Chapter 8

Noisy data

8.1 ROW NORMALIZED PEF

We often run into **bursty noise**. This can overwhelm the estimate of a prediction-error filter. To overcome this problem we can use a weighting function. The weight for each row in fitting matrix (**??**) or (**??**) is adjusted so that each row has about the same contribution as each other row. A first idea is that the weight for the n-th row would be the inverse of the sum of the absolute values of the row. This is easy to compute: First make a vector the size of the PEF $\mathbf{a}$ but with each element unity. Second, take a copy of the signal vector $\mathbf{y}$ but with the absolute value of each component. Third, convolve the two. The convolution of the ones with the absolute values could be the inverse of the weighting function we seek. However, any time we are forming an inverse we need to think about the possibility of dividing by zero, how it could arise, and how divisions by "near zero" could be even worse (because a poor result is not immediately recognized). This recalls many ideas in chapter 7. Perhaps we should use something between L_1 and L_2 or Cauchy. In any case, we must choose a scaling parameter that separates "average" rows from unusually large ones. For this choice in subroutine `rnpef1()`, I chose the median.

8.2 DEBURST

We can use the same technique to throw out fitting equations from defective data that we use for missing data. Recall the theory and discussion leading up to Figure 7.4. There we identified defective data by its lack of continuity. We used the fitting equations $0 \approx w_i(y_{i+1} - 2y_i + y_{i-1})$ where the weights w_i were chosen to be approximately the inverse to the residual $(y_{i+1} - 2y_i + y_{i-1})$ itself.

Here we will first use the second derivative (Laplacian in 1-D) to throw out bad points, while we determine the PEF. Having the PEF, we use it to fill in the missing data.

```
module pefest {      # Estimate a PEF avoiding zeros and bursty noise on input.
  use quantile_mod
  use helicon
  use misinput
  use pef
contains
  subroutine pefest1 (niter, yy, aa, lag, mx) {
    integer, intent (in)             :: niter
    real,    dimension (:), pointer  :: yy, aa
    integer, dimension (:), pointer  :: lag
    logical, dimension (:), pointer  :: mx
    real,    dimension (size (yy))   :: rr, yfre
    real                             :: rbar
    integer                          :: stat
    call helicon_init (aa, lag)                    # starting guess
    stat = helicon_lop (.false., .false., yy, rr)
    rbar = quantile( size( yy)/3, abs( rr))        # rbar=(r safe below rbar)
    where( yy == 0. .or. abs( rr) > 5 * rbar)
       yfre = 0.                                   # risky
    elsewhere
       yfre = 1.                                   # safe enuf
    where (mx) yy = 0.
    call find_mask( yfre, aa, lag, mx)
    call find_pef( yy, aa, lag, mx, niter)
  }
}
```

The result of this “PEF-deburst” processing is shown in Figure 1.1.

Given the PEF that comes out of `pefest1()`, subroutine `fixbad1()` below convolves it with the data and looks for anomalous large outputs. For each that is found, the input data is declared defective and set to zero. Then subroutine `mis1()` on page 122 is invoked to replace the zeroed values by reasonable ones.

```
module fixbad { # Given a PEF, find bad data and restore it.
  use mis2
  use quantile_mod
  use helicon
contains
  subroutine fixbad1 (niter, aa, lag, yy) {
    integer, intent (in)            :: niter
    real,    dimension (:), pointer :: aa
    integer, dimension (:), pointer :: lag
    real,    dimension (:)          :: yy
    real,    dimension (size (yy))  :: rr
    logical, dimension (size (yy))  :: known
    integer                         :: stat
    call helicon_init (aa, lag)
    stat = helicon_lop (.false., .false., yy, rr); rr = abs (rr)
    known = ( yy > 0.) .and. ( rr < 4. * quantile( size(rr)/2, rr))
    call mis1 (niter, yy, aa, lag, known, .true., .true.)
 }
}
```

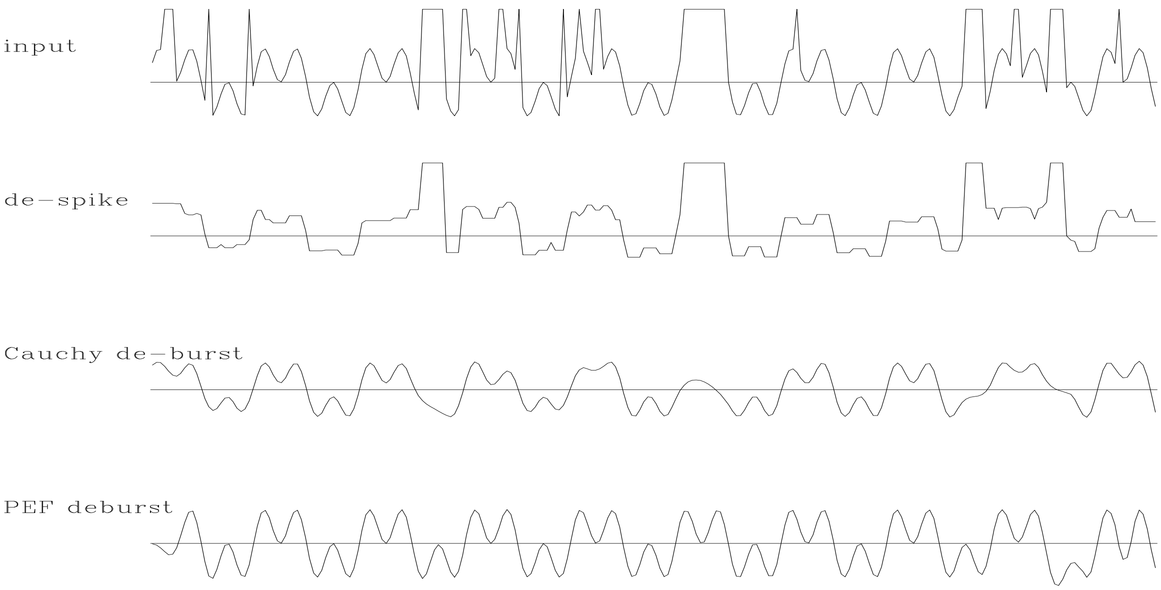

Figure 8.1: Top is synthetic data with noise spikes and bursts. (Some bursts are fifty times larger than shown.) Next is after running medians. Next is Laplacian filter Cauchy deburst processing. Last is PEF-deburst processing. gem-pefdeburst90 [ER]

8.2.1 Potential seismic applications of two-stage infill

Two-stage data infill has many applications that I have hardly begun to investigate.

