Chapter 1

Nonstationarity: patching

There are many reasons for cutting data planes or image planes into pieces (patches), operating on the pieces, and then putting them back together again as depicted in Figure 1.1. 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 (inverse covariance matrix) in estimating any earth properties. In statistical estimation theory, the word to describe changing statistical properties is "nonstationary".

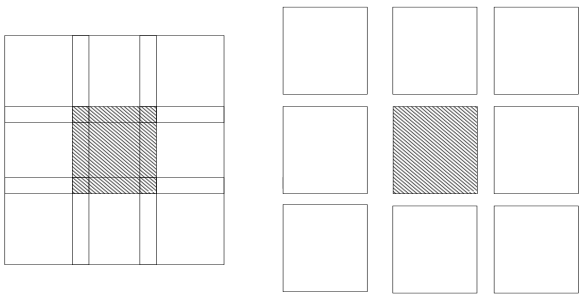

Figure 1.1: Left is space of inputs and outputs. Right is during analysis. pch-rayab2D [NR]

We begin this chapter with basic patching concepts along with supporting utility code. As in chapter ??, I exhibit two-dimensional subroutines only. Three-dimensional code is an easy extension but to reduce clutter, I do not display it. It is in the CD-ROM library. 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.* Then we examine some two-dimensional, time and space variable applications: adaptive steep-dip rejection, noise reduction by prediction, and segregation of signals and noises.

1.1 PATCHING TECHNOLOGY

A plane of information, either data or an image, say `wall(n1,n2)` will be divided up into an array of overlapping **windows** each of size `(w1,w2)`. You choose the number of windows by specifying `(k1,k2)`. The fraction `(w1*k1)/n1` measures overlap on the 1-axis. We can traverse all the windows, grabbing them from the wall, or putting them back, using subroutine `patch2()` (page 2). To run this traverse, we need only looping variables `(j1,j2)` to run from `(1,1)` to `(k1,k2)`.

The intricate indexing statements in `patch2()` (page 2) assure that patches cover all edges and corners of the given data plane even when `n1/w1` and `n2/w2` are not integers. With noninteger ratios, spacing of patches is slightly uneven but you will realize it does not matter after I explain how easy it is to reassemble seamlessly the full plane from the patches.

```
# patch2 ---- copy the (j1,j2)-th of (k1,k2) subplanes from a plane.
#
subroutine patch2( adj, add, j1,j2, k1,k2, wall, n1,n2,  wind, w1,w2)
integer            adj, add, j1,j2, k1,k2,       n1,n2,        w1,w2
integer i1,i2, s1,s2, a1,a2
real                                     wall( n1,n2), wind( w1,w2)
call  adjnull(     adj, add,             wall, n1*n2,  wind, w1*w2)

if( k2 != 1) {  s2 = 1.5 + (n2 - w2) * (j2-1.)/(k2-1.)}    else { s2= 1}
if( k1 != 1) {  s1 = 1.5 + (n1 - w1) * (j1-1.)/(k1-1.)}    else { s1= 1}

do i2= 1, w2 {  a2= i2 + s2 - 1
do i1= 1, w1 {  a1= i1 + s1 - 1
        if( adj == 0 )
                wind( i1,i2) = wind( i1,i2) + wall( a1,a2)
        else
                wall( a1,a2) = wall( a1,a2) + wind( i1,i2)
        }}
return; end
```

The start `(s1,s2)` of the first window on the wall of data is at `(1,1)` and the last start is at the end of the wall `(n1,n2)` less the window width plus one. An exceptional case arises when the user specifies `k1=1` which means the 1-axis contains only one window. Then unless `w1==n1`, the window cannot fit both the beginning and the end of the wall so it is arbitrarily fit only at the beginning. 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.

We should always use subroutine `patch2()` (page 2) for patching and avoid copying any of its messy internals into our application code. Thus when a better method of patching comes along, we have it all in one place and can change it all at once. Figure 1.2 shows an example.

1.1.1 Weighting and reconstructing

The adjoint of extracting all the patches is adding them back. Because of overlaps, the adjoint is not the inverse. In many applications, inverse patching is required,

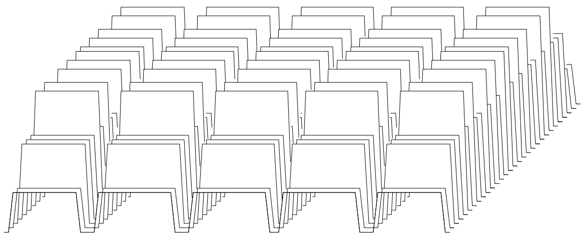

Figure 1.2: A plane of identical values after cutting into patches and then adding all the patches. Results are shown for `n1=100 w1=17 k1=5 n2=30 w2=6 k2=11` . For these parameters, there is gapping on the horizontal axis and overlap on the depth axis. pch-parcel [NR]

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

$$\tilde{\mathbf{d}} \quad = \quad [\mathbf{W}_{\rm wall}\mathbf{P}'\mathbf{W}_{\rm wind}\mathbf{P}]\mathbf{d} \quad = \quad \tilde{\mathbf{I}}_p\,\mathbf{d} \tag{1.1}$$

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). $\mathbf{W}_{\rm wind}$ is your chosen weighting function in the window, and $\mathbf{W}_{\rm wall}$ is the weighting function for the whole wall. You specify any $\mathbf{W}_{\rm wind}$ you like, and subroutine `mkwallwt2()` (page 3) below builds the weighting function $\mathbf{W}_{\rm wall}$ that you need to apply to your wall of reconstructed data to 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 $\mathbf{W}_{\rm wall}$ takes care of it. The method is to add the weights of each window onto the wall using adjoint `patch2()` (page 2), and finally to invert the sum wherever it is non-zero. (You lose data wherever the sum is zero).

```
# 2-D make wall weights from window weights.
#
subroutine mkwallwt2( k1,k2, windwt, w1,w2,  wallwt, n1,n2)
integer    i1,i2,     k1,k2,         w1,w2,          n1,n2
real                          windwt( w1,w2), wallwt( n1,n2)
call null(                                    wallwt, n1*n2)
do i2= 1, k2 {
do i1= 1, k1 {
        call patch2( 1,1, i1,i2, k1,k2,  wallwt, n1,n2,
                                         windwt, w1,w2 )
        }}
do i2= 1, n2 {
do i1= 1, n1 {
        if(  wallwt( i1,i2) != 0.)
             wallwt( i1,i2)  = 1. / wallwt( i1,i2)
        }}
return; end
```

No matrices are needed to show 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. (Zero valued weights would be OK but you would lose data there unless it was picked up by an overlapping window.) The general strategy allows 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 1.3 shows the result 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.

