Chapter 1

Noisy data

Noise comes in two distinct flavors. First is erratic bursty noise which is difficult to fit into a statistical model. It bursts out of our simple models. To handle this noise we need "robust" estimation procedures which we consider first.

Next is noise that has a characteristic spectrum, temporal spectrum, spatial spectrum, or dip spectrum. Low frequency drift of the mean value of a signal is often called secular noise.

In real life, we need to handle both bursty noise and secular noise at the same time.

1.1 MEANS, MEDIANS, PERCENTILES AND MODES

Means, **median**s, and **mode**s are different averages. Given some data values d_i for $i = 1, 2, ..., N$, the arithmetic mean value m_2 is

$$m_2 \quad = \quad \frac{1}{N} \sum_{i=1}^{N} d_i \tag{1.1}$$

It is useful to notice that this m_2 is the solution of the simple fitting problem $d_i \approx m_2$ or $\mathbf{d} \approx m_2$, in other words, $\min_{m_2} \sum_i (m_2 - d_i)^2$ or

$$0 \quad = \quad \frac{d}{dm_2} \sum_{i=1}^{N} (m_2 - d_i)^2 \tag{1.2}$$

The median of the d_i values is found when the values are sorted from smallest to largest and then the value in the middle is selected. The median is delightfully well behaved even if some of your data values happen to be near infinity. Analytically, the median arises from the optimization

$$\min_{m_1} \sum_{i=1}^{N} |m_1 - d_i| \tag{1.3}$$

To see why, notice that the derivative of the absolute value function is the signum function,

$$\mathrm{sgn}(x) \quad = \quad \lim_{\epsilon \longrightarrow 0} \frac{x}{|x|+\epsilon} \tag{1.4}$$

The gradient vanishes at the minimum.

$$0 \quad = \quad \frac{d}{dm_1} \sum_{i=1}^{N} |m_1 - d_i| \tag{1.5}$$

The derivative is easy and the result is a sum of sgn() functions,

$$0 \quad = \quad \sum_{i=1}^{N} \mathrm{sgn}(m_1 - d_i) \tag{1.6}$$

In other words it is a sum of plus and minus ones. If the sum is to vanish, the number of plus ones must equal the number of minus ones. Thus m_1 is greater than half the data values and less than the other half, which is the definition of a median. The mean is said to minimize the L_2 norm of the residual and the median is said to minimize its L_1 norm.

Before this chapter, our model building was all based on the L_2 norm. The median is clearly a good idea for data containing large bursts of noise, but the median is a single value while geophysical models are made from many unknown elements. The L_1 norm offers us the new opportunity to build multiparameter models where the data includes huge bursts of noise.

Yet another average is the "**mode**," which is the most commonly occurring value. For example, in the number sequence $(1,2,3,5,5)$ the mode is 5 because it occurs the most times. Mathematically, the mode minimizes the zero norm of the residual, namely $L_0 = |m_0 - d_i|^0$. To see why, notice that when we raise a residual to the zero power, the result is 0 if $d_i = m_0$, and it is 1 if $d_i \neq m_0$. Thus, the L_0 sum of the residuals is the total number of residuals less those for which d_i matches m_0. The minimum of $L_0(m)$ is the mode $m = m_0$. The zero power function is nondifferentiable at the place of interest so we do not look at the gradient.

$L_2(m)$ and $L_1(m)$ are convex functions of m (positive second derivative for all m), and this fact leads to the triangle inequalities $L_p(a) + L_p(b) \geq L_p(a+b)$ for $p \geq 1$ and assures slopes lead to a unique (if $p > 1$) bottom. Because there is no triangle inequality for L_0, it should not be called a "norm" but a "measure."

Because most values are at the mode, the mode is where a probability function is maximum. The mode occurs with the maximum likelihood. It is awkward to contemplate the mode for floating-point values where the probability is minuscule (and irrelevant) that any two values are identical. A more natural concept is to think of the mode as the bin containing the most values.

1.1.1 Percentiles and Hoare's algorithm

The median is the 50-th **percentile**. After residuals are ordered from smallest to largest, the 90-th percentile is the value with 10% of the values above and 90% below. At SEP the default

