Chapter 1

Moveout, velocity, and stacking

In this chapter we handle data as though the earth had no dipping reflectors. The earth model is one of stratified layers with velocity a (generally increasing) function of depth. We consider reflections from layers, which we process by normal moveout correction (NMO). The NMO operation is an interesting example of many general principles of linear operators and numerical analysis. Finally, using NMO, we estimate the earth's velocity with depth and we stack some data, getting a picture of an earth with dipping layers. This irony, that techniques developed for a stratified earth can give reasonable images of non-stratified reflectors, is one of the "lucky breaks" of seismic processing. We will explore the limitations of this phenomenon in the chapter on dip-moveout.

First, a few words about informal language. The inverse to velocity arises more frequently in seismology than the velocity itself. This inverse is called the "slowness." In common speech, however, the word "velocity" is a catch-all, so what is called a "velocity analysis" might actually be a plane of slowness versus time.

1.1 INTERPOLATION AS A MATRIX

Here we see how general principles of linear operators are exemplified by linear interpolation. Because the subject matter is so simple and intuitive, it is ideal to exemplify abstract mathematical concepts that apply to all linear operators.

Let an integer k range along a survey line, and let data values x_k be packed into a vector $\mathbf{x}$. (Each data point x_k could also be a seismogram.) Next we resample the data more densely, say from 4 to 6 points. For illustration, I follow a crude **nearest-neighbor interpolation** scheme by sprinkling ones along the diagonal of a rectangular matrix that is

$$\mathbf{y} \quad = \quad \mathbf{B}\,\mathbf{x} \tag{1.1}$$

where

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \tag{1.2}$$

The interpolated data is simply $\mathbf{y} = (x_1, x_2, x_2, x_3, x_4, x_4)$. The matrix multiplication (1.2) would not be done in practice. Instead there would be a loop running over the space of the outputs $\mathbf{y}$ that picked up values from the input.

1.1.1 Looping over input space

The obvious way to program a deformation is to take each point from the *input* space and find where it goes on the output space. Naturally, many points could land in the same place, and then only the last would be seen. Alternately, we could first erase the output space, then add in points, and finally divide by the number of points that ended up in each place. The biggest aggravation is that some places could end up with no points. This happens where the transformation **stretch**es. There we need to decide whether to interpolate the missing points, or simply low-pass filter the output.

1.1.2 Looping over output space

The alternate method that is usually preferable to looping over input space is that our program have a loop over the space of the *outputs,* and that each output find its input. The matrix multiply of (1.2) can be interpreted this way. Where the transformation **shrink**s is a small problem. In that area many points in the input space are ignored, where perhaps they should somehow be averaged with their neighbors. This is not a serious problem unless we are contemplating iterative transformations back and forth between the spaces.

We will now address interesting questions about the reversibility of these deformation transforms.

1.1.3 Formal inversion

We have thought of equation (1.1) as a formula for finding $\mathbf{y}$ from $\mathbf{x}$. Now consider the opposite problem, finding $\mathbf{x}$ from $\mathbf{y}$. Begin by multiplying equation (1.2) by the **transpose matrix** to define a new quantity $\tilde{\mathbf{x}}$:

$$\begin{bmatrix} \tilde{x}_1 \\ \tilde{x}_2 \\ \tilde{x}_3 \\ \tilde{x}_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \tag{1.3}$$

$\tilde{\mathbf{x}}$ is not the same as $\mathbf{x}$, but these two vectors have the same dimensionality and in many applications it may happen that $\tilde{\mathbf{x}}$ is a good approximation to $\mathbf{x}$. In general, $\tilde{\mathbf{x}}$ may be called an "image" of $\mathbf{x}$. Finding the image is the first step of finding $\mathbf{x}$ itself. Formally, the problem is

$$\mathbf{y} \quad = \quad \mathbf{B}\mathbf{x} \tag{1.4}$$

And the formal solution to the problem is

$$\mathbf{x} \quad = \quad (\mathbf{B}'\mathbf{B})^{-1}\,\mathbf{B}'\mathbf{y} \tag{1.5}$$

Formally, we verify this solution by substituting (1.4) into (1.5).

$$\mathbf{x} \quad = \quad (\mathbf{B}'\mathbf{B})^{-1}\,(\mathbf{B}'\mathbf{B})\mathbf{x} \quad = \quad \mathbf{I}\mathbf{x} \quad = \quad \mathbf{x} \tag{1.6}$$

In applications, the possible nonexistence of an inverse for the matrix $(\mathbf{B}'\mathbf{B})$ is always a topic for discussion. For now we simply examine this matrix for the interpolation problem. We see that it is diagonal:

$$\mathbf{B}'\mathbf{B} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad = \quad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} \tag{1.7}$$

So, $\tilde{\mathbf{x}}_1 = \mathbf{x}_1$; but $\tilde{\mathbf{x}}_2 = 2\mathbf{x}_2$. To recover the original data, we need to divide $\tilde{\mathbf{x}}$ by the diagonal matrix $\mathbf{B}'\mathbf{B}$. Thus, matrix inversion is easy here.

Equation (1.5) has an illustrious reputation, which arises in the context of "least squares." **Least squares** is a general method for solving sets of equations that have more equations than unknowns.

Recovering $\mathbf{x}$ from $\mathbf{y}$ using equation (1.5) presumes the existence of the inverse of $\mathbf{B}'\mathbf{B}$. As you might expect, this matrix is nonsingular when $\mathbf{B}$ *stretches* the data, because then a few data values are distributed among a greater number of locations. Where the transformation *squeezes* the data, $\mathbf{B}'\mathbf{B}$ must become singular, since returning uniquely to the uncompressed condition is impossible.

We can now understand why an adjoint operator is often an approximate inverse. This equivalency happens in proportion to the nearness of the matrix $\mathbf{B}'\mathbf{B}$ to an identity matrix. The interpolation example we have just examined is one in which $\mathbf{B}'\mathbf{B}$ differs from an identity matrix merely by a scaling.

1.2 THE NORMAL MOVEOUT MAPPING

Recall the traveltime equation (**??**).

$$v^2 t^2 = z^2 + x^2 \tag{1.8}$$

$$t^2 = \tau^2 + \frac{x^2}{v^2} \tag{1.9}$$

where τ is traveltime depth. This equation gives either time from a surface source to a receiver at depth τ, or it gives time to a surface receiver from an image source at depth τ.

A seismic **trace** is a signal $d(t)$ recorded at some constant x. We can convert the trace to a "vertical propagation" signal $m(\tau) = d(t)$ by stretching t to τ. This process is called "**normal moveout** correction" (NMO). Typically we have many traces at different x distances each of which theoretically produces the same hypothetical zero-offset trace. Figure 1.1 shows a marine shot profile before and after NMO correction at the water velocity. You can notice that the wave packet reflected from the ocean bottom is approximately a constant width on the raw data. After NMO, however, this waveform broadens considerably—a phenomenon known as "NMO stretch."