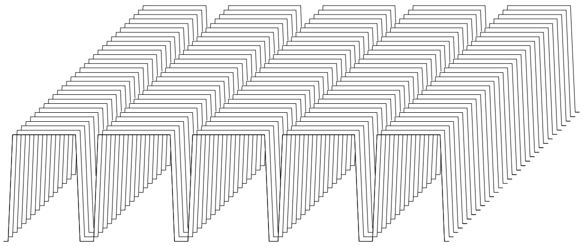

Figure 1.3: A plane of identical values passed through the idempotent patching reconstruction operator. Results are shown for the same parameters as Figure 1.2. pch-idempatch [NR]

1.1.2 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

$$\tilde{\mathbf{d}} \quad = \quad [\mathbf{W}_{\rm wall}\mathbf{P}'\mathbf{W}_{\rm wind}\mathbf{F}_p\mathbf{P}]\ \mathbf{d} \quad = \quad \tilde{\mathbf{F}}\ \mathbf{d} \tag{1.2}$$

A convenient subroutine for two-dimensional filtering is `icaf2()` (page ??). For this no-end-effect convolution routine I built a triangular weighting routine `tent2()` (page 5) that tapers from the center of the patch of the filter's *outputs* towards the edges. This 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 is shown for prediction-error filters in Figure 1.4. 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 `tent2()`. When studying these diagrams and comparing figure 1.4 to internal filtering

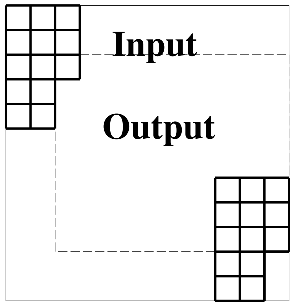

Figure 1.4: Domain of inputs and outputs of a two-dimensional prediction-error filter. pch-rabdomain [NR]

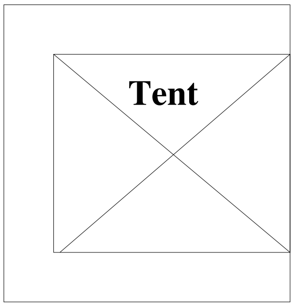

Figure 1.5: Placement of tent-like weighting function in the space of filter inputs and outputs. pch-rabtent [NR]

with subroutine `icaf2()` (page ??), it helps to remember that when the filter index `b1` equals `lag1` then the output 1-axis pointer `y1` matches the input pointer `x1` (and likewise for the 2-axis).

```
# triangle tent weights for 2-D convolution
#
subroutine tent2( a1,a2, lag1,lag2, windwt, w1,w2)
integer           a1,a2, lag1,lag2,         w1,w2
real                                windwt( w1,w2)
integer i1,i2,  s1,s2,  e1,e2
real mid1,mid2, wide1,wide2, x,y
                          call null( windwt, w1*w2)
s1= 1+a1-lag1;  e1= w1-lag1+1;  mid1=(e1+s1)/2.;  wide1=(e1-s1+1.)/2.
s2= 1+a2-lag2;  e2= w2-lag2+1;  mid2=(e2+s2)/2.;  wide2=(e2-s2+1.)/2.
#
do i2= s2, e2 {      y = abs((i2-mid2)/wide2)
do i1= s1, e1 {      x = abs((i1-mid1)/wide1)
# Cheop windwt(i1,i2) = amax1( 0., 2. - abs(x+y) - abs(x-y) )
        windwt(i1,i2) = amax1( 0., 1. - abs(x))  * amax1( 0., 1. - abs(y))
        }}
return; end
```

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 `tent2()` by setting filter dimensions and lags to unity such as shown in Figure 1.6.

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,

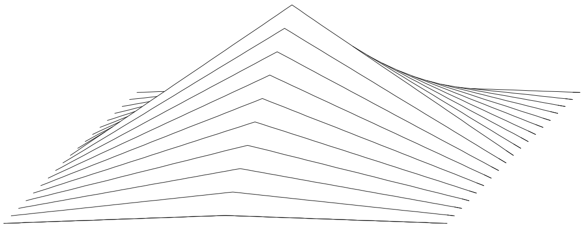

Figure 1.6: Window weights from `tent2()` with `w1=61, w2=19 a1=1, a2=1, lag1=1, lag2=1` . pch-windwt [NR]

$2 - |x - y| + |x + y|$. Instead, this property is possessed by the tent-like shapes in Figure 1.6 with the equation $(1 - |x|)(1 - |y|)$ as is shown in Figure 1.7 where, to add interest, I separated the windows by a little more than the precisely matching distance. In practice we may chose window shapes and overlaps for other reasons than the constancy of the sum of weights because `mkwallwt2()` (page 3) accounts for that.

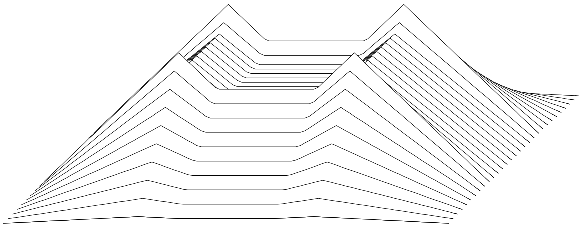

Figure 1.7: (Inverse) wall weights with `n1=100, w1=61, k1=2, n2=30, w2=19, k2=2` pch-wallwt [NR]

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 1.3. Here in Figure 1.8, both axes need more window density.

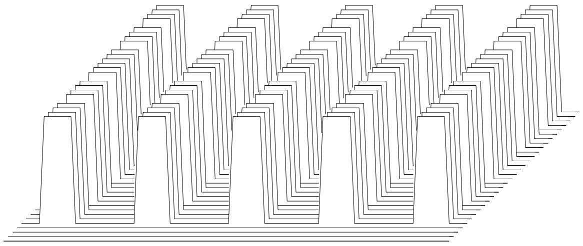

Figure 1.8: Filtering in patches with the same parameters as Figures 1.2 and 1.3. Additionally, the filter parameters are `a1=11 a2=5 lag1=6 lag2=1` . Thus, windows are centered on the 1-axis and pushed back out the 2-axis. pch-cinloip [NR]