Shot continuation is an obvious task for a data-cube extrapolation program. There are two applications of shot-continuation. First is the obvious one of repairing holes in data in an unobtrusive way. Second is to cooperate with reflection tomographic studies such as that proposed by Matthias **Schwab**.

Offset continuation is a well-developed topic because of its close link with **dip moveout** (**DMO**). DMO is heavily used in the industry. I do not know how the data-cube extrapolation code I am designing here would fit into DMO and stacking, but because these are such important processes, the appearance of a fundamentally new tool like this should be of interest. It is curious that the DMO operator is traditionally derived from theory, and the theory requires the unknown velocity function of depth, whereas here I propose estimating the offset continuation operator directly from the data itself, without the need of a velocity model.

Obviously, one application is to extrapolate off the sides of a **constant-offset section**. This would reduce migration semicircles at the survey's ends.

Another application is to extrapolate off the **cable ends** of a common-midpoint gather or a common shot point gather. This could enhance the prediction of multiple reflections or reduce artifacts in velocity analysis.

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

8.3 TWO 1-D PEFS VERSUS ONE 2-D PEF

Here we look at the difference between using two 1-D PEFs, and one 2-D PEF. Figure 1.2 shows an example of sparse tracks; it is not realistic in the upper-left corner (where it will be used for testing), in a quarter-circular disk where the data covers the model densely. Such a dense region is ideal for determining the 2-D PEF. Indeed, subroutine `pef2()` on the current page cannot determine a 2-D PEF from the sparse data lines, because at any place you put the filter (unless there are enough adjacent data lines), unknown filter coefficients will multiply missing data. So every fitting goal is nonlinear and hence abandoned by the algorithm.

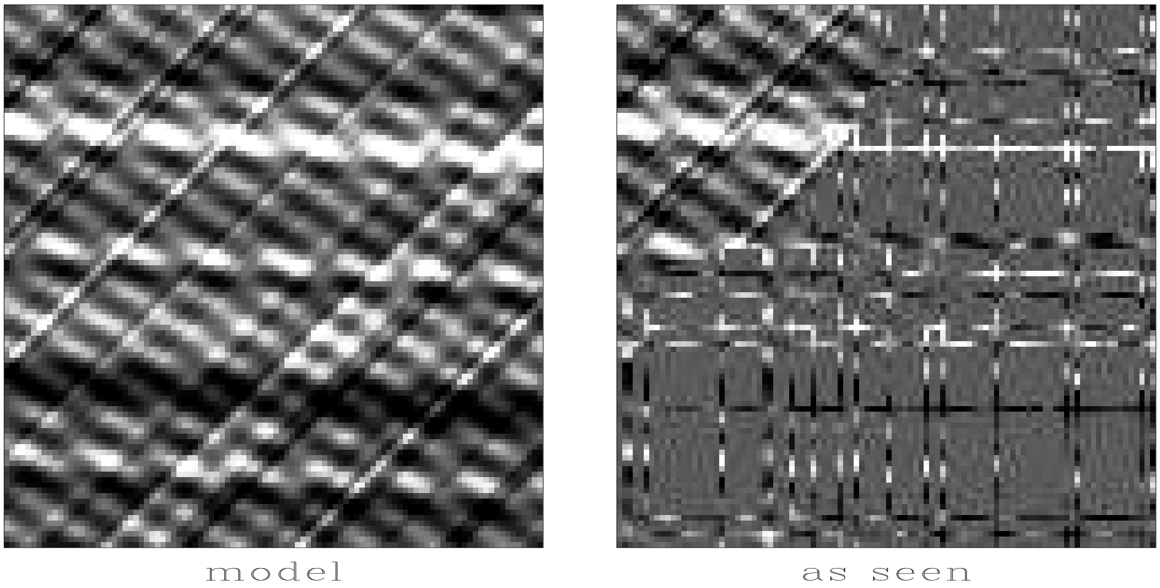

Figure 8.2: Synthetic wavefield (left) and as observed over survey lines (right). The wavefield is a superposition of waves from three directions. gem-duelin [ER]

The set of test data shown in Figure 1.2 is a superposition of three functions like plane waves. One plane wave looks like low-frequency horizontal layers. Notice that the various layers vary in strength with depth. The second wave is dipping about 30° down to the right and its waveform is perfectly sinusoidal. The third wave dips down 45° to the left and its waveform is bandpassed random noise like the horizontal beds. These waves will be handled differently by different processing schemes, so I hope you can identify all three. If you have difficulty, view the figure at a grazing angle from various directions.

Later we will make use of the dense data region, but first let **U** be the east-west PE operator and **V** be the north-south operator and let the signal or image be

$\mathbf{h} = h(x, y)$. The fitting residuals are

$$
\begin{array}{rcl}
\mathbf{0} & \approx & (\mathbf{I} - \mathbf{J})(\mathbf{h} - \mathbf{d}) \\
\mathbf{0} & \approx & \mathbf{U}\ \mathbf{h} \\
\mathbf{0} & \approx & \mathbf{V}\ \mathbf{h}
\end{array}
\tag{8.1}
$$

where $\mathbf{d}$ is data (or binned data) and $(\mathbf{I} - \mathbf{J})$ masks the map onto the data.

Figure 1.3 shows the result of using a single one-dimensional PEF along either the vertical or the horizontal axis.