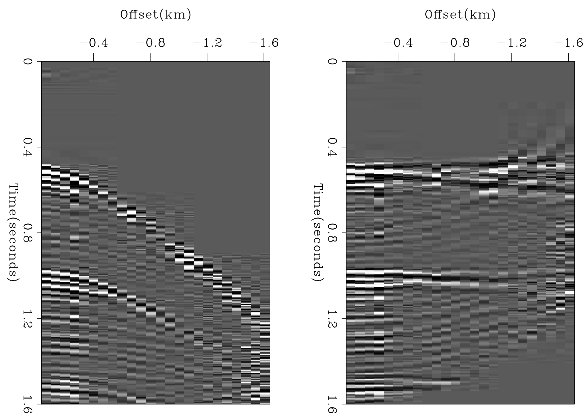

Figure 1.1: Marine data moved out with water velocity. Input on the left, output on the right. Press button for **movie** sweeping through velocity (actually through slowness squared). vela-stretch [ER,M]

The NMO transformation $\mathbf{N}$ is representable as a square matrix. The matrix $\mathbf{N}$ is a (τ,t)-plane containing all zeros except an interpolation operator centered along the hyperbola. The dots in the matrix below are zeros. The input signal d_t is put into the vector $\mathbf{d}$. The output vector $\mathbf{m}$—i.e., the NMO'ed signal—is simply $(d_6, d_6, d_6, d_7, d_7, d_8, d_8, d_9, d_{10}, 0)$. In real life examples such as Figure 1.1 the subscript goes up to about one thousand instead of merely to ten.

$$
\mathbf{m} \quad = \quad \mathbf{N}\mathbf{d} \quad = \quad
\begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \\ m_7 \\ m_8 \\ m_9 \\ m_{10} \end{bmatrix}
\quad = \quad
\begin{bmatrix}
. & . & . & . & . & 1 & . & . & . & . \\
. & . & . & . & . & 1 & . & . & . & . \\
. & . & . & . & . & 1 & . & . & . & . \\
. & . & . & . & . & . & 1 & . & . & . \\
. & . & . & . & . & . & 1 & . & . & . \\
. & . & . & . & . & . & . & 1 & . & . \\
. & . & . & . & . & . & . & 1 & . & . \\
. & . & . & . & . & . & . & . & 1 & . \\
. & . & . & . & . & . & . & . & . & 1 \\
. & . & . & . & . & . & . & . & . & .
\end{bmatrix}
\begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \\ d_9 \\ d_{10} \end{bmatrix}
\tag{1.10}
$$

You can think of the matrix as having a horizontal t-axis and a vertical τ-axis. The 1's in the matrix are arranged on the hyperbola $t^2 = \tau^2 + x_0^2/v^2$. The transpose matrix defining some $\tilde{\mathbf{d}}$ from $\mathbf{m}$ gives synthetic data $\tilde{\mathbf{d}}$ from the zero-offset (or stack) model $\mathbf{m}$, namely,

$$
\tilde{\mathbf{d}} \quad = \quad \mathbf{N}'\mathbf{m} \quad = \quad
\begin{bmatrix} \tilde{d}_1 \\ \tilde{d}_2 \\ \tilde{d}_3 \\ \tilde{d}_4 \\ \tilde{d}_5 \\ \tilde{d}_6 \\ \tilde{d}_7 \\ \tilde{d}_8 \\ \tilde{d}_9 \\ \tilde{d}_{10} \end{bmatrix}
\quad = \quad
\begin{bmatrix}
. & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . \\
1 & 1 & 1 & . & . & . & . & . & . & . \\
. & . & . & 1 & 1 & . & . & . & . & . \\
. & . & . & . & . & 1 & 1 & . & . & . \\
. & . & . & . & . & . & . & 1 & . & . \\
. & . & . & . & . & . & . & . & 1 & .
\end{bmatrix}
\begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \\ m_7 \\ m_8 \\ m_9 \\ m_{10} \end{bmatrix}
\tag{1.11}
$$

A program for **nearest-neighbor normal moveout** as defined by equations (1.10) and (1.11) is `nmo0()`. Because of the limited alphabet of programming languages, I used the keystroke `z` to denote τ.

```
subroutine nmo0( adj, add, slow,    x, t0, dt, n,zz,  tt )
integer  it, iz, adj, add,                          n
real  xs, t , z,           slow(n), x, t0, dt, zz(n), tt(n)
call adjnull(    adj, add,                      zz,n,  tt,n)
do iz= 1, n {    z = t0 + dt*(iz-1)              # Travel-time depth
                 xs= x * slow(iz)
                 t = sqrt ( z * z + xs * xs)
                 it= 1 + .5 + (t - t0) / dt      # Round to nearest neighbor.
                 if( it <= n )
                         if( adj == 0 )
                                 tt(it) = tt(it) + zz(iz)
                         else
                                 zz(iz) = zz(iz) + tt(it)
        }
return; end
```

A program is a "pull" program if the loop creating the output covers each location in the output and gathers the input from wherever it may be. A program is a "push" program if it takes each input and pushes it to wherever it belongs. Thus this NMO program is a "pull" program for doing the model building (data processing), and it is a "push" program for the data building. You could write a program that worked the other way around, namely, a loop over t with z found by calculation $z = \sqrt{t^2/v^2 - x^2}$. What is annoying is that if you want a push program going both ways, those two ways cannot be adjoint to one another.

Normal moveout is a linear operation. This means that data can be decomposed into any two parts, early and late, high frequency and low, smooth and rough, steep and shallow dip, etc.; and whether the two parts are NMO'ed either separately or together, the result is the same. The reason normal moveout is a linear operation is that we have shown it is effectively a matrix multiply operation and that operation fulfills $\mathbf{N}(\mathbf{d_1}+\mathbf{d_2}) = \mathbf{N}\mathbf{d_1}+\mathbf{N}\mathbf{d_2}$.

1.3 COMMON-MIDPOINT STACKING

Typically, many receivers record every shot, and there are many shots over the reflectors of interest. It is common practice to define the midpoint $y = (x_s + x_g)/2$ and then to sort the seismic traces into "**common-midpoint** gathers". After sorting, each trace on a common-midpoint gather can be transformed by NMO into an equivalent zero-offset trace and the traces in the gather can all be added together. This is often called "common-depth-point (**CDP**) stacking" or, more correctly, "**common-midpoint stack**ing".

The adjoint to this operation is to begin from a model that is identical to the zero-offset trace and spray this trace to all offsets. There is no "official" definition of which operator of an operator pair is the operator itself and which is the adjoint. On the one hand, I like to think of the modeling operation itself as *the* operator. On the other hand, the industry machinery keeps churning away at many processes that have well-known names, so people often think of one of them as *the* operator. Industrial data-processing operators are typically **adjoint**s to modeling operators.