1.1.3 Designing a separate filter for each patch

Recall the 2-D prediction-error filter subroutine `pef2()` (page ??). 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. Thus it is natural to apply subroutine `pef2()` in local patches. This is done by subroutine `lopef2()` (page 7). First it builds the tent weights and erases the output residual. Then it loops over all patches. Then it calls `pef2()` (page ??) to get the prediction residual for the patch. Next, the residual is weighted by the window weight using `diag()` (page ??), and the the patches are reassembled. Finally the **wall weight** multiplies the reassembly to account for overlapping patches.

```
# lopef2  ---- LOcal Prediction Error Filter in 2-D
#
subroutine lopef2( data, resi, n1,n2,w1,w2,k1,k2, niter, lag1,lag2, afre, a1,a2)
integer     i1,i2, n1,n2,w1,w2,k1,k2, niter, lag1,lag2,                   a1,a2
real              data(n1,n2),resi(n1,n2),                             afre(a1,a2)
temporary real  wallwt(n1,n2),                                         aa(  a1,a2)
temporary real  pdata(w1,w2), presi(w1,w2),  windwt(w1,w2)
call     tent2(                        a1,a2, lag1,lag2, windwt,w1,w2)
call null(                                               resi,  n1*n2)
do i2= 1, k2 {
do i1= 1, k1 {
        call patch2( 0,0, i1,i2,k1,k2, data,n1,n2,pdata, w1,w2)
        call pef2( pdata,presi, w1,w2, niter,lag1,lag2,afre,aa,a1,a2)
        call diag( 0, 0, windwt, w1*w2,     presi,presi        )
        call patch2( 1,1, i1,i2,k1,k2, resi,n1,n2,presi, w1,w2)
        }}
call mkwallwt2( k1,k2, windwt, w1,w2, wallwt, n1,n2)
call diag  (       0, 0,              wallwt, n1*n2,   resi, resi )
return; end
```

The output of `lopef2()` is necessarily zero around some of its outer periphery because that of `pef2()` is so. The output borders necessarily have some zeros because filter outputs are smaller than the inputs. Where this happens, I usually replace the (zero) output by the input (scaled by the variance ratio).

1.1.4 Triangular patches

I have been running patching code for a few years and my first general comment is that realistic applications often call for patches of different sizes and different shapes. (Tutorial, non-interactve 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, allowing, of course, for overlap the size of the filter. Subroutine `pef2()` (page ??) automatically ignores the zeroed portion of the triangle and it is irrelevant what `mis2()` (page ??) 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 $\mathbf{W}_{\rm wall}\mathbf{P}'\mathbf{W}_{\rm wind}\mathbf{P}$ 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 set window weights to zero where there is no data. Since there will be a different set of weights in each window, you will need to make a simple generalization to `mkwallwt2()` (page 3).

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

1.2 STEEP DIP DECON

Normally an autoregression filter (PEF) predicts a value at a point using 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 regression 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 regression 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 because looking along a trajectory at water velocity slope, water noise is transformed to zero-frequency and hence is easily predicted. It is even better on land where noises move more slowly at irregular velocities, and are more likely to be aliased.

The overall process proceeds independently in each of many overlapping windows using `lopef2()` (page 7). The most important practical aspect is the filter masks, described next.

1.2.1 Dip rejecting known-velocity waves

Consider the two-dimensional filter

$$\begin{array}{ccc} & +1 & \\ -1 & 0 & -1 \\ & +1 & \end{array} \tag{1.3}$$

This filter applied to a field profile with 4 milliseconds time sampling and 6 meter trace spacing should perfectly extinguish 1.5 km/s water-velocity noises. Likewise, the filter

$$
\begin{array}{ccc}
 & +1 & \\
 & 0 & \\
 & 0 & \\
-1 & 0 & -1 \\
 & 0 & \\
 & 0 & \\
 & +1 &
\end{array}
\tag{1.4}
$$

should perfectly extinguish water noise when the trace spacing is 18 meters. 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 the other. So you expect 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:

$$
\begin{array}{ccccccccc}
x & x & x & x & x & x & x & x & x \\
x & x & x & x & x & x & x & x & x \\
. & x & x & x & x & x & x & x & . \\
. & x & x & x & x & x & x & x & . \\
. & x & x & x & x & x & x & x & . \\
. & . & x & x & x & x & x & . & . \\
. & . & x & x & x & x & x & . & . \\
. & . & x & x & x & x & x & . & . \\
. & . & . & x & x & x & . & . & . \\
. & . & . & x & x & x & . & . & . \\
. & . & . & x & x & x & . & . & . \\
. & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . \\
. & . & . & . & 1 & . & . & . & .
\end{array}
\tag{1.5}
$$

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 diamond instead of triangle. I invite you to experiment with the various shapes that suggest themselves.

1.2.2 Field data tests

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 1.9.

Figure 1.9: Gravel plain ground roll (Middle East) Worth testing. pch-gravel2D [NR]

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). Since 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 I 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 1.10 is a sample of them.

