, eps, nh) {
    integer,          intent( in)    :: niter, nh      # number of iterations
    real,             intent( in)    :: eps            # epsilon
    type( nfilter)                   :: aa             # estimated PEF output.
    type( nfilter),   intent( in)    :: rr             # roughening filter.
    real,    dimension(:), pointer   :: dd             # input data
    integer                          :: ip, ih, np, nr # filter lengths
    real, dimension (:), allocatable :: flt            # np*na filter coefs
    np = size( aa%hlx)                  # data   length
    nr = np*nh
    allocate( flt( nr))
    call nhconest_init( dd, aa, nh)
    call npolydiv2_init( nr, rr)
    call solver_prec( nhconest_lop, cgstep, x= flt, dat= -dd, niter= niter,
```

```
                    prec= npolydiv2_lop, nprec= nr, eps= eps)
    call   cgstep_close()
    call npolydiv2_close()
    call nhconest_close()
    do ip = 1, np
       do ih = 1, size( aa%hlx(ip)%lag)
          aa%hlx( ip)%flt( ih) = flt( (ip-1)*nh + ih)
    deallocate( flt)
    }
}
```

Near the end of module `npef` is a filter `reshape` from a 1-D array to a 2-D array. If you find it troublesome that `nhconest` on page 232 was using the filter during the optimization as already multidimensional, perhaps again, it is time to examine the Fortran code. The answer is that there has been a conversion back and forth partially hidden by Loptran.

Figure 9.15 shows a synthetic data example using these programs. As we hope for deconvolution, events are compressed. The compression is fairly good, even though each event has a different spectrum. What is especially pleasing is that satisfactory results are obtained in truly small numbers of iterations (about three). The example is for two free filter coefficients $(1, a_1, a_2)$ per output point. The roughening operator $\mathbf{R}$ was taken to be $(1, -2, 1)$ which was factored into causal and anticausal finite difference.

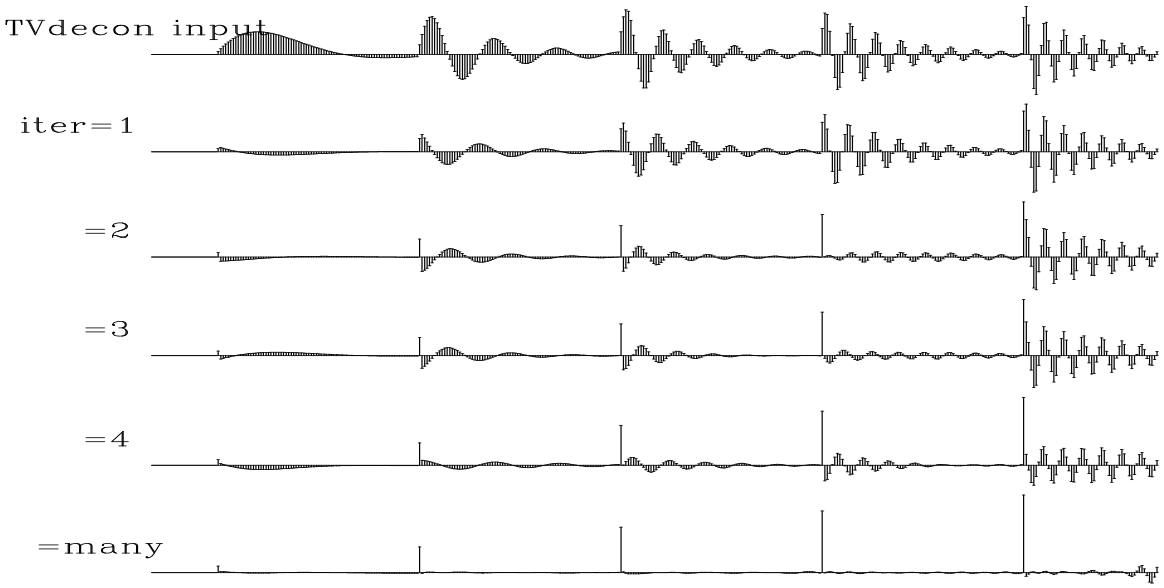

Figure 9.15: Time variable deconvolution with two free filter coefficients and a gap of 6. Four iterations is better than many. pch-tvdecon90 [ER]

I hope also to find a test case with field data, but experience in seismology is that spectral changes are slow, which implies unexciting results. Many interesting examples should exist in two- and three-dimensional filtering, however, because reflector dip is always changing and that changes the *spatial* spectrum.

In multidimensional space, the smoothing filter $\mathbf{R}^{-1}$ can be chosen with interesting directional properties. Sergey, Bob, Sean and I have joked about this code being the "double helix" program because there are two multidimensional helixes in it, one the smoothing filter, the other the deconvolution filter. Unlike the biological helixes, however, these two helixes do not seem to form a symmetrical pair.

EXERCISES:

1 Is `nhconest` on page 232 the inverse operator to `npolydiv` on page 233? Do they commute?

2 Sketch the matrix corresponding to operator `nhconest` on page 232. HINTS: Do not try to write all the matrix elements. Instead draw short lines to indicate rows or columns. As a "warm up" consider a simpler case where one filter is used on the first half of the data and another filter for the other half. Then upgrade that solution from two to about ten filters.

REFERENCES

Shearer, P. M., 1991a, Constraints on upper mantle discontinuities from observations of long period reflected and converted phases: J. Geophys. Res., **96**, no. B11, 18147–18182.

Shearer, P. M., 1991b, Imaging global body wave phases by stacking long-period seismograms: J. Geophys. Res., **96**, no. B12, 20535–20324.

Index

Cecil and Ida Green, 228
createnhelixmod module, 232
Cumro, 221

deconvolution
    steep dip, 219
    time-variable, 231
dip reject, 220

field arrays, 224
filter
    multidip, 227
    time and space variable, 215
    time-variable, 231
Fourier analysis, 225
Fourier space, 226

geophysical inverse theory, 225
Green, Cecil and Ida, 228
ground roll, 224

idempotent, 214
inverse patching, 213
inverse theory, 225

loconvol operator module, 218
lopef module, 218

mkwallwt module, 214
module
    `createnhelixmod`, create non-stationary helix, 232
    `lopef`, local PEF, 218
    `mkwallwt`, make wall weight, 214
    `nhelix`, non-stationary convolution, 231
    `npef`, non-stationary PEF, 234
    `patching`, generic patching, 215
    `tent`, tent weights, 216
movies, 230

multidip filtering, 227

nhconest operator module, 232
nhelix module, 231
nonstationarity, 211
nonstationary, 211
npef module, 234
npolydiv operator module, 233

operator
    `loconvol`, local convolution, 218
    `nhconest`, non-stationary convolution, 232
    `npolydiv`, non-stationary polynomial division, 233
    `patch`, extract patches, 212
    `signoi`, signal and noise separation, 226

patch
    triangular, 219
patch operator module, 212
patching, 211
patching module, 215

Shearer, 228
Shearer-IDA stack, 228
signoi operator module, 226
steep dip decon, 219

tent module, 216
time-variable deconvolution, 231
triangle weighting functions, 217

wall, 212
weighting patches, 213
window, 212

Yilmaz, 221