Figure 1.2 illustrates the operator pair, consisting of spraying out a zero-offset trace (the model) to all offsets and the adjoint of the spraying, which is **stack**ing. The moveout and stack operations are in subroutine `stack0()`.

```
subroutine stack0( adj, add, slow,     t0,dt, x0,dx, nt,nx, stack, gather)
integer ix,        adj, add,                         nt,nx
real     x,                  slow(nt), t0,dt, x0,dx,   stack(nt), gather(nt,nx)
call adjnull(      adj, add,                           stack,nt,  gather,nt*nx)
do ix= 1, nx {
        x = x0 + dx * (ix-1)
        call nmo0( adj, 1, slow, x, t0,dt, nt, stack, gather(1,ix))
        }
return; end
```

Let $\mathbf{S}'$ denote NMO, and let the stack be defined by invoking `stack0()` with the `adj=1` argument. Then $\mathbf{S}$ is the modeling operation defined by invoking `stack0()` with the `adj=0` argument. Figure 1.2 illustrates both. Notice the roughness on the waveforms caused by

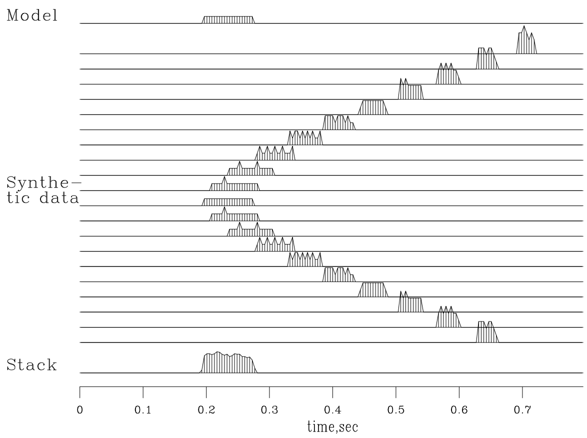

Figure 1.2: Top is a model trace **m**. Center shows the spraying to synthetic traces, $\mathbf{S}'\mathbf{m}$. Bottom is the stack of the synthetic data, $\mathbf{S}\mathbf{S}'\mathbf{m}$. vela-stack [ER]

different numbers of points landing in one place. Notice also the increase of **AVO** (**amplitude** versus **offset**) as the waveform gets compressed into a smaller space. Finally, notice that the stack is a little rough, but the energy is all in the desired time window.

We notice a contradiction of aspirations. On the one hand, an operator has smooth outputs if it "loops over output space" and finds its input where ever it may. On the other hand, it is nice to have modeling and processing be exact adjoints of each other. Unfortunately, we cannot have both. If you loop over the output space of an operator, then the adjoint operator has a loop over input space and a consequent roughness of its output.

1.3.1 Crossing traveltime curves

Since velocity increases with depth, at wide enough offset a deep enough path will arrive sooner than a shallow path. In other words, traveltime curves for shallow events must cut across the curves of deeper events. Where traveltime curves cross, NMO is no longer a one-to-one transformation. To see what happens to the **stack**ing process I prepared Figures 1.3-1.5 using a typical marine recording geometry (although for clarity I used larger $(\Delta t, \Delta x)$) and we will use a typical Texas gulf coast average velocity, $v(z) = 1.5 + \alpha z$ where $\alpha = .5$. First we repeat the calculation of Figure 1.2 with constant velocity $\alpha = 0$ and more reflectors. We see in Figure 1.3 that the **stack** reconstructs the model except for two details: (1) the **amplitude** diminishes with time, and (2) the early waveforms have become rounded. Then we repeat the calculation with the Gulf coast typical velocity gradient $\alpha = 1/2$. The polarity reversal on the first arrival of the wide offset trace in Figure 1.4 is evidence that in practice traveltime curves do cross. (As was plainly evident in Figures **??**, **??** and **??** crossing traveltime curves are even more significant elsewhere in the world.) Comparing Figure 1.3 to Figure 1.4 we see that an effect of the velocity gradient is to degrade the **stack**'s reconstruction of the model. Velocity gradient has ruined the waveform on the shallowest event, at about 400ms. If the plot were made on a finer mesh with higher frequencies, we could expect ruined waveforms a little deeper too.

Our NMO and stack subroutines can be used for modeling or for data processing. In designing these programs we gave no thought to signal **amplitude**s (although results showed

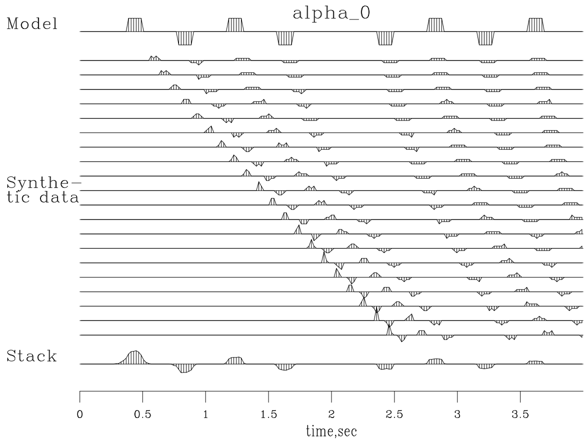

Figure 1.3: Synthetic CMP gather for constant velocity earth and reconstruction. vela-nmo0alfa0 [ER]

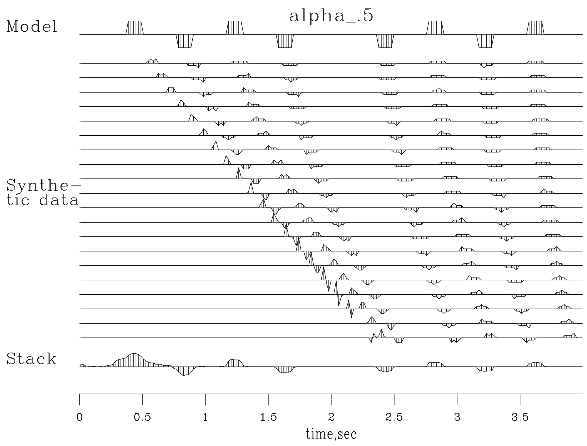

Figure 1.4: Synthetic CMP gather for velocity linearly increasing with depth (typical of Gulf of Mexico) and reconstruction. vela-nmo0alfa.5 [ER]

an interesting **AVO** effect in Figure 1.2.) We could redesign the programs so that the modeling operator has the most realistic **amplitude** that we can devise. Alternately, we could design the **amplitude**s to get the best approximation to $\mathbf{S}'\mathbf{S} \approx \mathbf{I}$ which should result in "`Stack`" being a good approximation to "`Model`." I experimented with various weighting functions until I came up with subroutines `nmo1()` on this page and `stack1()` (like `stack0()` on page 6) which embodies the **weighting function** $(\tau/t)(1/\sqrt{t})$ and which produces the result in Figure 1.5. The result in Figure 1.5 is very pleasing. Not only is the **amplitude** as a function of time