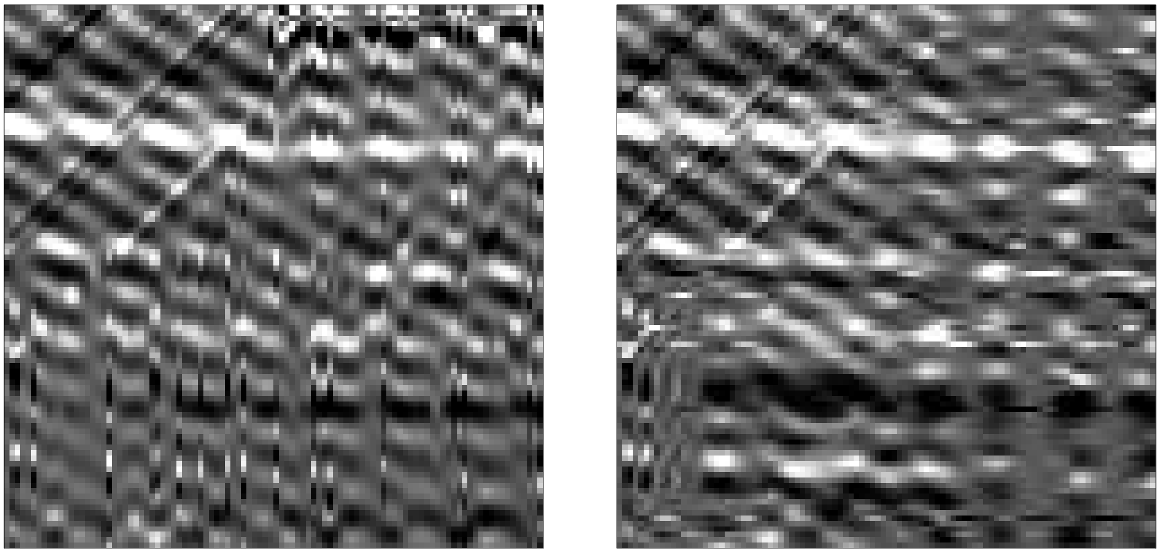

Figure 8.3: Interpolation by 1-D PEF along the vertical axis (left) and along the horizontal axis (right). gem-dueleither [ER]

Figure 1.4 compares the use of a pair of 1-D PEFs versus a single 2-D PEF (which needs the "cheat" corner in Figure 1.2. Studying Figure 1.4 we conclude (what theory predicts) that

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

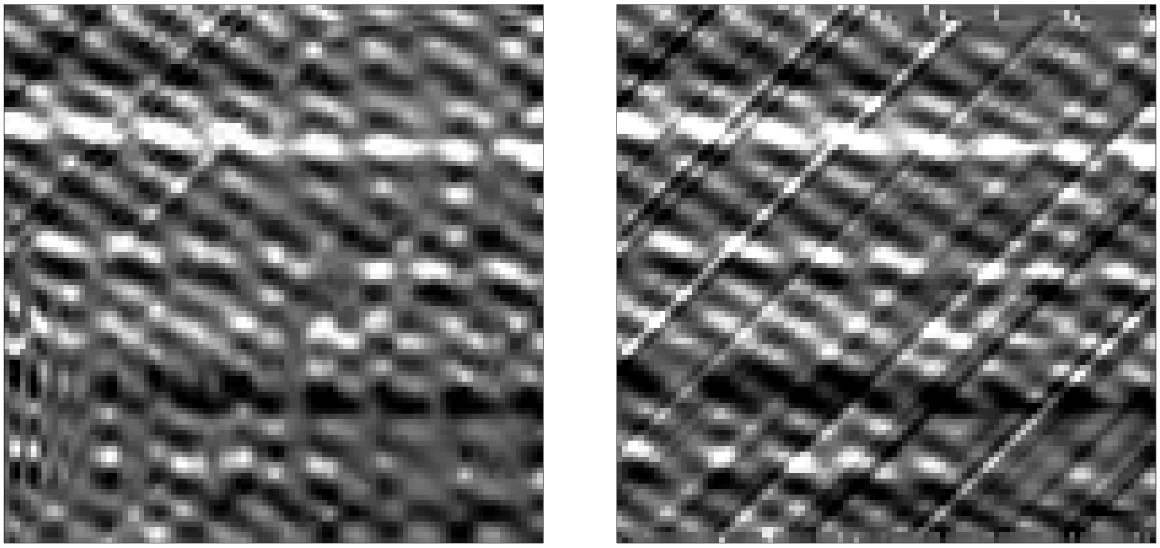

Figure 8.4: Data infilled by a pair of 1-D PEFs (left) and by a single 2-D PEF (right). gem-duelversus [ER]

8.4 SPARSE TRACKS IN SATELLITE ALTIMETRY

Earth satellites orbiting over the poles go round us in north-flying tracks and south-flying tracks. Their orbits would be north-south lines, but the earth's rotation causes them to appear to drift westward (with the sun) so the north-going runs leave northwest-going tracks and the southward runs leave southwest-going tracks. Figure 1.5 shows an interesting topography striated in two non-orthogonal directions (much like superposed plane waves `;-)` and some tracks. For my convenience I have plotted northwest tracks in a vertical direction and southwest tracks in a horizontal direction. In reality the track types are not orthogonal, so my displays will have a shearing distortion (not affecting the analysis). The satellites considered here measure **sea surface** altitude. You might think the sea surface should be at the same altitude everywhere, but the gravitational attraction of the mountains beneath the sea causes the altitude of sea level to vary from place to place. It would be an arduous task to survey all the oceans to find their depth and map it, or to map the strength of gravity on all the ocean surfaces, but either map looks much like each other and both look like the satellite altimeter map, the differences being mainly in calibration and spatial filtering.