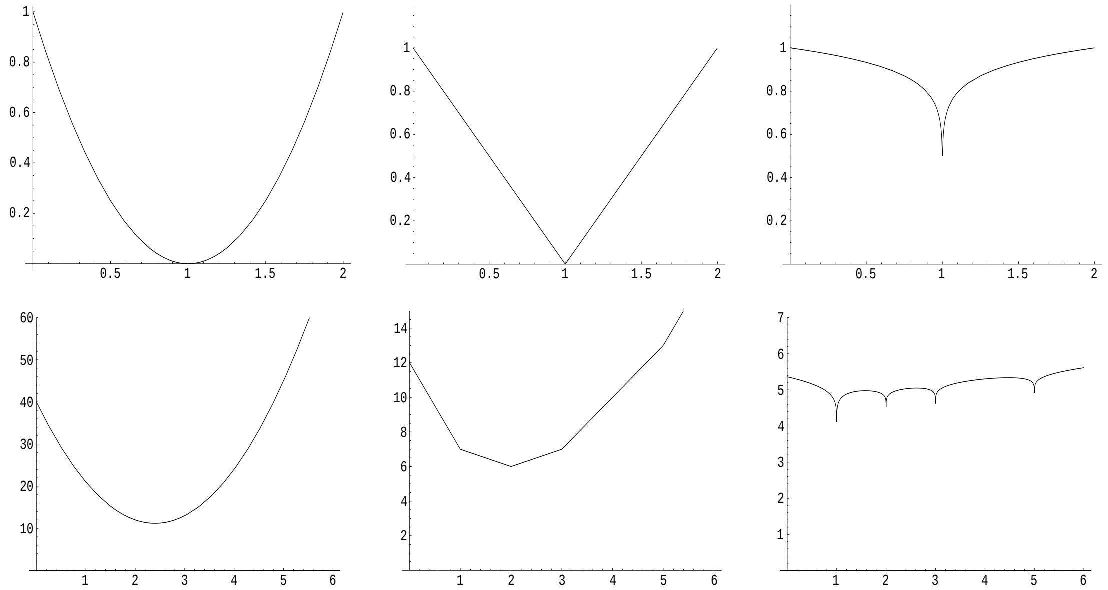


Figure 1.1: Mean, median, and mode. The coordinate is m. Top is the L_2, L_1, and $L_{1/10} \approx L_0$ measures of $m-1$. Bottom is the same measures of the data set $(1,1,2,3,5)$. $\boxed{\text{noiz-norms}}$ [CR]

value for clipping plots of field data is at the 98th percentile. In other words, magnitudes above the 98-th percentile are plotted at the 98-th percentile. Any percentile is most easily defined if the population of values a_i, for $i = 1,2,...,n$ has been sorted into order so that $a_i \leq a_{i+1}$ for all i. Then the 90-th percentile is a_k where $k = (90n)/100$.

We can save much work by using **Hoare's algorithm** which does not fully order the whole list, only enough of it to find the desired quantile. Hoare's algorithm is an outstanding example of the power of a recursive function, a function that calls itself. The main idea is this: We start by selecting a random value taken from the list of numbers. Then we split the list into two piles, one pile all values greater than the selected, the other pile all less. The quantile is in one of these piles, and by looking at their sizes, we know which one. So we repeat the process on that pile and ignore the other other one. Eventually the pile size reduces to one, and we have the answer.

In Fortran 77 or C it would be natural to split the list into two piles as follows:

> We divide the list of numbers into two groups, a group below a_k and another group above a_k. This reordering can be done "in place." Start one pointer at the top of the list and another at the bottom. Grab an arbitrary value from the list (such as the current value of a_k). March the two pointers towards each other until you have an upper value out of order with a_k and a lower value out of order with a_k. Swap the upper and lower value. Continue until the pointers merge somewhere midlist. Now you have divided the list into two sublists, one above your (random) value a_k and the other below.

Fortran 90 has some higher level intrinsic vector functions that simplify matters. When `a` is a vector and `ak` is a value, `a>ak` is a vector of logical values that are true for each component that is larger than `ak`. The integer count of how many components of `a` are larger than `ak` is given by the Fortran intrinsic function `count(a>ak)`. A vector containing only values less than `ak` is given by the Fortran intrinsic function `pack(a,a<ak)`.

Theoretically about $2n$ comparisons are expected to find the median of a list of n values. The code below (from Sergey Fomel) for this task is `quantile`.

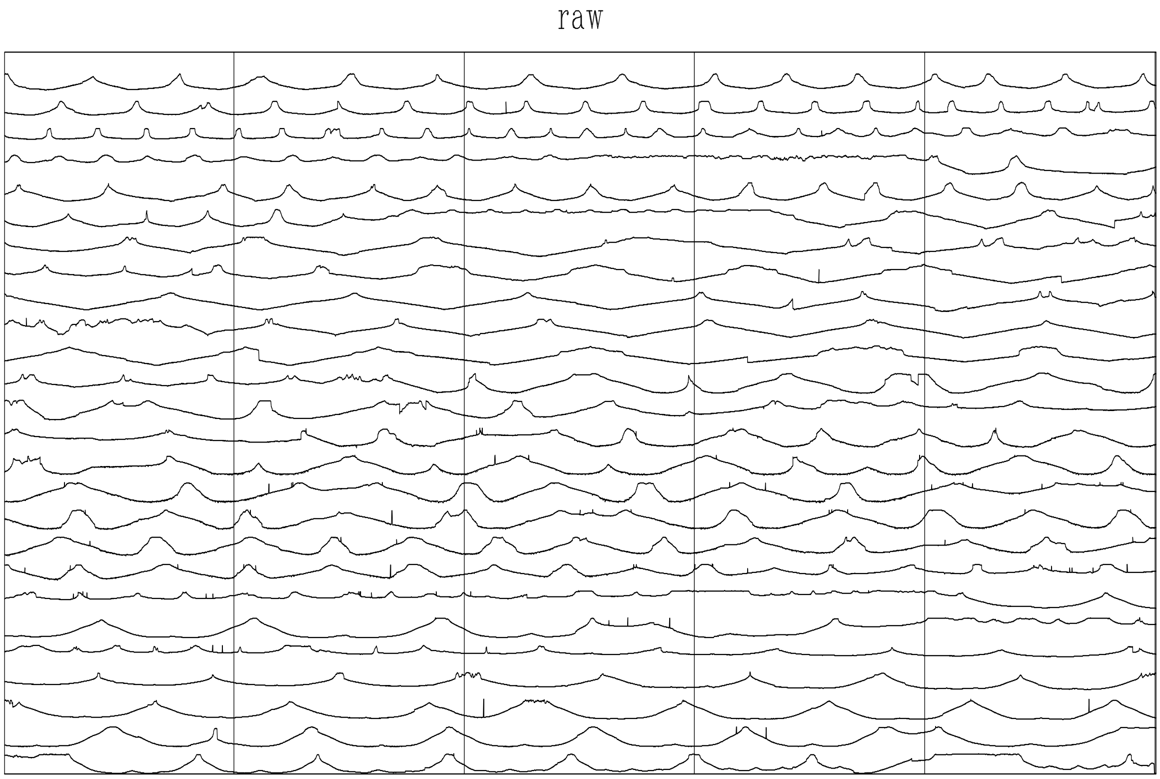
```
module quantile_mod {    # quantile finder.    median = quantile( size(a)/2, a)
contains
  recursive function quantile( k, a) result( value) {
    integer,             intent (in)  :: k          # position in array
    real, dimension (:), intent (in)  :: a
    real                              :: value      # output value of quantile
    integer                           :: j
    real                              :: ak
    ak = a( k)
    j = count( a < ak)                              # how many a(:) < ak
    if( j >= k)
                value = quantile( k, pack( a, a < ak))
    else {
        j = count( a > ak) + k - size( a)
        if( j > 0)
                value = quantile( j, pack( a, a > ak))
        else
                value = ak
        }
    }
}
```