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 with

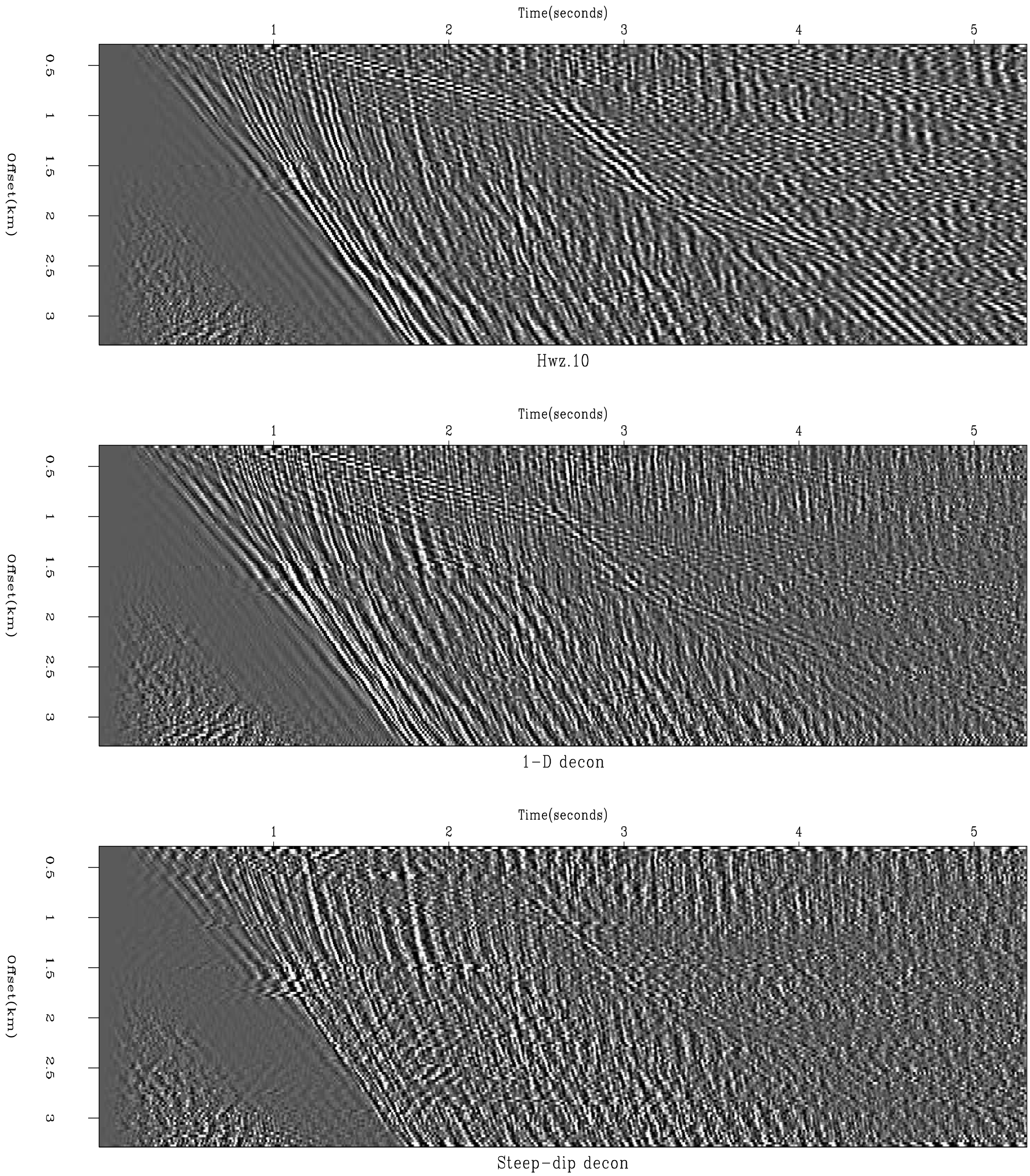

Figure 1.10: Bottom is a North African vibrator shot profile (Y&C #10) after AGC. Middle is gapped 1-D decon. Top is steep-dip decon. pch-wz.10 [NR]

spatially one-sided filters like

$$
\begin{array}{ccccc}
x & x & x & x & x \\
. & x & x & x & x \\
. & x & x & x & x \\
. & . & x & x & x \\
. & . & x & x & x \\
. & . & . & x & x \\
. & . & . & x & x \\
. & . & . & . & . \\
. & . & . & . & . \\
. & . & . & . & 1
\end{array}
\tag{1.6}
$$

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

1.2.3 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, such as certain 3-D geometries. A recently arrived 3-D shot profile from the sand dunes in the Middle East is Figure 1.11. 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?

1.2.4 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 it is really helpful. Also, it would, of course, be fun

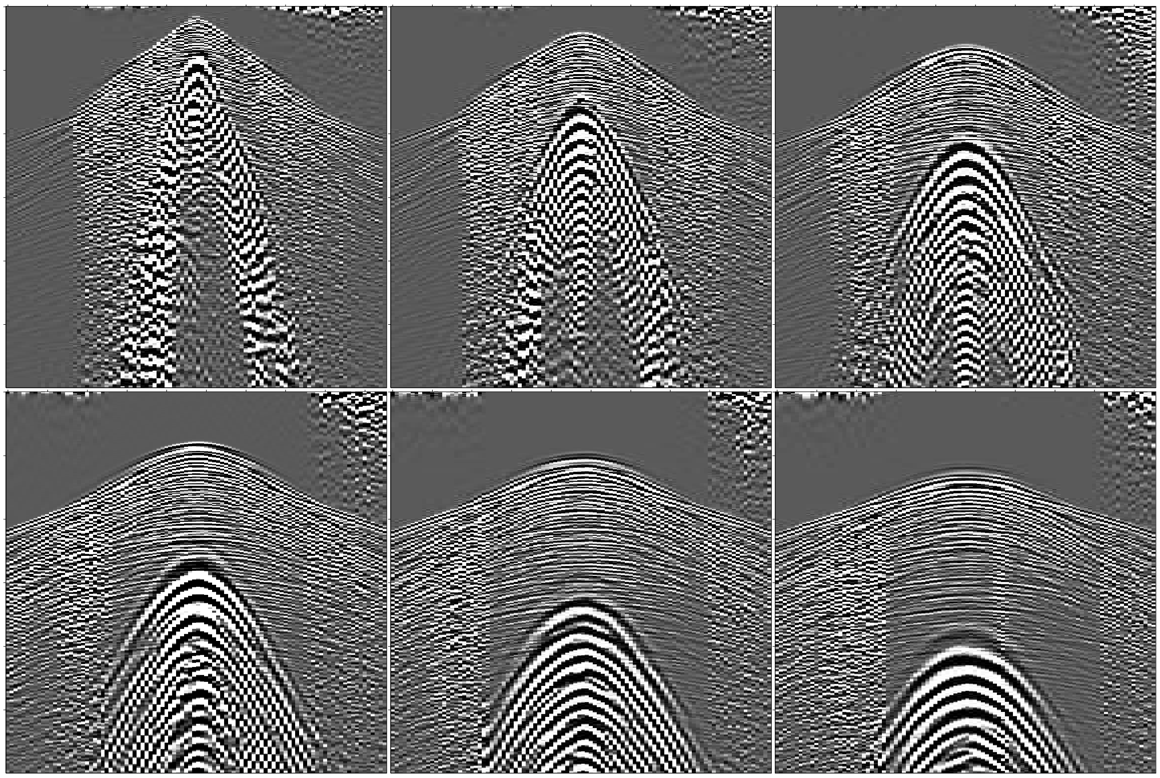

Figure 1.11: Sand dunes. One shot, six parallel receiver lines. pch-dune3D [NR]

to get some complete data sets (rather than a single shot profile) to see the difference in the final stack.