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

Notice that some tracks are brighter and some are dimmer. This mimics a cali-

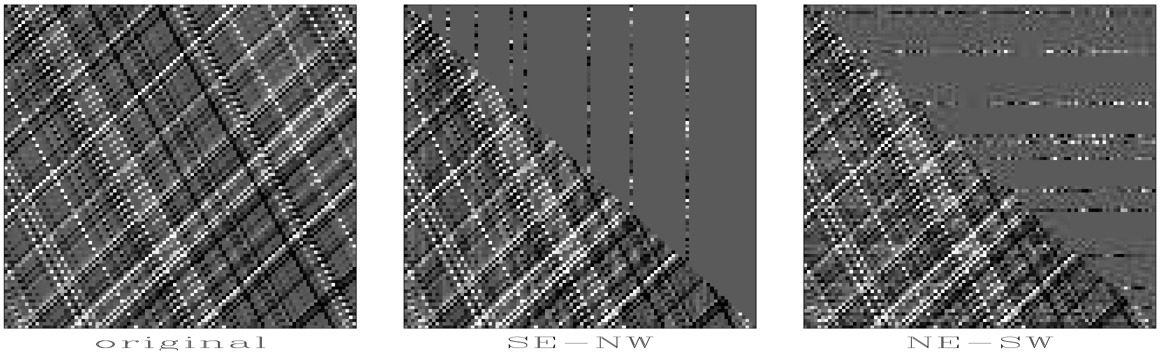

Figure 8.5: Synthetic topography (left), northward tracks (center), and southward tracks (right). The south pole is the lower left-hand corner and thirty degrees south latitude is the diagonal. gem-synsat [ER]

bration problem that causes each track to have a different mean, or more generally, a very low-frequency, along-the-track noise.

Although the geometry of Figure 1.5 is somewhat unusual, it is an attractive example for study for several reasons. First, it exhibits the familiar problem that survey lines are never as dense as we wish them to be. Second, it is not unusual for crossing lines to have inconsistency. Third, having dense coverage near sparse coverage offers us a good chance to study the transition from a region of high information density to a region of low density and ambiguity (spatial aliasing).

The obvious way to fill in between the tracks is with some kind of smoothness criterion. For example, we could have the solution fit the track data exactly and between the tracks we could have the solution satisfy Laplace's equation. For Figure 1.6, I chose instead to minimize the *energy* in the output of the **Laplacian operator** (which amounts to satisfying the *squared* Laplace equation). The problem with either of these approaches is the implicit presumption of isotropy, i.e., that the interpolation be independent of orientation. The best result is obviously the last one in Figure 1.6 and we'll explore that next.

8.4.1 A deeper model than anisotropy

From the data south of 30° latitude in Figure 1.5, we see the orientation of the two **striations** (along with the third striation of along-the-track noise). But there are two orientations in the topography, not just one, so the problem is not so simple that we can merely find a skewing and stretching of coordinates that would make the image appear isotropic. The approach I advocate is to seek a partial differential equation that the image will be a solution of. In reality, a complicated image will satisfy no simple differential equation, so instead I seek a filter with minimum power out. I first find this filter in the region of good spatial coverage. Then I carry it north to the area of sparse coverage where I find the solution by the principle of minimum power out.

The result is shown on the right in Figure 1.6. The result is overwhelmingly better than the isotropic approach. The result is almost perfect but the solutions might still be a little weaker in the middle of large track gaps because I limited the number of conjugate-direction iterations to 20. (The figure computes in a few seconds.)

Figure 8.6: Adjoint reconstruction (left), reconstruction by Laplacian (center), and reconstruction by 2-D PEF (right). gem-seasat [ER]

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

8.4.2 Distribution of sparsity

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

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

8.4.3 Seasat modeling and solution formulation

The problem formulation is,

$$\mathbf{0} \approx \mathbf{A}_n(\mathbf{Th} - \mathbf{d}) \tag{8.2}$$
$$\mathbf{0} \approx \mathbf{A}_h \, \epsilon \, \mathbf{h} \tag{8.3}$$

where $\mathbf{A}_n$ is a noise whitener, $\mathbf{A}_h$ is a topography whitener (PEF), $\mathbf{d}$ is data, $\mathbf{h}$ is altitude, $\mathbf{T}$ is the operator that makes data tracks from an altitude map, and ϵ is the usual damping parameter. Figure 1.6 uses two different values of $\mathbf{A}_h$, one a Laplacian and one a PEF.

Subroutine `icai2()` is yet another 2-D convolution program, a straightforward generalization of the 1-D subroutine `icaf1()` on page 10, but with the adjoint being the input instead of the filter. We need it to apply the spatial PEF to the topography without overflowing the edges.

```
# Internal Convolution, Adjoint is the Input.  2-dimensional
#
subroutine icai2( adj,add, lag1,lag2, aa,na1,na2,  xx,n1,n2,  yy )
integer           adj,add, lag1,lag2,    na1,na2,     n1,n2
real                                aa(na1,na2), xx(n1,n2), yy(n1,n2)
integer                                 a1, a2,     x1,x2,     y1,y2
call adjnull(     adj,add,                         xx,n1*n2,  yy,n1*n2)
do a2=1,na2 {  do y2= 1+na2-lag2, n2-lag2+1 {  x2= y2 - a2 + lag2
do a1=1,na1 {  do y1= 1+na1-lag1, n1-lag1+1 {  x1= y1 - a1 + lag1
        if( adj == 0 )
                        yy( y1,y2) = yy( y1,y2) + xx( x1,x2) * aa( a1,a2)
        else
                        xx( x1,x2) = xx( x1,x2) + yy( y1,y2) * aa( a1,a2)
        }}  }}
return; end
```

Subroutine `track()` shows how tracks are defined from topography, and it also does the adjoint reconstruction of topography from tracks. The array `dknow(,,)` is 1 where data is recorded and 0 where it is not. The subroutine also carries along the $\epsilon\mathbf{I}$ operator with subroutine `ident()` on page 10.

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

```
              hh(i1,i2)    =  hh(i1,i2)    + dd(i1,i2,i3) * dkno(i1,i2,i3)
call ident( adj, 1, eps, n1*n2, hh,           dd( 1, 1, 3) )
return; end
```

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

8.4.4 Along-track noise and crooked tracks

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

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

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

```
call icai2( adj,0, 1,1, del, nd,1, rr(rn), n1,n2, wr(rn))
call icai2( adj,0, 1,1, del, 1,nd, rr(rs), n1,n2, wr(rs))
call icai2( adj,0, 1,1, aa,a1,a2,  rr(rh), n1,n2, wr(rh))
return; end
```

The idea behind subroutine `track()` applies also to crooked survey lines. For use with crooked lines, however, subroutine `trakwit()` would need modification so that it could apply the data noise-whitening filter along the actual track and not simply along the coordinate axis.