An interesting application of medians is eliminating **noise spikes** through the use of a running median. A **running median** is a median computed in a moving window. Figure 1.2 shows depth-sounding data from the Sea of Galilee before and after a running median of 5 points was applied. The data as I received it is 132044 triples; i.e., (x_i, y_i, z_i) where i is measured along the vessel's track. In Figure 1.2 the depth data z_i appears as one long track although the surveying was done in several episodes that do not always continue the same track. For Figure 1.2 I first abandoned the last 2044 of the 132044 triples and all the (x_i, y_i)-pairs. Then I broke the remaining long signal into the 26 strips you see in the figure. Typically the depth is a "U"-shaped function as the vessel crosses the lake. You will notice that many spikes are missing on the bottom plot. For more about these tracks, see Figure 1.12.

1.1.2 The weighted mean

The **weighted mean** m is

$$m \quad = \quad \frac{\sum_{i=1}^{N} \; w_i^2 d_i}{\sum_{i=1}^{N} \; w_i^2} \tag{1.7}$$



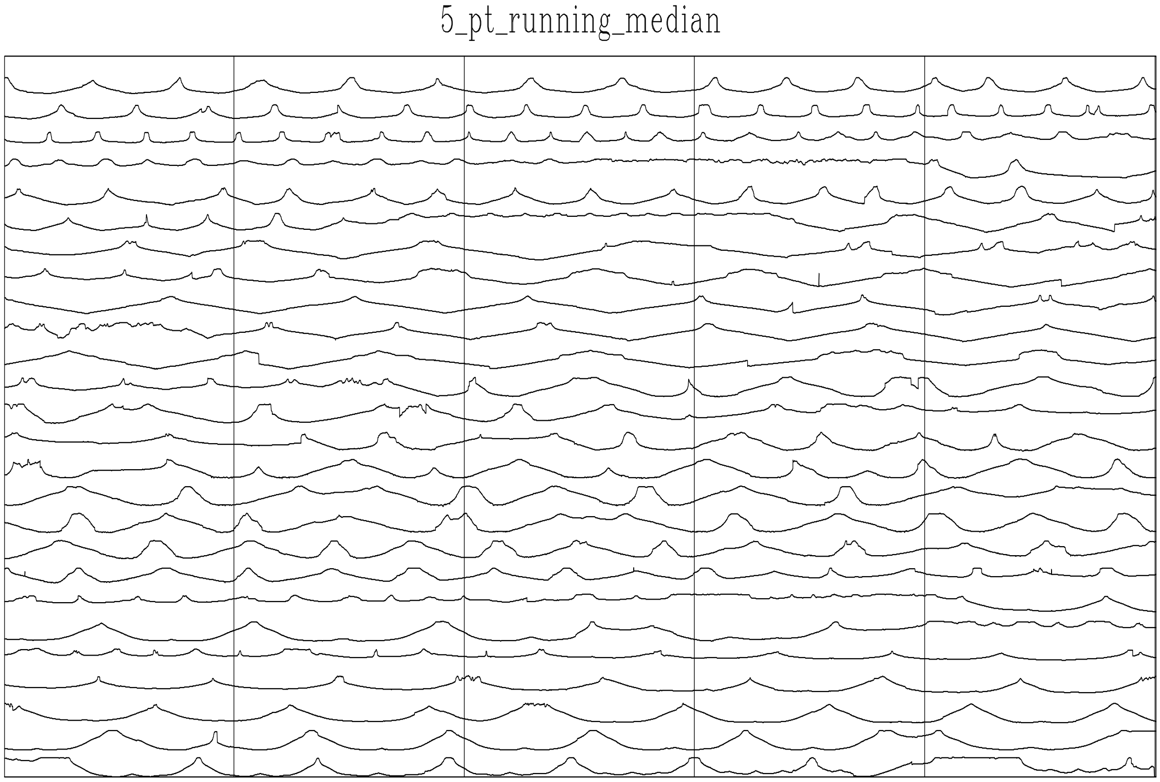


Figure 1.2: Depth of the Sea of Galilee along the vessel's track. noiz-median590 [ER,M]