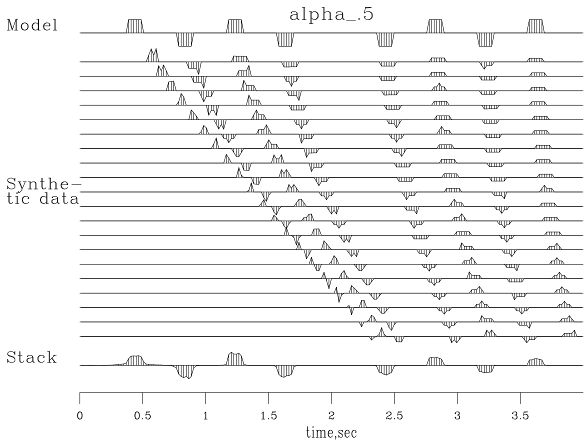

Figure 1.5: Synthetic CMP gather for velocity linearly increasing with depth and reconstruction with weighting functions in subroutine `nmo1()`. Lots of adjustable parameters here. vela-nmo1alfa.5 [ER]

better preserved, more importantly, the shallow wavelets are less smeared and have recovered their rectangular shape. The reason the reconstruction is much better is the cosine weighting implicit in τ/t. It has muted away much of the energy in the shallow asymptote. I think this energy near the asymptote is harmful because the waveform stretch is so large there. Perhaps a similar good result could be found by experimenting with muting programs such as `mutter()` on page ??. However, subroutine `nmo1()` on the current page differs from `mutter()` in two significant respects: (1) `nmo1()` is based on a theoretical concept whereas `mutter()` requires observational parameters and (2) `mutter()` applies a weighting in the coordinates of the (t,x) input space, while `nmo1()` does that but also includes the coordinate τ of the the output space. With `nmo1()` events from different τ depths see different mutes which is good where a shallow event asymptote crosses a deeper event far from its own asymptote. In practice the problem of crossing traveltime curves is severe, as evidenced by Figures **??**-**??** and both weighting during NMO and muting should be used.

```
subroutine nmo1( adj, add, slow,    x, t0, dt, n,zz,  tt )
integer  it, iz, adj, add,                         n
real  xs, t , z,           slow(n), x, t0, dt, zz(n), tt(n), wt
call adjnull(    adj, add,                    zz,n,  tt,n)
do iz= 1, n {   z = t0 + dt*(iz-1)
                xs = x * slow(iz)
                t = sqrt ( z * z + xs * xs) + 1.e-20
                wt = z/t * (1./sqrt(t))                  #  weighting function
                it = 1 + .5 + (t - t0) / dt
                if( it <= n )
                        if( adj == 0 )
                                tt(it) = tt(it) + zz(iz) * wt
                        else
                                zz(iz) = zz(iz) + tt(it) * wt
```

```
            }
return; end
```

It is important to realize that the most accurate possible physical **amplitude**s are not necessarily those for which $\mathbf{S'S} \approx \mathbf{I}$. Physically accurate amplitudes involve many theoretical issues not covered here. It is easy to include some effects (spherical divergence based on velocity depth variation) and harder to include others (surface ghosts and arrays). We omit detailed modeling here because it is the topic of so many other studies.

1.3.2 Ideal weighting functions for stacking

The difference between **stack**ing as defined by `nmo0()` on page 5 and by `nmo1()` on the page before is in the weighting function $(\tau/t)(1/\sqrt{t})$. This weight made a big difference in the resolution of the stacks but I cannot explain whether this weighting function is the best possible one, or what systematic procedure leads to the best weighting function in general. To understand this better, notice that $(\tau/t)(1/\sqrt{t})$ can be factored into two weights, τ and $t^{-3/2}$. One weight could be applied before NMO and the other after. That would also be more efficient than weighting inside NMO, as does `nmo1()`. Additionally, it is likely that these weighting functions should take into account data truncation at the cable's end. Stacking is the most important operator in seismology. Perhaps some objective measure of quality can be defined and arbitrary powers of t, x, and τ can be adjusted until the optimum stack is defined. Likewise, we should consider weighting functions in the spectral domain. As the weights τ and $t^{-3/2}$ tend to cancel one another, perhaps we should filter with opposing filters before and after **NMO** and stack.

1.3.3 Gulf of Mexico stack and AGC

Next we create a "CDP stack" of our the Gulf of Mexico data set. Recall the moved out common-midpoint (CMP) gather Figure **??**. At each midpoint there is one of these CMP gathers. Each gather is summed over its offset axis. Figure 1.6 shows the result of stacking over offset, at each midpoint. The result is an image of a cross section of the earth. In Figure 1.6 the early signals are too weak to see. This results from the small number of traces at early times because of the mute function. (Notice missing information at wide offset and early time on Figure **??**.) To make the stack properly, we should divide by the number of nonzero traces. The fact that the mute function is tapered rather than cut off abruptly complicates the decision of what is a nonzero trace. In general we might like to apply a weighting function of offset. How then should the stack be weighted with time to preserve something like the proper signal strength? A solution is to make constant synthetic data (zero frequency). Stacking this synthetic data gives a weight that can be used as a divisor when stacking field data. I prepared code for such weighted stacking, but it cluttered the NMO and stack program and required two additional new subroutines, so I chose to leave the clutter in the electronic book and not to display it here. Instead, I chose to solve the signal strength problem by an old standby method,