8.4.5 Seasat optimization

To prepare to minimize the fitting residuals (1.2), we begin by restating the residual as the output of the operator that takes a model perturbation $\Delta\mathbf{h}$ and produces the weighted residual perturbation $\Delta\mathbf{r}$.

$$\begin{bmatrix} \\ \Delta\mathbf{r} \\ \\ \end{bmatrix} \quad = \quad \begin{bmatrix} \mathbf{A}_n & \cdot & \cdot \\ \cdot & \mathbf{A}_n & \cdot \\ \cdot & \cdot & \mathbf{A}_h \end{bmatrix} \begin{bmatrix} \mathbf{N} \\ \mathbf{S} \\ \epsilon\mathbf{I} \end{bmatrix} [\Delta\mathbf{h}] \tag{8.4}$$

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

To find the best-fitting topography $\mathbf{h} = h(x,y)$, we prepare the solver subroutine `seatopo()`. Begin by loading the residual vector with the negative of the data $\mathbf{r} = -\mathbf{d}$ and load the solution vector with zeros $\mathbf{h} = \mathbf{0}$. Then proceed as we did with earlier weighted residuals like `pef1()` on this page or `pef2()` on the current page. First we revise the data to weighted data by a single application of subroutine `trakwit()` on page 10. Then begin the usual conjugate-direction iteration containing first the adjoint of operator (1.4) and then the forward operator. The only thing new here is that the *filter weighting* cannot have the output and input share the same computer memory, as is allowed by the more usual *scalar weighting* routines, `ident()` on page 10 and `diag()` on this page, so an extra array `dr2` must be declared. (In a computer language better than Fortran there would be no need to list separate programs for each application of weighted least squares—one routine should meet all needs.)

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

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

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

```
# Find the topo map from data and either a previous topo map or a PEF.
#
subroutine seastat( dd,n1,n2, niter, a1,a2, hh, laplace)
integer i1,i2,i3,      n1,n2, niter, a1,a2,     laplace,      lag1,lag2, iter
real                dd(n1,n2,2),            hh(n1,n2), eps
temporary real    dkno(n1,n2,2), afre(a1,a2)
temporary real     res(n1,n2  ),   aa(a1,a2)
eps= 1.;        lag1=1+a1/2;    lag2=1
do i3=1, 2{
do i2=1,n2{
do i1=1,n1  {   if( dd(i1,i2,i3) == 0.)   dkno(i1,i2,i3) = 0.0
                else                      dkno(i1,i2,i3) = 1.0
                }}};           call null( aa, a1*a2)
if( laplace > 0 ) {                     aa(lag1-1,2) =  .25
                aa(lag1+0,1) = .25;     aa(lag1  ,2) = -1.0;    aa(lag1,3)= .25
                                        aa(lag1+1,2) =  .25                   }
else {  do i1=1,n1{
        do i2=1,n2   {
          if( dkno(i1,i2,1) == 0. || dkno(i1,i2,2) == 0.  )   hh(i1,i2) = 0.  }}
        call setfree(                     lag1,lag2, 1, afre,    a1,a2,1, iter)
        call pef2( hh, res, n1,n2,   iter, lag1,lag2,    afre, aa,a1,a2 )
        }
call seatopo( dkno, dd,n1,n2, niter, eps, aa,a1,a2,  hh )
return; end
```

8.5 ELIMINATING SHIP TRACKS IN GALILEE

All **Galilee** imaging formulations until now have produced images with survey-vessel tracks in them. We do not want those tracks. Allow me a hypothetical explanation for the tracks. Perhaps the level of the lake went up or down because of rain or drought during the months of the survey. Perhaps some days the vessel was more heavily loaded and the sensor was deeper in the water. We would not have this difficulty if instead of measuring depth, we measured water bottom slope, say by subtracting two successive depth measurements. This gives a new problem, that of finding the map of the topography of the water bottom from measurements of its slopes along ships' tracks. We can express this as the fitting goal

$$\mathbf{0} \approx \frac{d}{dt}(\mathbf{h} - \mathbf{d}) \tag{8.5}$$

where d/dt is the derivative along the data track. The operator d/dt is applied to both the observed depth and the theoretical depth. The track derivative follows the survey ship and if the ship goes in circles the track derivative does too. We represent the derivative by the $(1, -1)$ operator. There is a Fourier space in which this operator is simply a weighting function that weights the zero spatial frequency to zero value.

> To eliminate vessel tracks in the map, we apply along the track a derivative to both the model and the data.

A beginner might believe that if the ship changes speed or stops while the depth sounder continues running, that we should divide the depth differences by the distance traveled. We could try that, but it might be neither necessary nor appropriate because the $(1, -1)$ operator is simply a weighting function for a statistical estimation problem, and weighting functions do not need to be known to great accuracy. Perhaps the best weighting function is the prediction-error filter determined from the residual itself. Without further ado, we write the noise-weighting operator as $\mathbf{A}$ and consider it to be either d/dt or a PEF. Notice that we encounter PEFs in both data space and model space. We have been using $\mathbf{U}$ and $\mathbf{V}$ to denote PEFs on the final map, and now in the data space we have the PEF $\mathbf{A}$ on the residual.