1.3 SIGNAL ENHANCEMENT BY PREDICTION

In historic exploration industry use, prediction-error filtering provides temporal predictions which are immediately subtracted from the data itself. In recent years, Luis **Canales** proposed and developed a process of looking at the *spatial* predictions themselves and this process has become quite popular. The idea is that since noise is unpredictable, better looking seismic data can result from looking at the spatial predictions than looking at the data itself. Although Canales' process is done in the temporal frequency domain, we can also do it in the time domain where we can maintain tighter control over nonstationarity and statistical fluctuations. The prediction-error filter form is

$$\begin{array}{ccccccc} a & a & a & \cdot & \cdot & \cdot & \cdot \\ a & a & a & \cdot & \cdot & \cdot & \cdot \\ a & a & a & 1 & \cdot & \cdot & \cdot \\ a & a & a & \cdot & \cdot & \cdot & \cdot \\ a & a & a & \cdot & \cdot & \cdot & \cdot \end{array} \tag{1.7}$$

and the prediction is the same without the "1". It is perplexing that the spatial prediction has a horizontal direction. Some people average the left and the right, but here I have not. An alternative is to use interpolation,

$$\begin{array}{ccccccc} a & a & a & \cdot & a & a & a \\ a & a & a & \cdot & a & a & a \\ a & a & a & 1 & a & a & a \\ a & a & a & \cdot & a & a & a \\ a & a & a & \cdot & a & a & a \end{array} \tag{1.8}$$

but here I have not. In either case, it is important to realize that after the filter coefficients are determined by minimizing output power, the "1" in the filter is replaced by zero before it is used. Thus the methods are prediction or interpolation and they should not be called "deconvolution".

I selected the interesting data set shown in Figure 1.12 to compare the spatial predictions to the data itself. The data plane is a stack of **earthquake**s. At early times, before 93 minutes travel time, the data resembles a common-midpoint gather. At later times the strong surface waves travel round the earth and past the antipodes and come back towards the source. Otherwise, there are remarkable similarities to conventional exploration seismic data. Many fewer earthquakes are observed near 180 degrees than near 90 degrees for the simple geometrical reason that ten degrees surrounding the equator is a much bigger area than ten degrees surrounding the pole. Thus the quality of the stacks degrades rapidly toward the poles. Although data quality is poor at the poles themselves, notice that waves going beyond the antipodes come back toward the source. The data has a large dynamic range that I compressed by various range and time dependent gain multipliers and in the last step before display, I took the signed square roots of the values of the stack.

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.

1.3.1 Parameters for signal enhancement by prediction

The predictions in Figure 1.12 were derived from prediction errors computed from subroutine `lopef2()` (page 7), seen earlier in another application. The prediction is

[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 (?) (?) 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`.

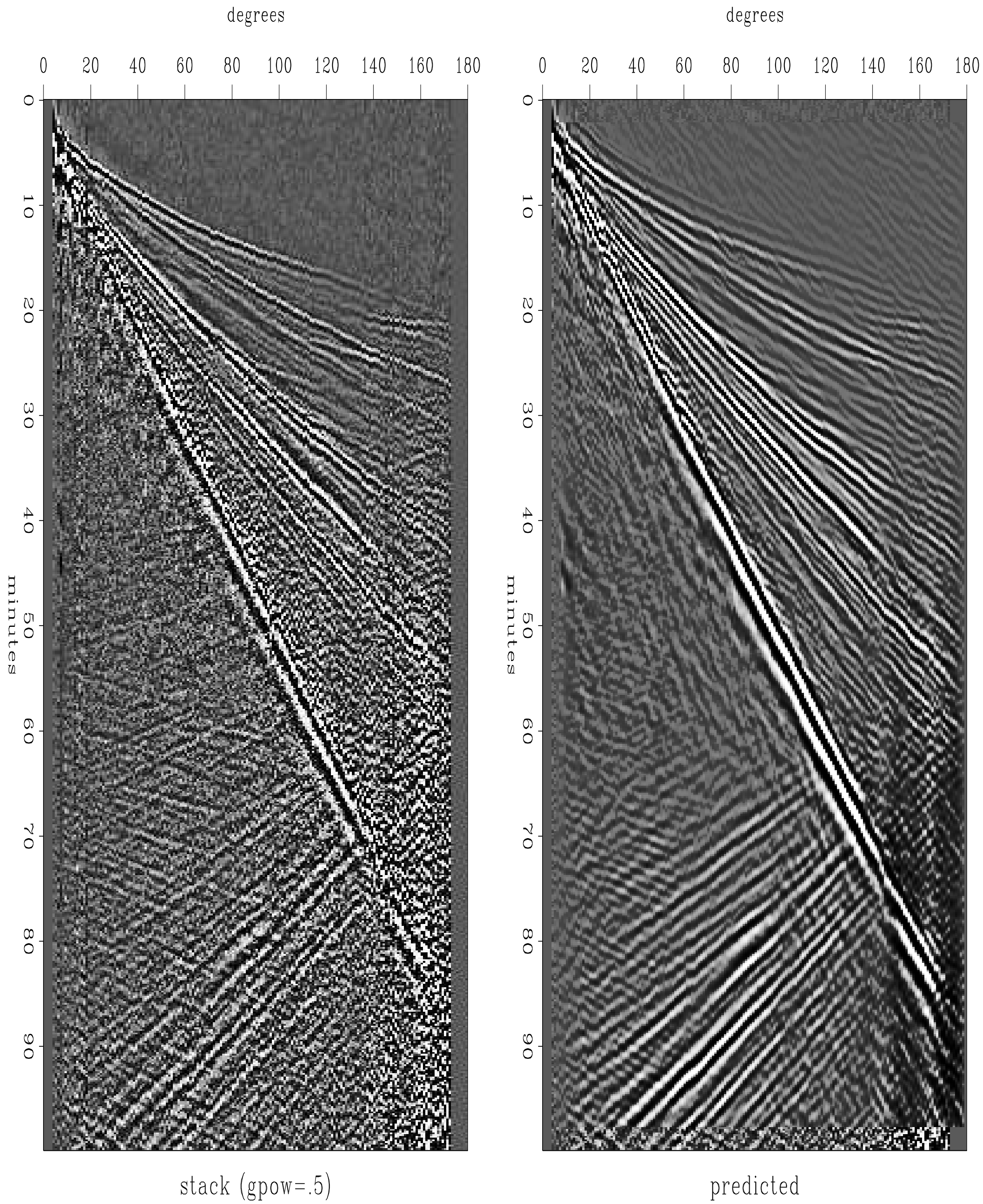

Figure 1.12: Stack of Shearer's IDA data (left). Prediction (right). Notice that the time scale is minutes and the offset is degrees of angle on the earth's surface. pch-idapred [NR]

simply the data minus the prediction error.

Data is analyzed in many overlapping windows which are then merged. Since the quality of results depends on the window sizes, I report here the reasoning behind my choices. First, the Canales process is generally applied in the temporal frequency domain. The number of coefficients on the space axis for the predictions is generally taken much larger than the wave slope count in a typical window. This is common practice and I explain the larger size by saying that since the prediction of the data is based on noisy data itself, it needs a sizeable window in which to do statistical averaging.