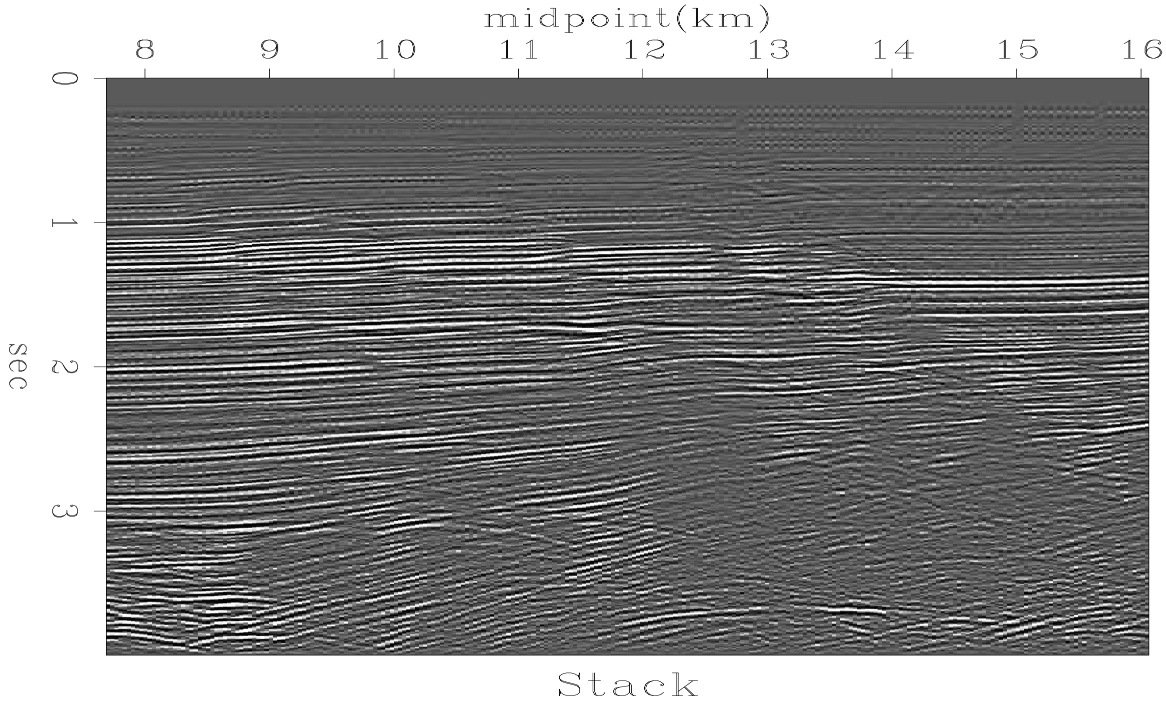

Figure 1.6: Stack done with a given velocity profile for all midpoints. vela-wgstack [ER]

Automatic Gain Control (AGC). A divisor for the data is found by smoothing the absolute values of the data over a moving window. To make Figure 1.7 I made the divisor by smoothing in triangle shaped windows about a half second long. To do this, I used subroutine `triangle()` on page 214.

1.4 VELOCITY SPECTRA

An important transformation in exploration geophysics takes data as a function of shot-receiver offset and transforms it to data as a function of apparent velocity. Data is summed along hyperbolas of many velocities. This important industrial process is adjoint to another that may be easier to grasp: data is synthesized by a superposition of many hyperbolas. The hyperbolas have various asymptotes (velocities) and various tops (apexes). Pseudocode for these transformations is

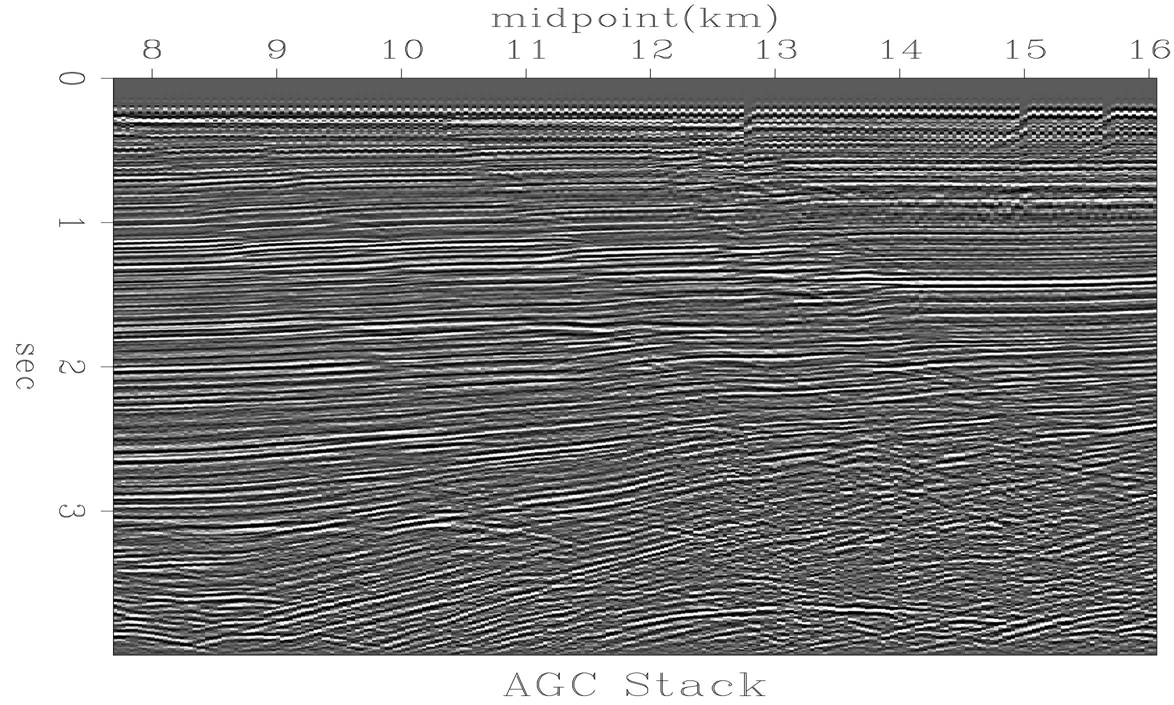

Figure 1.7: Stack of Figure 1.6 after AGC. $\boxed{\text{vela-agcstack}}$ [ER,M]

do v {
do τ {
do x {

$t = \sqrt{\tau^2 + x^2/v^2}$

if hyperbola superposition

data(t,x) = data(t,x) + vspace(τ,v)

else if velocity analysis

vspace(τ,v) = vspace(τ,v) + data(t,x)

}}}

We can ask the question, if we transform data to velocity space, and then return to data space, will we get the original data? Likewise we could begin from the velocity space, synthesize some data, and return to velocity space. Would we come back to where we started? The answer is yes, in some degree. Mathematically, the question amounts to this: Given the operator $\mathbf{A}$, is $\mathbf{A}'\mathbf{A}$ approximately an identity operator, i.e. is $\mathbf{A}$ nearly a unitary operator? It happens that $\mathbf{A}'\mathbf{A}$ defined by the pseudocode above is rather far from an identity transformation, but we can bring it much closer by including some simple scaling factors. It would be a lengthy digression here to derive all these weighting factors but let us briefly see the motivation for them. One weight arises because waves lose **amplitude** as they spread out. Another weight arises because some angle-dependent effects should be taken into account. A third weight arises because in creating a velocity space, the near offsets are less important than the wide offsets and we do not even need the zero-offset data. A fourth weight is a frequency dependent one which is explained in chapter 6. Basically, the summations in the velocity transformation are like integrations, thus they tend to boost low frequencies. This could be compensated by scaling in the frequency domain with frequency as $\sqrt{-i\omega}$ with subroutine `halfdifa()` on page 91.