where $w_i^2 > 0$ is the squared weighting function. This is the solution to the L_2 fitting problem $0 \approx w_i(m - d_i)$; in other words,

$$0 \quad = \quad \frac{d}{dm} \sum_{i=1}^{N} [w_i(m - d_i)]^2 \tag{1.8}$$

1.1.3 Weighted L.S. conjugate-direction template

The pseudocode for minimizing the *weighted* residual $\mathbf{0} \approx \mathbf{r} = \mathbf{W}(\mathbf{Fm} - \mathbf{d})$ by conjugate-direction method, is effectively like that for the unweighted method except that the initial residual is weighted and the operator $\mathbf{F}$ has the premultiplier $\mathbf{W}$. Naturally, the adjoint operator $\mathbf{F}'$ has the postmultiplier $\mathbf{W}'$. In some applications the weighting operator $\mathbf{W}$ is simply a weighting function or diagonal matrix (so then $\mathbf{W} = \mathbf{W}'$) and in other applications, the weighting operator $\mathbf{W}$ may be an operator, like the derivative along a data recording trajectory (so then $\mathbf{W} \neq \mathbf{W}'$).

$$
\begin{array}{lll}
\mathbf{r} & \longleftarrow & \mathbf{W}(\mathbf{Fm} - \mathbf{d}) \\
\text{iterate } \{ & & \\
 & \begin{array}{lll}
 \Delta\mathbf{m} & \longleftarrow & \mathbf{F}'\mathbf{W}'\ \mathbf{r} \\
 \Delta\mathbf{r} & \longleftarrow & \mathbf{WF}\ \Delta\mathbf{m} \\
 (\mathbf{m}, \mathbf{r}) & \longleftarrow & \text{cgstep}(\mathbf{m}, \mathbf{r}, \Delta\mathbf{m}, \Delta\mathbf{r})
 \end{array} & \\
 & \} &
\end{array}
$$

1.1.4 Multivariate L_1 estimation by iterated reweighting

The easiest method of model fitting is linear least squares. This means minimizing the sums of squares of residuals (L_2). On the other hand, we often encounter huge noises and it is much safer to minimize the sums of absolute values of residuals (L_1). (The problem with L_0 is that there are multiple minima, so the gradient is not a sensible way to seek the deepest).

There exist specialized techniques for handling L_1 multivariate fitting problems. They should work better than the simple iterative reweighting outlined here.

A penalty function that ranges from L_2 to L_1, depending on the constant $\bar{r}$ is

$$E(\mathbf{r}) \quad = \quad \sum_i \left(\sqrt{1 + r_i^2/\bar{r}^2} - 1 \right) \tag{1.9}$$

Where $r_i/\bar{r}$ is small, the terms in the sum amount to $r_i^2/2\bar{r}^2$ and where $r_i^2/\bar{r}^2$ is large, the terms in the sum amount to $|r_i/\bar{r}|$. We define the residual as

$$r_i \quad = \quad \sum_j F_{ij} m_j - d_i \tag{1.10}$$

We will need

$$\frac{\partial r_i}{\partial m_k} \quad = \quad \sum_j F_{ij}\delta_{jk} \quad = \quad F_{ik} \tag{1.11}$$

where we briefly used the notation that δ_{jk} is 1 when $j = k$ and zero otherwise. Now, to let us find the descent direction $\Delta\mathbf{m}$, we will compute the k-th component g_k of the gradient $\mathbf{g}$. We have

$$g_k \quad = \quad \frac{\partial E}{\partial m_k} \quad = \quad \sum_i \frac{1}{\sqrt{1+r_i^2/\bar{r}^2}} \frac{r_i}{\bar{r}^2} \frac{\partial r_i}{\partial m_k} \tag{1.12}$$

$$\mathbf{g} \quad = \quad \Delta\mathbf{m} \quad = \quad \mathbf{F}' \,\mathbf{diag}\left(\frac{1}{\sqrt{1+r_i^2/\bar{r}^2}}\right)\mathbf{r} \tag{1.13}$$

where we have use the notation **diag**() to designate a diagonal matrix with its argument distributed along the diagonal.

Continuing, we notice that the new weighting of residuals has nothing to do with the linear relation between model perturbation and residual perturbation; that is, we retain the familiar relations, $\mathbf{r} = \mathbf{Fm} - \mathbf{d}$ and $\Delta\mathbf{r} = \mathbf{F}\Delta\mathbf{m}$.

In practice we have the question of how to choose $\bar{r}$. I suggest that $\bar{r}$ be proportional to median($|r_i|$) or some other percentile.

1.1.5 Nonlinear L.S. conjugate-direction template

Nonlinear optimization arises from two causes:

1. Nonlinear physics. The operator depends upon the solution being attained.
2. Nonlinear statistics. We need robust estimators like the L_1 norm.