To match the stepout of the dominant wave (an around world **Rayleigh wave**) I took the filter length and width to be `a1=27` and `a2=7`. Then for statistical smoothing I chose fitting windows to be ten times as large as the filter in both directions. Obviously, the statistics could be gathered in different amounts on the two axes and that could give significantly different results. Anyway, the result for my choices is that the entire page is divided into four windows horizontally and three vertically.

The temporal (half) extent of the filter is evident by the strong character change at the top and bottom. The spatial extent is not revealed in this way because of the vanishing traces (empty bins) along the edges.

I notice a disturbing darkness at late times and wide offsets. This is energy at zero frequency, a highly predictable frequency, that might have crept in because I used medians on bins with small numbers of traces, perhaps an *even* number so the median had a consistent bias.

Overall, the prediction process performs as expected. It is disappointing, however, in that it tends to swamp weak events in the "side lobes" of strong events. I believe the widespread acceptance of this process arises from its use on data of very low quality. Where there is barely one perceptible event, a process that strengthens that event is a welcome process.

1.4 SIGNAL-NOISE DECOMPOSITION BY DIP

A common form of linearized geophysical inverse theory is

$$\mathbf{0} \approx \mathbf{W}(\mathbf{Lm} - \mathbf{d}) \tag{1.9}$$

$$\mathbf{0} \approx \epsilon\mathbf{Am} \tag{1.10}$$

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 operations $\mathbf{W} = \mathbf{N}$ on the noise and $\mathbf{A} = \mathbf{S}$ on the signal. Thus the usual model-fitting regression becomes a regression for signal-noise separation:

$$0 \approx \mathbf{N}(-\mathbf{n}) = \mathbf{N}(\mathbf{s} - \mathbf{d}) \tag{1.11}$$

$$0 \approx \epsilon\mathbf{Ss} \tag{1.12}$$

Let noise $\mathbf{n}$ be energy that has no spatial correlation and signal $\mathbf{s}$ 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. Since we regard the noise as spatially uncorrelated, $\mathbf{N}$ has only coefficients 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 we deem to be locally present. In the examples here I take the number of slopes 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 regression (1.12) but without it the regression (1.11) would simply set the signal $\mathbf{s}$ equal 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 regressions (1.12) and (1.11) amounts to minimizing a quadratic form by setting to zero its derivative, say

$$0 \quad = \quad \frac{\partial}{\partial \mathbf{s}'}\left((\mathbf{s}' - \mathbf{d}')\mathbf{N}'\mathbf{N}(\mathbf{s} - \mathbf{d}) + \epsilon^2 \mathbf{s}'\mathbf{S}'\mathbf{S}\mathbf{s} \right) \tag{1.13}$$

which gives the answer

$$\mathbf{s} \quad = \quad \left(\frac{\mathbf{N}'\mathbf{N}}{\mathbf{N}'\mathbf{N} \; + \; \epsilon^2 \mathbf{S}'\mathbf{S}} \right) \mathbf{d} \tag{1.14}$$

$$\mathbf{n} \quad = \quad \mathbf{d} - \mathbf{s} \quad = \quad \left(\frac{\epsilon^2 \mathbf{S}'\mathbf{S}}{\mathbf{N}'\mathbf{N} \; + \; \epsilon^2 \mathbf{S}'\mathbf{S}} \right) \mathbf{d} \tag{1.15}$$

To make this really concrete, consider its meaning in one dimension where signal is white $\mathbf{S'S} = 1$ and noise has the frequency ω_0 which is killable with the multiplier $\mathbf{N'N} = (\omega - \omega_0)^2$. Now we recognize equation (1.14) is a notch filter and equation (1.15) as a narrow-band filter.

The analytic solutions in equations (1.14) and (1.15) are valid for 2-D equations too in 2-D **Fourier** space, where you could compute them. 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 regression in the form needed in computation.

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \quad \approx \quad \begin{bmatrix} \mathbf{N} \\ \epsilon\mathbf{S} \end{bmatrix} \mathbf{s} \; + \; \begin{bmatrix} -\mathbf{Nd} \\ \mathbf{0} \end{bmatrix} \tag{1.16}$$

Subroutine `signoi2()` (page 18) first computes the prediction-error filters $\mathbf{S}$ and $\mathbf{N}$ using `pef2()` (page ??). Then it loads the residual vector $\mathbf{r}$ with the negative of the noise whitened data $-\mathbf{Nd}$. Then it enters the usual conjugate-gradient iteration loop making adjustments of $\Delta\mathbf{r}$ and $\Delta\mathbf{s}$. Finally, it defines noise by $\mathbf{n} = \mathbf{d} - \mathbf{s}$.

```
# Decompose data into signal and noise, d=s+n given PEF size (s1,s2) of signal.
#
subroutine signoi2( d, s, n,d1,d2,  epsilon, niter, s1,s2)
integer                   d1,d2,           niter, s1,s2, n1,n2, ls1,ln1
real d(d1*d2), s( d1*d2), n(d1*d2), epsilon
integer                                   i, iter, d12, r12, rs,rn
temporary real                rr( d1*d2*2),   ss(s1,s2),    nn(s1,1)
temporary real      ds( d1,d2), dr( d1*d2*2), sfre(s1,s2),  nfre(s1,1)
temporary real      ps( d1,d2), pr( d1*d2*2)
                   d12= d1*d2;  r12=d1*d2;      rs=1;     rn= 1 + r12
ls1= 1+s1/2;    n1=ls1;    n2=1;    ln1=1
call setfree(ls1,1,1,sfre,s1,s2,1,i); call pef2(s,rr,d1,d2,i,ls1,1,sfre,ss,s1,s2)
call setfree(ln1,1,1,nfre,n1,n2,1,i); call pef2(n,rr,d1,d2,i,ln1,1,nfre,nn,n1,n2)

                                         call scale( epsilon, s1*s2, ss)
call null( s, d12); call null( rr, 2* r12)
call         icai2(  0,0,1,1, nn,n1,n2,           d, d1,d2,  rr(rn ))
call         scale( -1.,                             d1*d2,  rr(rn ))
do iter= 0, niter {
        call icai2(  1,0,1,1, nn,n1,n2,           ds,d1,d2,  rr(rn ))
        call icai2(  1,1,1,1,           ss,s1,s2, ds,d1,d2,  rr(rs ))
        call icai2(  0,0,1,1, nn,n1,n2,           ds,d1,d2,  dr(rn ))
        call icai2(  0,0,1,1,           ss,s1,s2, ds,d1,d2,  dr(rs ))
        call cgplus( iter,   d12, s,ds,ps,
                           2*r12, rr,dr,pr )
        }
do i=1, d12 { n(i) = d(i) - s(i) }
return; end
```

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`.[2]

[2] I am reminded that these solutions are slow compared to recursive solutions in 1-D. Since the stability of the 2-D recursions is not assured, however, we have no immediate alternative. This suggests that if we could find some way to stabilize the recursions, however crude, it might lead to a huge speedup iteration by conjugate *directions*.