The weighting issue will be examined in more detail later. Meanwhile, we can see nice quality examples from very simple programs if we include the weights in the physical domain, $w = \sqrt{1/t}\ \sqrt{x/v}\ \tau/t$. (Typographical note: Do not confuse the weight w (double you) with omega ω.) To avoid the coding clutter of the frequency domain weighting $\sqrt{-i\omega}$ I omit that, thus getting smoother results than theoretically preferable. Figure 1.8 illustrates this smoothing by starting from points in velocity space, transforming to offset, and then back and forth again.

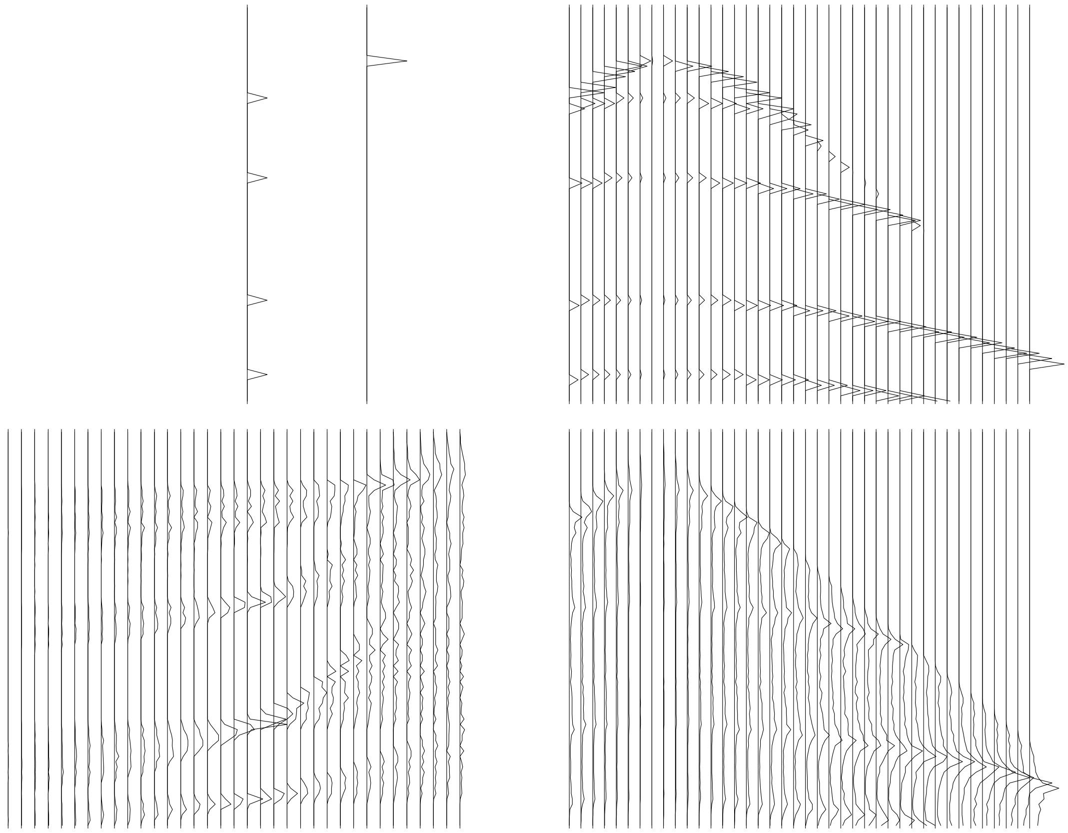

Figure 1.8: Iteration between spaces. Left are model spaces. Right are data spaces. Right derived from left. Lower model space derived from upper data space. vela-velvel [ER]

There is one final complication relating to weighting. The most symmetrical approach is to put w into both $\mathbf{A}$ and $\mathbf{A}'$. This is what subroutine `velsimp()` on this page does. Thus, because of the weighting by $\sqrt{x}$, the synthetic data in Figure 1.8 is nonphysical. An alternate view is to *define* $\mathbf{A}$ (by the pseudo code above, or by some modeling theory) and then for reverse transformation use $w^2\mathbf{A}'$.

```
# velsimp --- simple velocity transform
#
subroutine velsimp( adj,add, t0,dt,x0,dx,s0,ds, nt,nx,ns, modl, data)
integer it,ix,is,   adj,add,                    nt,nx,ns,             iz,nz
real x,s,sx, t,z, z0,dz,wt,   t0,dt,x0,dx,s0,ds, modl(nt,ns),data(nt,nx)
call adjnull(       adj,add,                     modl,nt*ns, data,nt*nx)
nz= nt;  z0=t0; dz= dt;                  # z is travel time depth
do is= 1, ns {  s = s0 + (is-1) * ds
do ix= 1, nx {  x = x0 + (ix-1) * dx
do iz= 2, nz {  z = z0 + (iz-1) * dz
        sx =  abs( s * x)
        t  = sqrt( z * z + sx * sx)
        it = 1.5 + (t - t0) / dt
        if ( it <= nt) {                                wt= (z/t) / sqrt( t)
                if( adj == 0 )
                        data(it,ix) = data(it,ix) + modl(iz,is) * sx * wt
                else
                        modl(iz,is) = modl(iz,is) + data(it,ix) * sx * wt
                }
        }}}
return; end
```

An example of applying subroutine `velsimp()` on the page before to field data is shown in Figure 1.9.