The computing template below is useful in both cases. It is almost the same as the template for weighted linear least-squares except that the residual is recomputed at each iteration. Starting from the usual weighted least-squares template we simply move the iteration statement a bit earlier.

iterate {

$$\begin{array}{lcl}
\mathbf{r} & \longleftarrow & \mathbf{Fm} - \mathbf{d} \\
\mathbf{W} & \longleftarrow & \mathbf{diag}[w(\mathbf{r})] \\
\mathbf{r} & \longleftarrow & \mathbf{Wr} \\
\Delta\mathbf{m} & \longleftarrow & \mathbf{F}'\mathbf{W}'\,\mathbf{r} \\
\Delta\mathbf{r} & \longleftarrow & \mathbf{WF}\,\Delta\mathbf{m} \\
(\mathbf{m}, \mathbf{r}) & \longleftarrow & \text{cgstep}(\mathbf{m}, \mathbf{r}, \Delta\mathbf{m}, \Delta\mathbf{r})
\end{array}$$

}

where $\mathbf{diag}[w(\mathbf{r})]$ is whatever weighting function we choose along the diagonal of a diagonal matrix.

Now let us see how the weighting functions relate to robust estimation: Notice in the code template that $\mathbf{W}$ is applied twice in the definition of $\Delta\mathbf{m}$. Thus $\mathbf{W}$ is the square root of the diagonal operator in equation (1.13).

$$\mathbf{W} \quad = \quad \mathbf{diag}\left(\frac{1}{\sqrt{\sqrt{1+r_i^2/\bar{r}^2}}}\right) \tag{1.14}$$

Module `weight_solver` on this page implements the computational template above. In addition to the usual set of arguments from the `solver()` subroutine on page ??, it accepts a user-defined function (parameter `wght`) for computing residual weights. Parameters `nmem` and `nfreq` control the restarting schedule of the iterative scheme.

```
module weight_solver {
  logical, parameter, private  :: T = .true., F = .false.
contains
  subroutine solver( oper, solv, x, dat, niter, nmem, nfreq, wght) {
    interface {
       integer function wght( res, w) {
           real, dimension (:)  :: res, w
           }
       integer function oper( adj, add, x, dat) {
           logical, intent (in) :: adj, add
           real, dimension (:)  :: x, dat
           }
       integer function solv( forget, x, g, rr, gg) {
           logical             :: forget
           real, dimension (:) :: x, g, rr, gg
           }
       }
    real, dimension (:), intent (in)  :: dat                # data
    real, dimension (:), intent (out) :: x                  # solution
    integer,             intent (in)  :: niter, nmem, nfreq # iterations
    real, dimension (size (x))        :: g                  # gradient
    real, dimension (size (dat))      :: rr, gg, wt         # res, CG, weight
    integer                           :: i, stat
    logical                           :: forget
    rr = -dat;  x = 0.;  wt = 1.;  forget = F         # initialize
    do i = 1, niter {
       forget = (i > nmem)                            # restart
       if( forget) stat = wght( rr, wt)               # compute weighting
       rr = rr * wt                                   # rr = W (Fx - d)
       stat = oper( T, F,  g, wt*rr)                  # g  = F' W' rr
       stat = oper( F, F,  g, gg)                     # gg = Fg
       gg = gg * wt                                   # gg = W F g
       if( forget)  forget = ( mod( i, nfreq) == 0)   # periodic restart
       stat = solv( forget, x, g, rr, gg)             # step in x and rr
       rr = - dat
       stat = oper( F, T, x, rr)                      # rr = Fx - d
```

```
    }
  }
}
```

We can ask whether `cgstep()`, which was not designed with nonlinear least-squares in mind, is doing the right thing with the weighting function. First, we know we are doing weighted linear least-squares correctly. Then we recall that on the first iteration, the conjugate-directions technique reduces to steepest descent, which amounts to a calculation of the scale factor α with

$$\alpha \quad = \quad - \frac{\Delta\mathbf{r}\cdot\mathbf{r}}{\Delta\mathbf{r}\cdot\Delta\mathbf{r}} \tag{1.15}$$

Of course, `cgstep()` knows nothing about the weighting function, but notice that the iteration loop above nicely inserts the weighting function both in $\mathbf{r}$ and in $\Delta\mathbf{r}$, as required by (1.15).

Experience shows that difficulties arise when the weighting function varies rapidly from one iteration to the next. Naturally, the conjugate-direction method, which remembers the previous iteration, will have an inappropriate memory if the weighting function changes too rapidly. A practical approach is to be sure the changes in the weighting function are slowly variable.

1.1.6 Minimizing the Cauchy function

A good trick (I discovered accidentally) is to use the weight

$$\mathbf{W} \quad = \quad \mathbf{diag}\left(\frac{1}{\sqrt{1+r_i^2/\bar{r}^2}}\right) \tag{1.16}$$

Sergey Fomel points out that this weight arises from minimizing the **Cauchy function**:

$$E(\mathbf{r}) \quad = \quad \sum_i \log(1+r_i^2/\bar{r}^2) \tag{1.17}$$

A plot of this function is found in Figure 1.3.