Subroutine `signoi2()` (page 18) is set up in a bootstrapping way. It uses not only data, but also signal and noise. The first time in, I use a crude estimate of the signal and noise that is merely a copy of the raw data. Notice that these copies are used only to compute **S** and **N**, the signal and noise **whiteners**. Since the program exits giving you improved values of signal and noise, you might consider a second invocation of the program. I tried multiple invocations to little avail. Another thing for contemplation is some automatic way of choosing ϵ.

1.4.1 2-D examples of multidip filtering

Figure 1.13 demonstrates the concept. I consider the result satisfactory though I was disappointed that the separation was not "perfect", since the input data is perfect before the noise is added. I was particularly disappointed that the weaker of the two dips is barely recovered but I believe the code is working according to the theory and results of about this quality are to be expected.

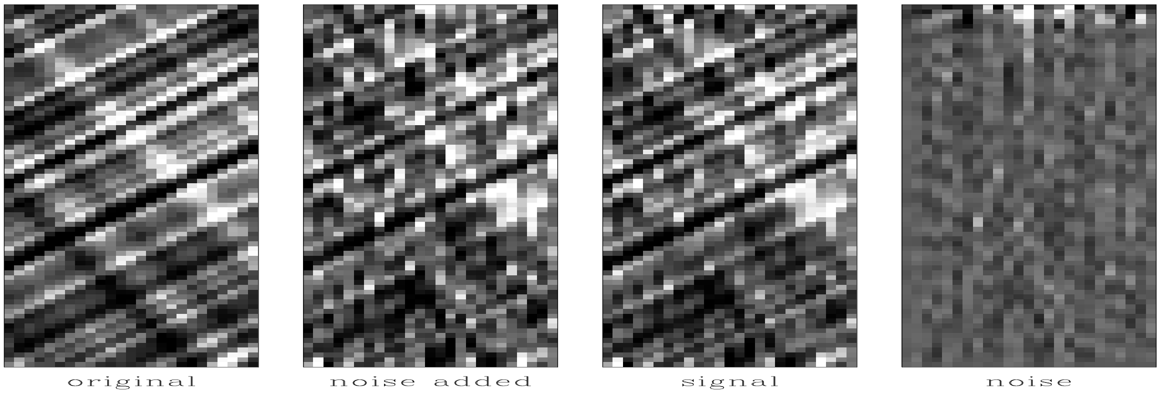

Figure 1.13: The input signal is on the left. Next is that signal with noise added. Next, for my favorite value of `epsilon=1.`, is the estimated signal and the estimated noise. pch-signoi [NR]

Various problems occur around the boundaries. In an effort to avoid these problems, I experimented with different edge conditions in the convolution. Best results were found with transient edge conditions when the problem was recast with the independent variable being **n** instead of **s**

Ray Abma first noticed that different results were obtained when the regression was cast in terms of **n** instead of **s**. At first I could not believe his result, but after repeating the computations independently I had to agree that the result does depend on the choice of independent variable. I sought an explanation in terms of differing null spaces but this is not yet satisfactory.

Figure 1.14 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.

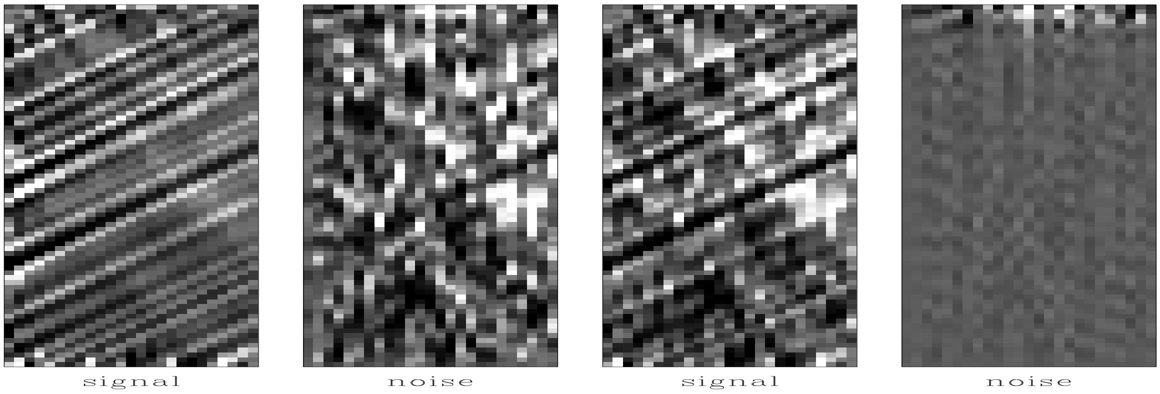

Figure 1.14: Left is an estimated signal-noise pair where `epsilon=4` has caused coherent events to be pushed into the noise. Right is a signal-noise pair where `epsilon=.5`, has caused the noise to be small and the signal to be almost the original data. pch-signeps [NR]

Figure 1.15 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.

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." Keeping the same aspect ratio `a1/a2`, 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.

1.4.2 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 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