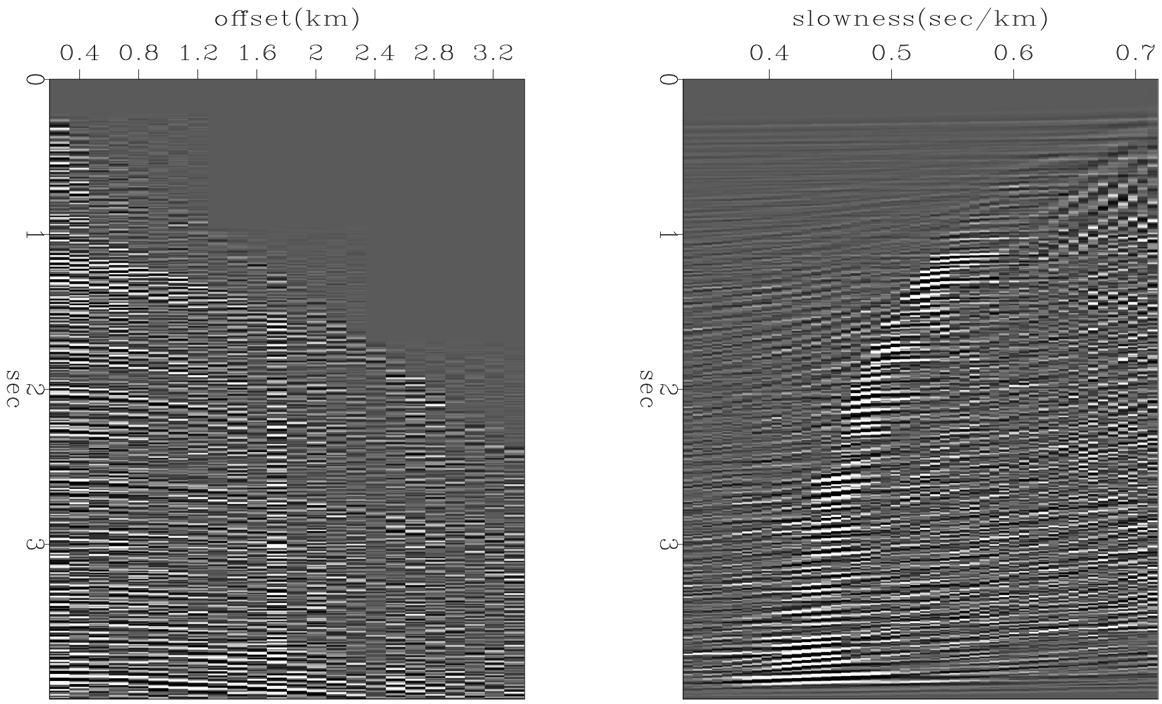

Figure 1.9: Transformation of data as a function of offset (left) to data as a function of slowness (velocity scans) on the right using subroutine `velsimp()`. $\boxed{\text{vela-mutvel}}$ [ER]

1.4.1 Velocity picking

For many kinds of data analysis, we need to know the velocity of the earth as a function of depth. To derive such information we begin from Figure 1.9 and draw a line through the maxima. In practice this is often a tedious manual process, and it needs to be done everywhere we go. There is no universally accepted way to automate this procedure, but we will consider one that is simple enough that it can be fully described here, and which works well enough for these demonstrations. (I plan to do a better job later.)

Theoretically we can define the velocity or slowness as a function of traveltime depth by the moment function. Take the absolute value of the data scans and smooth them a little on the time axis to make something like an unnormalized probability function, say $p(\tau,s) > 0$. Then the slowness $s(\tau)$ could be defined by the moment function, i.e.,

$$s(\tau) \quad = \quad \frac{\sum_s \; s \; p(\tau,s)}{\sum_s \; p(\tau,s)} \tag{1.12}$$

The problem with defining slowness $s(\tau)$ by the moment is that it is strongly influenced by noises away from the peaks, particularly water velocity noises. Thus, better results can be obtained if the sums in equation (1.12) are limited to a range about the likely solution. To begin with, we can take the likely solution to be defined by universal or regional experience. It is sensible to begin from a one-parameter equation for velocity increasing with depth where the form of the equation allows a ray tracing solution such as equation (**??**). Experience with Gulf of Mexico data shows that $\alpha \approx 1/2$ sec^{-1} is reasonable there for equation (**??**), and that is the smooth curve in Figure 1.10.

Experience with moments, equation (1.12), shows they are reasonable when the desired result is near the guessed center of the range. Otherwise, the moment is biased towards the initial guess. This bias can be reduced in stages. At each stage we shrink the width of the zone used to compute the moment. This procedure is used in subroutine `slowfit()` on this page which after smoothing to be described, gives the oscillatory curve you see in Figure 1.10.

```
subroutine slowfit( vsurface, alpha, t0,dt, s0,ds, scan,nt,ns, reg,    slow)
integer irange, it,is,                                   nt,ns
real num,den, t,s,  vsurface, alpha, t0,dt, s0,ds, scan(nt,ns),reg(nt),slow(nt)
do it= 1, nt {  t= t0 + dt*(it-1) + dt
        reg(it)  = 1./( vsurface * sqrt( (exp(alpha*t) - 1.)/(alpha*t) ))
        slow(it) = reg(it)
        }
do irange= ns/4, 5, -1 {                                 # shrink the fairway
do it=     1,   nt     {        t= t0 + dt*(it-1)
        do is= 1, ns   {        s= s0 + ds*(is-1)
                if( s > slow(it) + irange*ds)   scan(it,is) = 0.
                if( s < slow(it) - irange*ds)   scan(it,is) = 0.
                if( s > 1./1.6             )   scan(it,is) = 0.        # water
                }
        den= 0.0;       num= 0.0
        do is= 1, ns {  s= s0 + ds*(is-1)
                num = num + scan(it,is) * s
                den = den + scan(it,is)
```

```
                }
        slow(it) = num / ( den + 1.e-20)
        if( slow(it) == 0.)   slow(it) = 1./vsurface
        }}
return; end
```

A more customary way to view velocity space is to square the velocity scans and normalize them by the sum of the squares of the signals. This has the advantage that the remaining information represents velocity spectra and removes variation due to seismic **amplitude**s. Since in practice, reliability seems somehow proportional to **amplitude** the disadvantage of normalization is that reliability becomes more veiled.

An appealing visualization of velocity is shown in the right side of Figure 1.10. This was prepared from the absolute value of left side, followed by filtering spatially with an antisymmetric leaky integral function. (See PVI page 57). An example is shown on the right side of Figure 1.10.