8.5.1 Using a PEF on the Galilee residual

For simplicity, we begin with a simple gradient for the PEF of the map. We have

$$\begin{array}{lcl} 0 & \approx & \mathbf{A}(\mathbf{B}\mathbf{h} - \mathbf{d}) \\ 0 & \approx & \epsilon \nabla \mathbf{h} \end{array} \tag{8.6}$$

This is our first fitting system that involves all the raw data. Previous ones have involved the data only after binning. Dealing with all the raw data, we can expect even more difficulty with impulsive and erratic noises. The way to handle such noise is via the weighting functions of Chapter 7. Including such a weighting function gives us the map-fitting goals,

$$\begin{array}{lll} 0 & \approx & \mathbf{WA}(\mathbf{Bh} - \mathbf{d}) \\ 0 & \approx & \epsilon \nabla \mathbf{h} \end{array} \tag{8.7}$$

The weighting function depends on the residual itself, via such equations as (7.14) or (7.16). The first time I solved the problem, I chose the PEF $\mathbf{A}$ to be the derivative $(1, -1)$ along the data track. To avoid an excessive influence of noise bursts, I applied row-normalized PEF estimation using subroutine `rnpef1()` on this page. I tried various scalings between the first and second parts of (1.7) and various values of the average residual $\bar{r}$ required in the weights (7.14) and (7.16). Some sample noise PEFs I found were

```
Noise PEFs in the problem:   min ||WA(Bm-d)|| + ||Gm||
 sum     PEF of residual.
 0.10    1.00 -0.34 -0.45 -0.06  0.01 -0.03 -0.01 -0.03 -0.00  0.00
 0.09    1.00 -0.28 -0.48 -0.09  0.01 -0.03 -0.01 -0.03 -0.00 -0.00
 0.05    1.00 -0.44 -0.46 -0.05  0.03 -0.03 -0.00 -0.02  0.02  0.02
```

Involving as it does so many aspects of this book, the solver `potato()` is quite a clutter. Sorry about that. Maybe it will look cleaner after I learn more about all the features in Fortran 90.

```
subroutine potato( xy,       dd,nd, o1,d1,o2,d2, mm,m1,m2, niter)
integer iter, i,      nd,                            m1,m2, niter
real               xy(nd,2), dd(nd),o1,d1,o2,d2, mm(m1,m2), rbar, aa(100)
integer nr, rd,rm, na
temporary real               rr( nd + 2*m1*m2), ad(nd), wr(nd), ww(nd)
temporary real  dm(m1*m2),   dr( nd + 2*m1*m2)
temporary real  sm(m1*m2),   sr( nd + 2*m1*m2)
                             nr= nd + 2*m1*m2;   rd=1;  rm=1+nd
na=4;  aa(1)=1.;  aa(2)= -.44;  aa(3)= -.46;  aa(4)=-.05
call scaleit( 2.,  na,  aa)
do iter= 0, niter {
        call negcopy(     nd,          dd,           ad       ) #        -d
        call dpbin2( 0, 1, o1,d1,o2,d2,xy,mm,m1,m2, ad    ,nd) # r_d=   -d+Bm
        rbar=.5; rbar=2.5; rbar=1.0
        do i=1,nd
                ww(i) = 1./sqrt( 1. + (ad(i)/rbar)**2 )
        call icai1(  0, 0, 1, aa,na,        ad ,nd, rr(rd)   ) # r_d= A(-d+Bm)
        call diag(   0, 0, ww, nd,          rr(rd), rr(rd)   ) # r_d=WA(-d+Bm)
        call igrad2( 0, 0,   m1,m2,       mm,       rr(rm)   ) # r_m =    G m
        call diag(   1, 0, ww, nd,        wr,       rr(rd)   ) #          Wr_d
        call icai1(  1, 0, 1, aa,na,      ad ,nd,   wr       ) # dm =   A'Wr_d
        call dpbin2( 1, 0, o1,d1,o2,d2,xy,dm,m1,m2, ad    ,nd) # dm = B'A'r_d
        call igrad2( 1, 1,   m1,m2,       dm,       rr(rm)   ) # dm= dm + G'r_m
        call dpbin2( 0, 0, o1,d1,o2,d2,xy,dm,m1,m2, ad    ,nd) # dr_d =  B dm
        call icai1(  0, 0, 1, aa,na,      ad,nd,    wr       ) # dr_d = AB dm
        call diag(   0, 0, ww, nd,        wr,       dr(rd)   ) # dr_d =WAB dm
        call igrad2( 0, 0,   m1,m2,       dm,       dr(rm)   ) # dr_m =  G dm
```

```
        call cgplus( iter, m1*m2, mm, dm, sm, _
                           nr, rr, dr, sr )
        }
return; end
```

Results are in Figure 1.7. It is pleasing to see the ship's tracks gone at last.