Because the second derivative is not positive everywhere, the Cauchy function introduces the possibility of multiple solutions, but because of the good results we see in Figure 1.4, you might like to try it anyway. Perhaps the reason it seems to work so well is that it uses mostly residuals of "average size," not the big ones or the small ones. This happens because $\Delta\mathbf{m}$ is made from $\mathbf{F}'$ and the components of $\mathbf{W}^2\mathbf{r}$ which are a function $r_i/(1+r_i^2/\bar{r}^2)$ that is maximum for those residuals near $\bar{r}$.

Module `irls` on this page supplies two useful weighting functions that can be interchanged as arguments to the reweighted scheme on the preceding page.

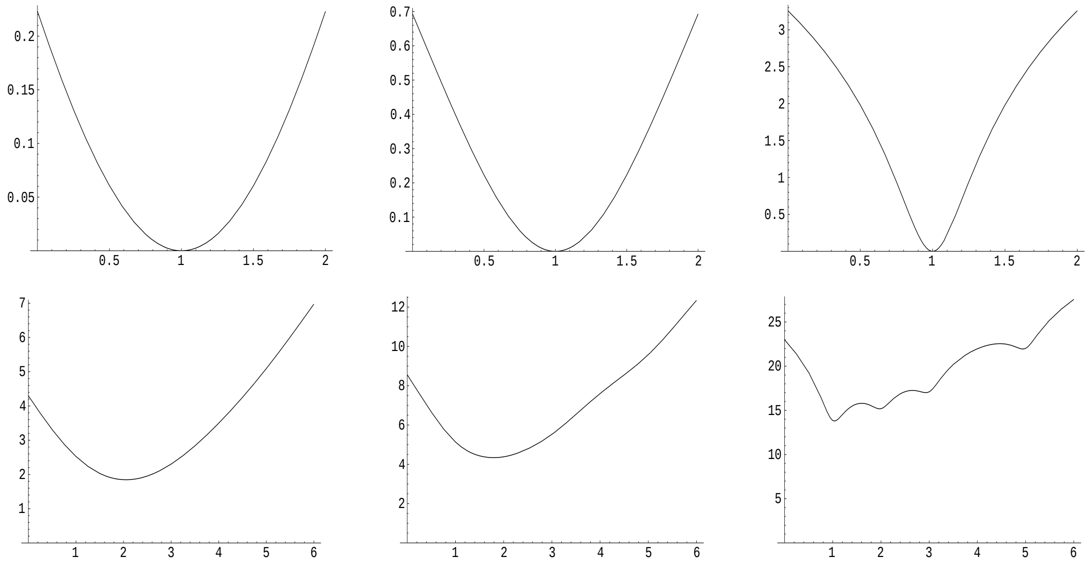


Figure 1.3: The coordinate is m. Top is Cauchy measures of $m-1$. Bottom is the same measures of the data set $(1,1,2,3,5)$. Left, center, and right are for $\bar{r} = (2,1,.2)$. noiz-cauchy [CR]

```
module irls {
  use quantile_mod
contains
  integer function l1 (res, weight)  {
    real, dimension (:)  :: res, weight
    real                 :: rbar
    rbar = quantile( int( 0.5*size(res)), abs (res))            # median
    weight = 1. / sqrt( sqrt (1. + (res/rbar)**2));   l1 = 0
    }
  integer function cauchy (res, weight)  {
    real, dimension (:)  :: res, weight
    real                 :: rbar
    rbar = quantile( int( 0.5*size(res)), abs (res))             # median
    weight = 1. / sqrt (1. + (res/rbar)**2);          cauchy = 0
    }
}
```

1.2 NOISE BURSTS

Sometimes noise comes in isolated **spikes**. Sometimes it comes in **bursts** or bunches (like grapes). Figure 1.4 is a simple one-dimensional example of a periodic signal plus spikes and bursts. Three processes are applied to this data, **despike** and two flavors of **deburst**. Here we will examine the processes used. (For even better results, see Figure 1.6.)

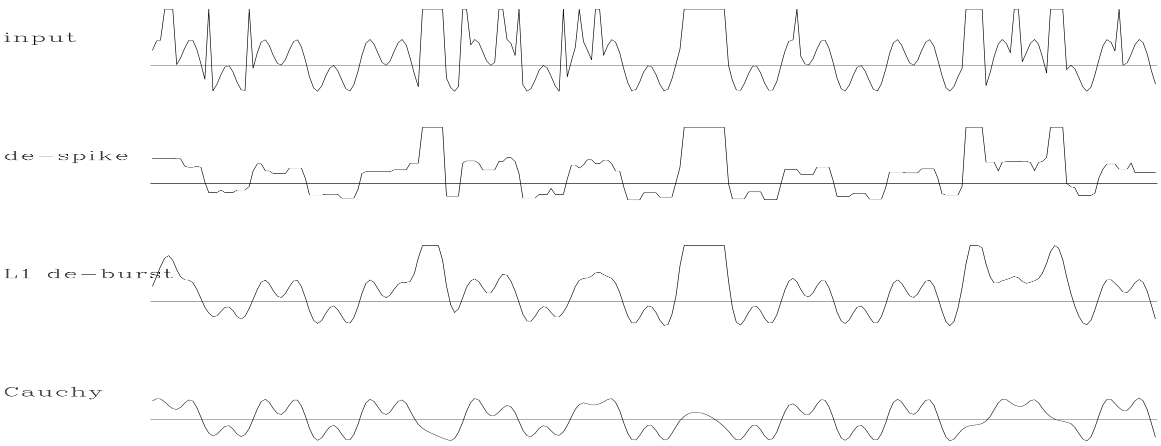


Figure 1.4: Top is synthetic data with noise spikes and bursts. (Most bursts are a hundred times larger than shown.) Next is after running medians. Bottom is after the two processes described here. noiz-burst90 [ER]

1.2.1 De-spiking with median smoothing

The easiest method to remove spikes is to pass a moving window across the data and output the median value in the window. This method of despiking was done in Figure 1.4, which shows a problem of the method: The window is not long enough to clean the long bursts, but it is already so long that it distorts the signal by flattening its peaks. The window size really should not be chosen in advance but should depend upon by what is encountered on the data. This I have not done because the long-burst problem is solved by another method described next.