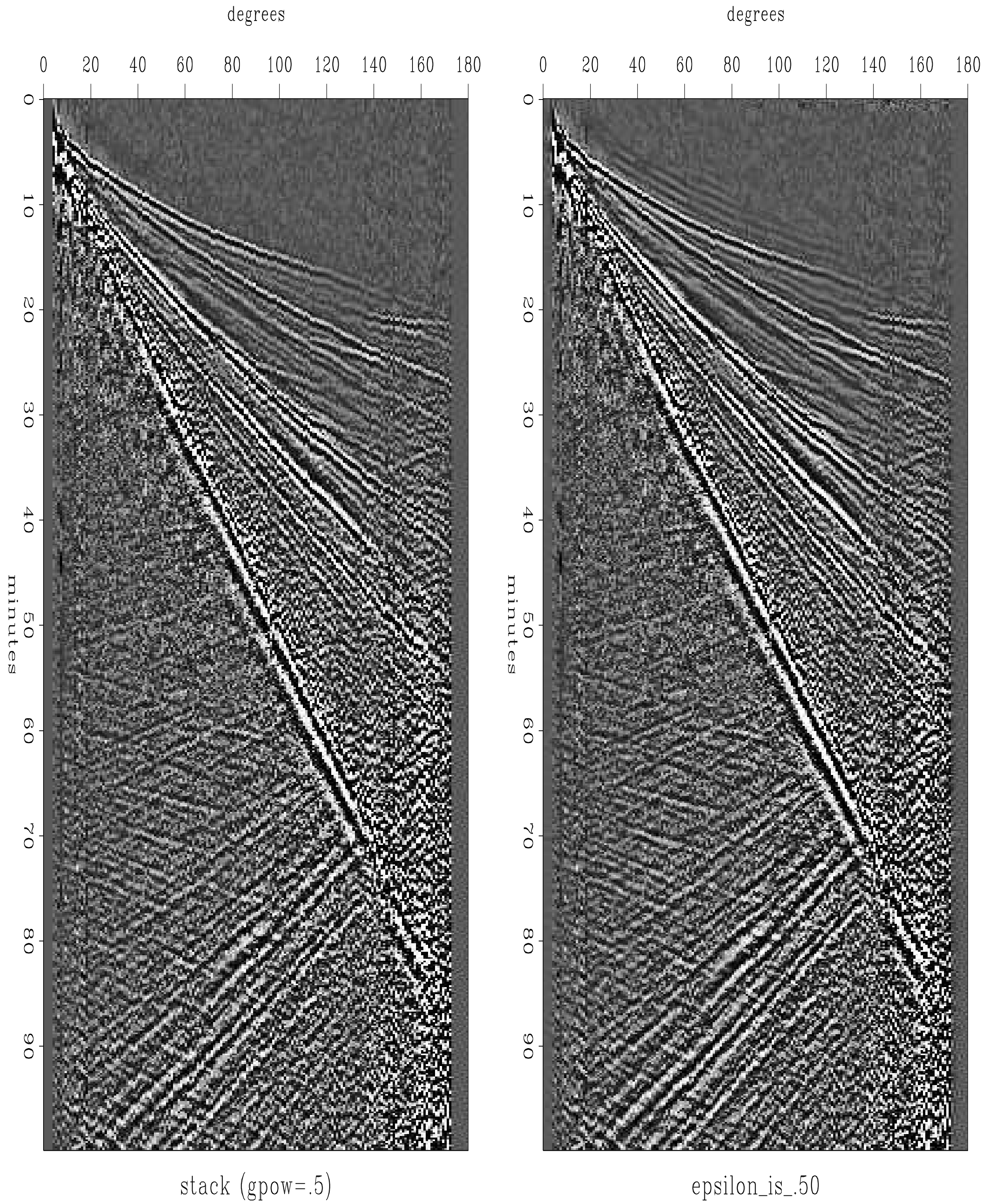

Figure 1.15: Stack of Shearer's IDA data (left). Multidip filtered (right). It is pleasing that the noise is reduced while weak events are preserved. pch-sneps+.50 [NR]

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 1.15. 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 1.15.

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 1.15. But Figure 1.15 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 by turning your head along with the motion.

> If the purpose of data processing is to collapses 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.

Subroutine `losn2()` (page 22) does the "patching", invokes the signal partitioning subroutine `signoi2()` (page 18) on each patch, applies a tent-like weighting function, and reassembles the patches compensating for their overlap. The tents are easier to make here than they were in local prediction error (`lopef2()` (page 7)) because they cover the space of filter *inputs* instead of *outputs*. (In early versions of this code I used internal convolution instead of transient convolution, and it left me confused about the proper shape of the output data space.)

```
# losn  ---- LOcal Signal Noise separation in 2-D
#
```

```
subroutine losn2(  data, sign, n1,n2,w1,w2,k1,k2, epsilon, miter, a1,a2)
integer   i1,i2,              n1,n2,w1,w2,k1,k2,          miter, a1,a2
real        data(n1,n2),sign( n1,n2),           epsilon
temporary real wallwt(n1,n2),                           aa( a1,a2)
temporary real  pdata(w1,w2), psign(w1,w2), pnoiz(w1,w2), windwt(w1,w2)
call     tent2(                            1,1,    1,1,    windwt,w1,w2)
call null(              sign, n1*n2)
do i2= 1, k2 {
do i1= 1, k1 {
        call patch2( 0,0, i1,i2,k1,k2, data,n1,n2, pdata, w1,w2)
        call copy(                         w1*w2, pdata, pnoiz)
        call copy(                         w1*w2, pdata, psign)

        call signoi2( pdata, psign, pnoiz,  w1,w2, epsilon, miter, a1,a2)

        call diag( 0, 0, windwt, w1*w2,     psign,psign        )
        call patch2( 1,1, i1,i2,k1,k2, sign,n1,n2,psign, w1,w2)
        }}
call mkwallwt2( k1,k2, windwt, w1,w2, wallwt, n1,n2)
call diag  (       0, 0,              wallwt, n1*n2,   sign, sign )
return; end
```

1.4.3 Theory for noise along with missing data

Data $\mathbf{d}$ space can be decomposed into known plus missing parts $\mathbf{d} = \mathbf{k} + \mathbf{m}$. We partition an identity operator $\mathbf{I}$ on the data space into parts that separate the known from the missing data $\mathbf{I} = \mathbf{K} + \mathbf{M}$. Thus data space can be written as

$$\mathbf{d} \quad = \quad \mathbf{Kd} + \mathbf{Mm} \tag{1.17}$$

where $\mathbf{Kd}$ is zero-padded known data and all the components of $\mathbf{m}$ are freely adjustable.

Data space $\mathbf{d}$ can also be decomposed into signal plus noise $\mathbf{d} = \mathbf{s} + \mathbf{n}$. Thus

$$\mathbf{s} + \mathbf{n} \quad = \quad \mathbf{Kd} + \mathbf{Mm} \tag{1.18}$$

Writing regressions for signal and noise and then eliminating the noise by the constraint equation (??) gives

$$\mathbf{0} \;\approx\; \mathbf{Nn} \quad = \quad \mathbf{N}(\mathbf{Kd} + \mathbf{Mm} - \mathbf{s}) \tag{1.19}$$

$$\mathbf{0} \;\approx\; \mathbf{Ss} \quad = \quad \mathbf{Ss} \tag{1.20}$$

Putting this in matrix form we have the operator needed in computation

$$\begin{bmatrix} \mathbf{0} \\ \mathbf{0} \end{bmatrix} \quad \approx \quad \begin{bmatrix} -\mathbf{N} & \mathbf{NM} \\ \mathbf{S} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{s} \\ \mathbf{m} \end{bmatrix} + \begin{bmatrix} \mathbf{NKd} \\ \mathbf{0} \end{bmatrix} \tag{1.21}$$

I have not had time to prepare an example.