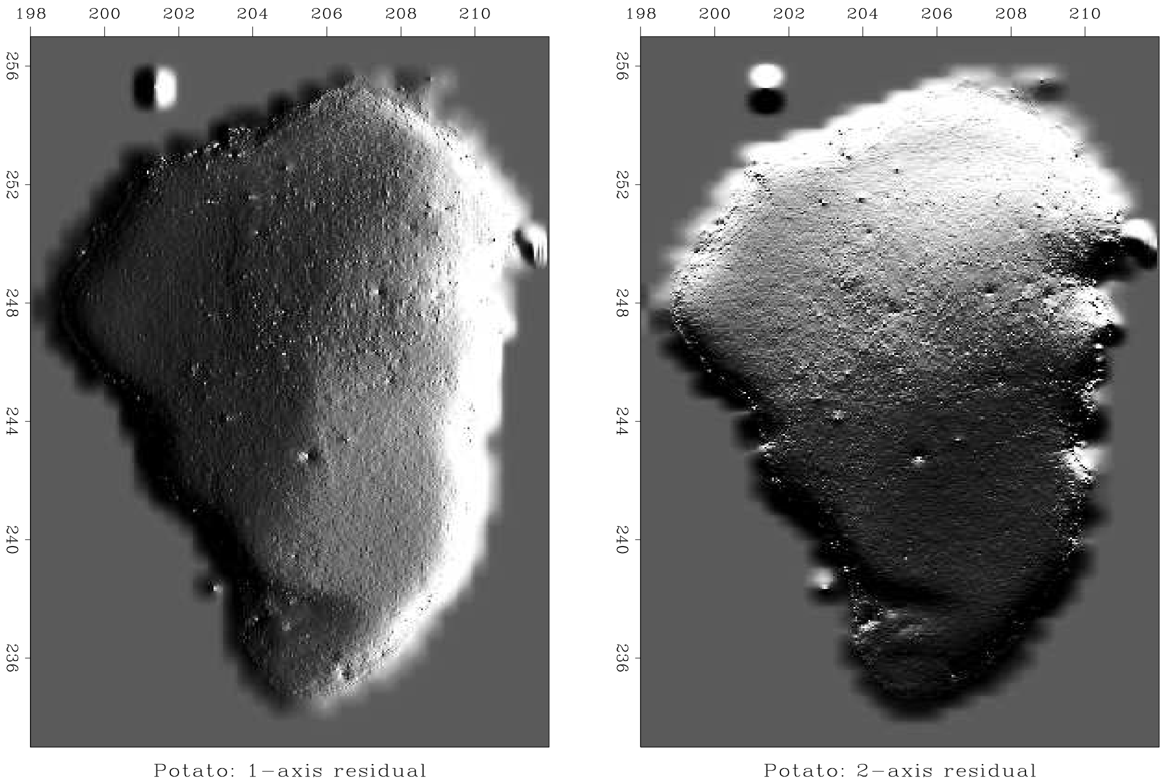

Figure 8.7: Gradient of the Galilee map from (1.7). gem-potato [ER]

8.5.2 PEFs on both model space and residual space

Finally, let us use PEFs in both data space and map space.

$$\begin{array}{lll} 0 & \approx & \mathbf{W}\mathbf{A}(\mathbf{B}\mathbf{h}-\mathbf{d}) \\ 0 & \approx & \epsilon\mathbf{U}\mathbf{h} \\ 0 & \approx & \epsilon\mathbf{V}\mathbf{h} \end{array} \tag{8.8}$$

I omit the display of my subroutine for the goals (1.8) because the code is so similar to `potato()`. (Its name is `pear()` and it is in the library.)

A disadvantage of the previous result in Figure 1.7 is that for the horizontal gradient, the figure is dark on one side and light on the other, and likewise for the vertical gradient. Looking at the result in Figure 1.8 we see that this is no longer true. Thus although the topographic PEFs look similar to a gradient, the difference is substantial.

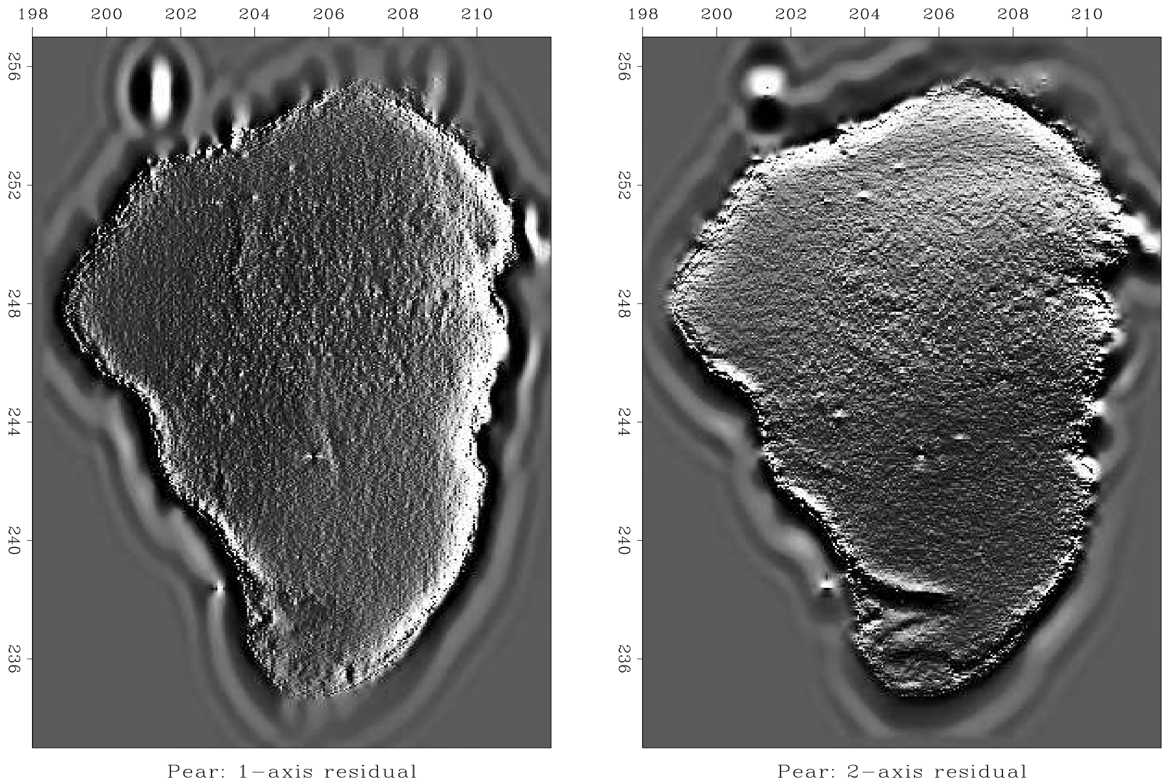

Figure 8.8: Galilee residuals estimated by (1.8). gem-pear [ER]

Subjectively comparing Figures 1.7 and 1.8 our preference depends partly on what we are looking at and partly on whether we view the maps on paper or a computer screen. Having worked on this so long, I am disappointed that most of my 1997 readers are limited to the paper. Another small irritation is that we have two images for each process when we might prefer one. We could have a single image if we go to a single model roughener. Sergey Fomel tested ∇^2 and found it disappointing. An alternative would be to use the symmetrical roughener `rufftri2()` on the current page.

I have wondered whether any significant improvements might result from using linear interpolation $\mathbf{L}$ instead of simple binning $\mathbf{B}$. The subroutine arguments are identical in subroutine `lint2()` on page 18 and subroutine `bin2()` on page 13, so they are "plug compatible" and we could easily experiment.

8.5.3 Regridding

Because of the weighting $\mathbf{W}$, which is a function of the residual itself, the fitting problems (1.7) and (1.8) are nonlinear. Thus a nonlinear solver is required. Unlike linear solvers, nonlinear solvers need a good starting approximation so they do not land in a false minimum. (Linear solvers benefit too by converging more rapidly when started from a good approximation.) I chose the starting solution $\mathbf{h}_0$ beginning from median binning on a coarse mesh. Then I refined the mesh with linear interpolation.