1.2.2 De-bursting

Most signals are smooth, but running medians assume they have no curvature. An alternate expression of this assumption is that the signal has minimal curvature $0 \approx h_{i+1} - 2h_i + h_{i-1}$; in other words, $\mathbf{0} \approx \nabla^2 \mathbf{h}$. Thus we propose to create the cleaned-up data $\mathbf{h}$ from the observed data $\mathbf{d}$ with the fitting problem

$$\begin{array}{lll} 0 & \approx & \mathbf{W}(\mathbf{h} - \mathbf{d}) \\ 0 & \approx & \epsilon \, \nabla^2 \mathbf{h} \end{array} \tag{1.18}$$

where $\mathbf{W}$ is a diagonal matrix with weights sprinkled along the diagonal, and where ∇^2 is a matrix with a roughener like $(1, -2, 1)$ distributed along the diagonal. This is shown in Figure 1.4 with $\epsilon = 1$. Experience showed similar performances for $0 \approx \nabla \mathbf{h}$ and $0 \approx \nabla^2 \mathbf{h}$. Better results, however, will be found later in Figure 1.6, where the ∇^2 operator is replaced by an operator designed to predict this very predictable signal.

1.3 MEDIAN BINNING

We usually add data into bins. When the data has erratic noise, we might prefer to take the median of the values in each bin. Subroutine `medianbin2()` (in the library, but not listed here) performs the chore. It is a little tricky because we first need to find out how many data values go into each bin, then we must allocate that space and copy each data value from its track location to its bin location. Finally we take the median in the bin. A small annoyance with medians is that when bins have an even number of points, like two, there no middle. To handle this problem, subroutine `medianbin2()` uses the average of the middle two points.

A useful byproduct of the calculation is the residual: For each data point its bin median is subtracted. The residual can be used to remove suspicious points before any traditional least-squares analysis is made. An overall strategy could be this: First a coarse binning with many points per bin, to identify suspicious data values, which are set aside. Then a sophisticated least squares analysis leading to a high-resolution depth model. If our search target is small, recalculate the residual with the high-resolution model and reexamine the suspicious data values.

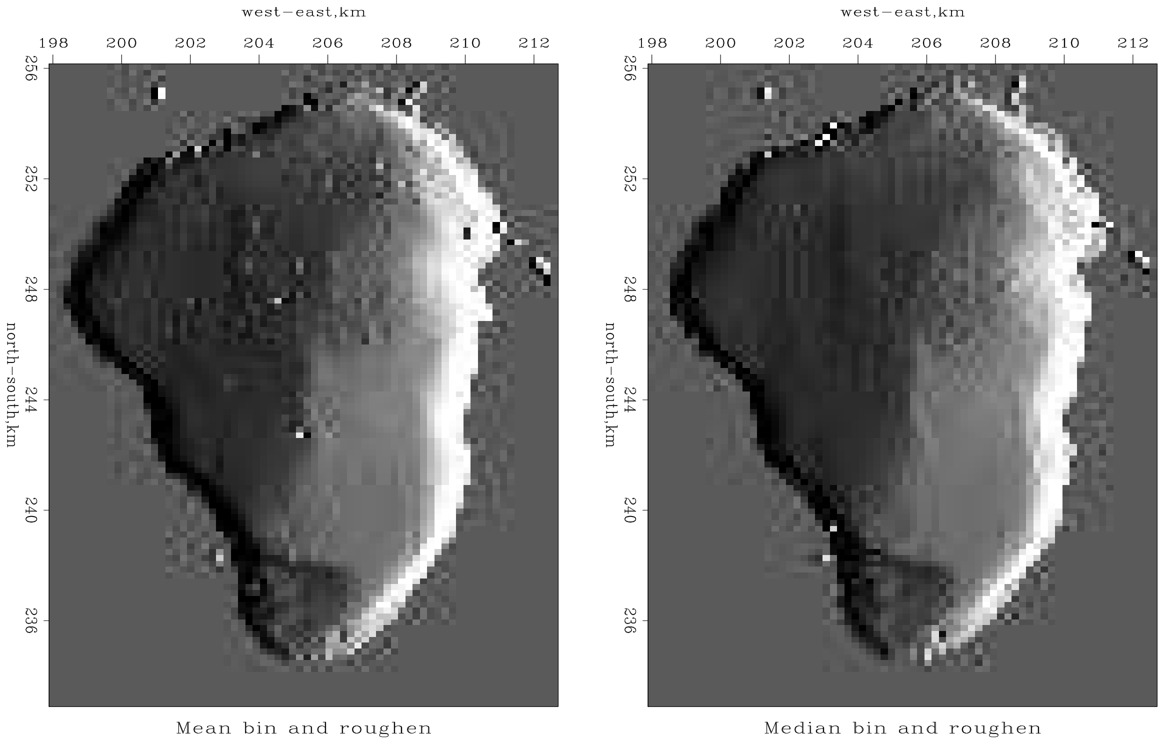


Figure 1.5: Galilee water depth binned and roughened. Left is binning with the mean, right with the median. noiz-medbin90 [ER,M]