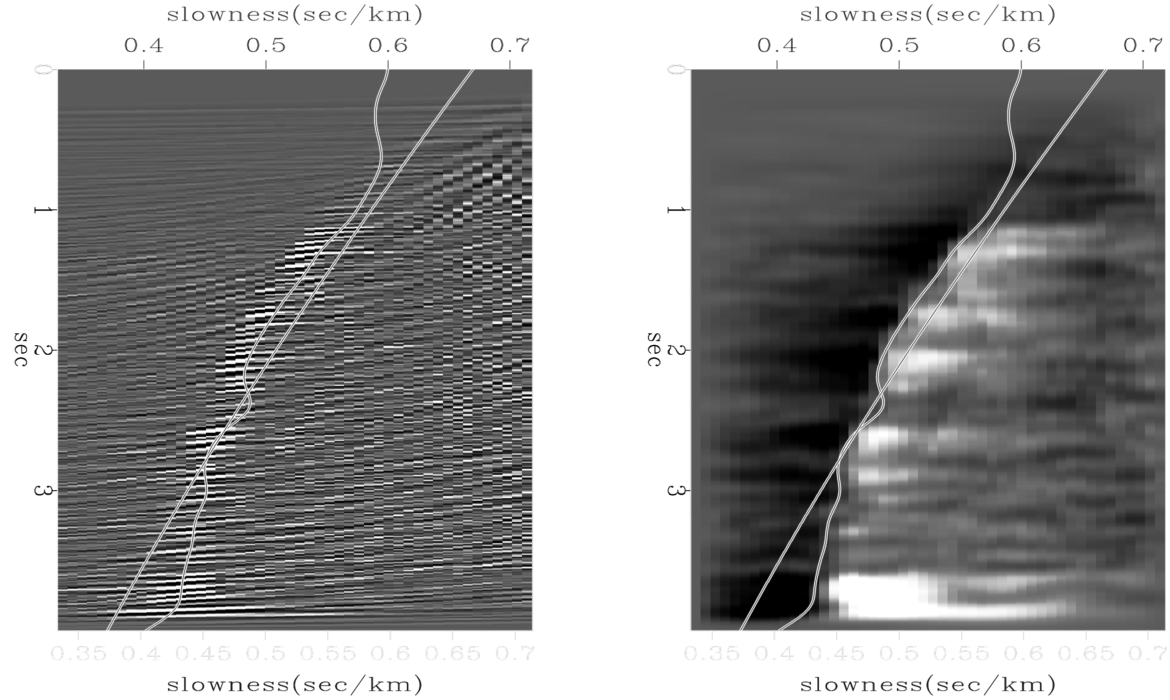

Figure 1.10: Left is the slowness scans. Right is the slowness scans after absolute value, smoothing a little in time, and antisymmetric leaky integration over slowness. Overlaying both is the line of slowness picks. vela-slowfit [ER]

1.4.2 Stabilizing RMS velocity

With velocity analysis, we estimate the RMS velocity. Later we will need both the RMS velocity and the **interval velocity**. (The word "interval" designates an interval between two reflectors.) Recall from chapter **??** equation (**??**)

$$t^2 \quad = \quad \tau^2 + \frac{4h^2}{V^2(\tau)}$$

Routine `vint2rms()` on the current page converts from interval velocity to RMS velocity and vice versa.

```
# Invertible transform from interval velocity to RMS.
#
subroutine vint2rms(  inverse, vminallow, dt, vint, nt,  vrms )
integer it, wide,     inverse,                      nt
real    vmin,                  vminallow, dt, vint( nt), vrms( nt)
temporary real                                vis( nt),  sum( nt)
if( inverse == 0) {             do it= 1, nt
                                        vis(it) = vint(it) ** 2
        sum(1) = 0.;            do it= 2, nt
                                        sum(it) = sum(it-1) + vis(it) * dt
        vrms(1) = vint(1);      do it= 2, nt
                                        vrms(it) = sqrt( sum(it) / ((it-1)*dt) )
        }
else {  do it= 1, nt
                sum(it)= ((it-1)*dt) * amax1( vrms(it)**2, vminallow**2 )
        vis(1) = vrms(1) ** 2
        do it= 2, nt
                vis(it) = ( sum(it) - sum(it-1) )/ dt
        wide= 2;  repeat {
                vmin = vis(1);  do it=1,nt { if( vis(it)<vmin)  vmin = vis(it) }
                if( vmin > vminallow**2 )   break
                call triangle( wide, 1, nt, vis, vis)           # smooth vis()
                wide = wide + 1
                if( wide >= nt/3) call erexit('Velocity less than allowable.')
                }
        do it= 1, nt
                vint(it) = sqrt( vis(it))
        }
return; end
```

The forward conversion follows in straightforward steps: square, integrate, square root. The inverse conversion, like an adjoint, retraces the steps of the forward transform but it does the inverse at every stage. There is however, a messy problem with nearly all field data that must be handled along the inverse route. The problem is that the observed RMS velocity function is generally a rough function, and it is generally unreliable over a significant portion of its range. To make matters worse, deriving an **interval velocity** begins as does a derivative, roughening the function further. We soon find ourselves taking square roots of negative numbers, which requires judgement to proceed. The technique used in `vint2rms()` on this page is to average the squared interval velocity in ever expanding neighborhoods until there are no longer any negative squared interval velocities. As long as we are restricting v^2 from being negative, it is easy to restrict it to be above some allowable velocity, say `vminallow`. Figures 1.11 and 1.12 were derived from the velocity scans in Figure 1.10. Figure 1.11 shows the RMS velocity before and after a trip backward and forward through routine `vint2rms()` on the current page. The interval velocity associated with the smoothed velocity is in figure 1.12.

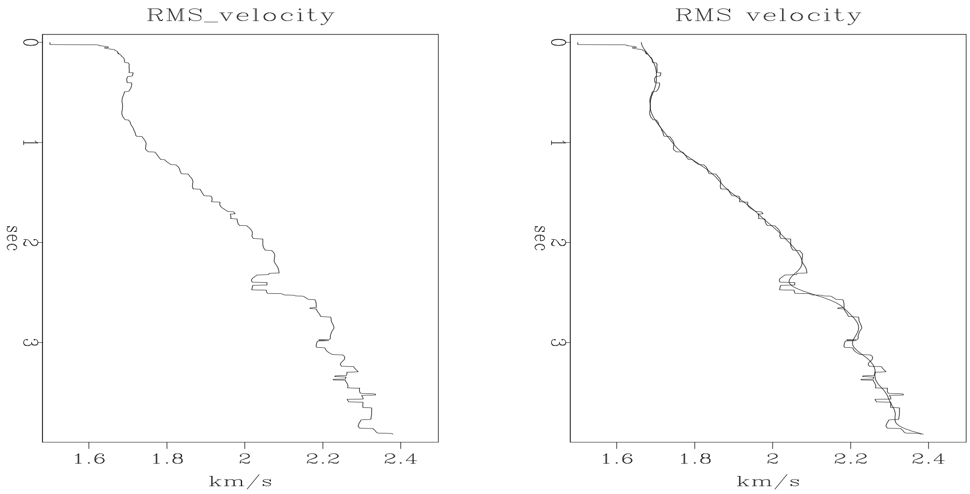

Figure 1.11: Left is the raw RMS velocity. Right is a superposition of RMS velocities, the raw one, and one constrained to have realistic interval velocities. vela-rufsmo [ER]

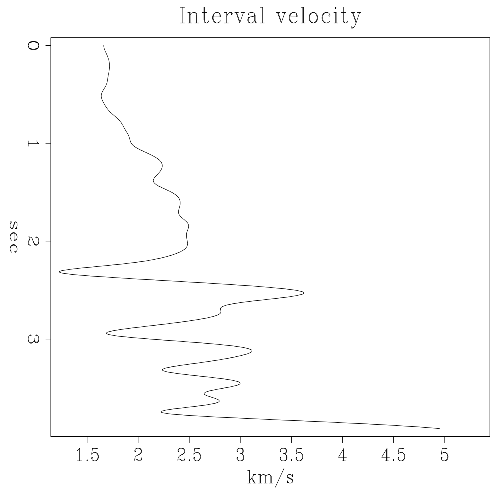

Figure 1.12: Interval velocity associated with the smoothed RMS velocity of Figure 1.11. Pushbutton allows experimentation with `vminallow`. vela-vrmsint [ER]

Index