The regridding chore reoccurs on many occasions so I present reusable code. When a continuum is being mapped to a mesh, it is best to allocate to each mesh point an equal area on the continuum. Thus we take an equal interval between each point, and a half an interval beyond the end points. Given `n` points, there are `n-1` intervals between them, so we have

```
min = o - d/2
max = o + d/2 + (n-1)*d
```

which may be back solved to

```
d = (max-min)/n
o = (min*(n-.5) + max/2)/n
```

which is a memorable result for `d` and a less memorable one for `o`. With these not-quite-trivial results, we can invoke the linear interpolation subroutine `lint2()`. It is designed for data points at irregular locations, but we can use it for regular locations too. Subroutine `refine2()` defines pseudoirregular coordinates for the bin centers on the fine mesh and then invokes `lint2()` to carry data values from the coarse regular mesh to the pseudoirregular finer mesh. Upon exiting from `refine2()`, the data space (normally irregular) is a model space (always regular) on the finer mesh.

```
# Refine mesh. Input mm(m1,m2) is coarse.  Output dd(n1,n2) linear interpolated.
#
subroutine refine2( co1,cd1,co2,cd2, mm,m1,m2,    fo1,fd1,fo2,fd2, dd, n1,n2)
integer i1,i2, id,                     m1,m2,                         n1,n2
real                                mm(m1,m2),                     dd( n1*n2)
real                co1,cd1,co2,cd2,             fo1,fd1, fo2,fd2
#     inputs= coarse o1  d1  o2  d2  mesh.      | Outputs= fine    mesh
real xmin,xmax, ymin,ymax, x,y
temporary       real                  xy( n1*n2, 2)
xmin = co1-cd1/2;   xmax = co1 +cd1*(m1-1) +cd1/2       # Great formula!
ymin = co2-cd2/2;   ymax = co2 +cd2*(m2-1) +cd2/2
fd1= (xmax-xmin)/n1;  fo1= (xmin*(n1-.5) + xmax/2)/n1    # Great formula!
fd2= (ymax-ymin)/n2;  fo2= (ymin*(n2-.5) + ymax/2)/n2
do i2=1,n2 {  y = fo2 + fd2*(i2-1)
do i1=1,n1 {  x = fo1 + fd1*(i1-1)
        id = i1+n1*(i2-1)
        xy( id, 1) = x
        xy( id, 2) = y
        }}
call lintr2( 0,0, co1,cd1, co2,cd2, xy, mm,m1,m2, dd, n1*n2)
return; end
```

Finally, here is the 2-D linear interpolation subroutine `lint2()`, which is a trivial extension of the 1-D version `lint1()` on page 15.

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

8.5.4 Treasure hunting at Galilee

Before I go diving, digging, or dreaming at **Galilee**, there are a few more things to attend to. The original data is recorded at about 25-m intervals but the maps shown here are at 50 m, so to find small treasure I should make maps at higher resolution. This aggravates the noise problem. We see that all the Galilee maps contain glitches that are suspiciously nongeological and nonarcheological. Ironically, the process of getting rid of the tracks in Figures 1.7 and 1.8 creates glitches at track ends. There would be no such glitches if the vessel switched on the depth sounder in port in the morning and switched it off after return. The problem arises when tracks start and stop in the middle of the lake. To handle the problem we should keep track of individual tracks and, with the noise PEF (approximately $(1, -1)$), convolve *internally* within each track. In principle, the extra bookkeeping should not be a chore in a higher-level computing language. Perhaps when I become more accustomed to F90, I will build a data container that is a collection of tracks of unequal lengths and a filter program to deal with that structure. Figure 1.9 shows not the tracks (there are too many), but the gaps between the end of one track and the beginning of the next wherever that gap exceeds 100 m, four times the nominal data-point separation.

Besides these processing defects, the defective data values really should be removed. The survey equipment seemed to run reliably about 99% of the time but

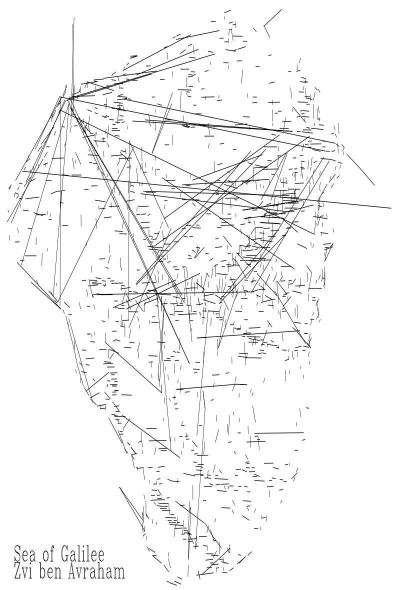

Figure 8.9: The lines are the gaps between successive vessel tracks, i.e., where the vessel turned off the depth sounder. At the end of each track is a potential glitch. gem-seegap [ER]

when it failed, it often failed for multiple measurements. Even with a five-point running median (which reduces resolution accordingly), we are left with numerous doubtful blips in the data. We also see a few points and vessel tracks outside the lake (!); these problems suggest that the navigation equipment is also subject to occasional failure and that a few tracks inside the lake may also be mispositioned.

Rather than begin hand-editing the data, I suggest the following processing scheme: To judge the quality of any particular data-point, we need to find from each of two other nearby tracks the nearest data point. If the depth at our point is nearly the same depth of *either* of the two then we judge it to be good. We need *two* other tracks and not two points from a single other track because bad points often come in bunches.

I plan to break the data analysis into tracks. A new track will be started wherever $(x_i - x_{i-1})^2 + (y_i - y_{i-1})^2$ exceeds a threshold. Additionally, a new track will be started whenever the apparent water-bottom slope exceeds a threshold. After that, I will add a fourth column to the (x_i, y_i, z_i) triplets, a weighting function w_i which will be set to zero in short tracks. As you might imagine, this will involve a little extra clutter in the programs. The easy part is the extra weighting function that comes along with the data. More awkward is that loops over data space become two loops, one over tracks and one within a track.

Index