Figure 1.5 compares the water depth in the Sea of Galilee with and without median binning. The difference does not seem great here but it is more significant than it looks. Later processing will distinguish between empty bins (containing an exact zero) and bins with small values in them. Because of the way the depth sounder works, it often records an erroneously near-zero depth. This will make a mess of our later processing (missing data fill) unless we cast out those data values. This was done by median binning in Figure 1.5 but the change is disguised by the many empty bins.

Median binning is a useful tool, but where bins are so small that they hold only one or two points, there the median for the bin is the same as the usual arithmetic average.

1.4 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 (**??**) 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). 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.

1.5 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 1.4. 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) {
    integer, intent( in)         :: niter
    real, dimension( :), pointer :: yy
    type( filter)                :: aa
    real, dimension( size( yy))  :: rr
```

```
    real                          :: rbar
    integer                       :: stat
    call helicon_init( aa)                           # starting guess
    stat = helicon_lop( .false., .false., yy, rr)
    rbar = quantile( size( yy)/3, abs( rr))          # rbar=(r safe below rbar)
    where( aa%mis)  yy = 0.
    call find_mask(( yy /= 0. .and. abs( rr) < 5 * rbar), aa)
    call find_pef( yy, aa, niter)
  }
}
```

The result of this "PEF-deburst" processing is shown in Figure 1.6.

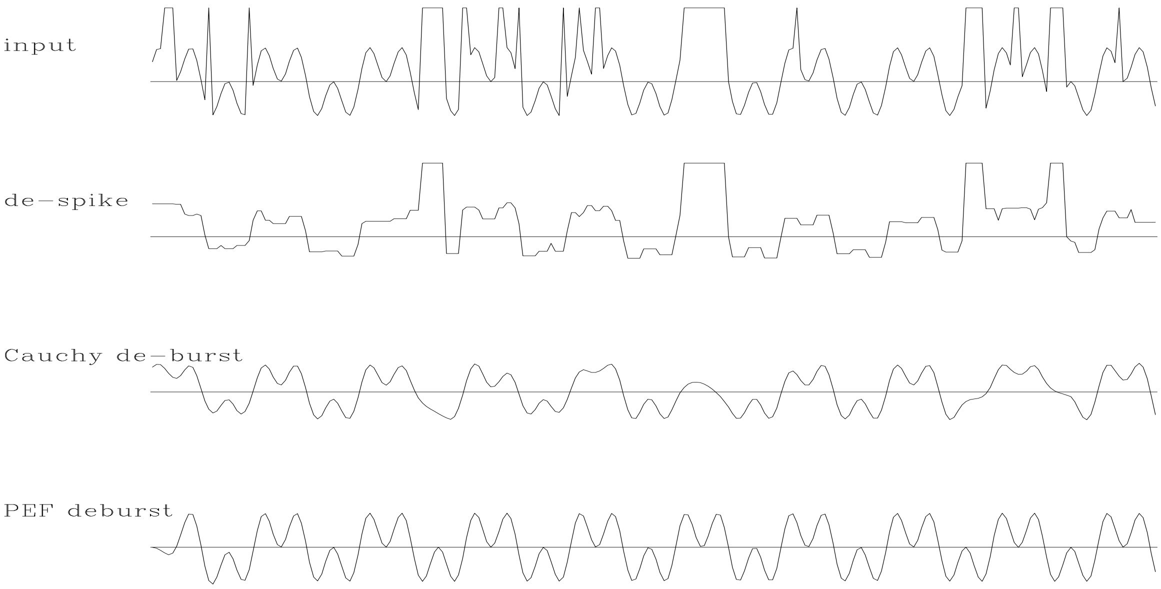


Figure 1.6: Top is synthetic data with noise spikes and bursts. (Some bursts are fifty times larger than shown.) Next is after running medians. Next is Laplacian filter Cauchy deburst processing. Last is PEF-deburst processing. noiz-pefdeburst90 [ER]

Given the PEF that comes out of `pefest1()`[1], 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 ?? is invoked to replace the zeroed values by reasonable ones.

```
module fixbad { # Given a PEF, find bad data and restore it.
  use mis2
  use helicon
  use quantile_mod
contains
  subroutine fixbad1 (niter, aa, yy) {
    integer,       intent (in)      :: niter
```

[1]If you are losing track of subroutines defined earlier, look at the top of the module to see what other modules it uses. Then look in the index to find page numbers of those modules.

```
    type( filter), intent (in)      :: aa
    real,    dimension (:)          :: yy
    real,    dimension (size (yy))  :: rr
    logical, dimension (size (yy))  :: known
    integer                         :: stat
    call helicon_init( aa)
    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, known, .true.)
 }
}
```

1.5.1 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).

1.6 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 1.7 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, we 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.

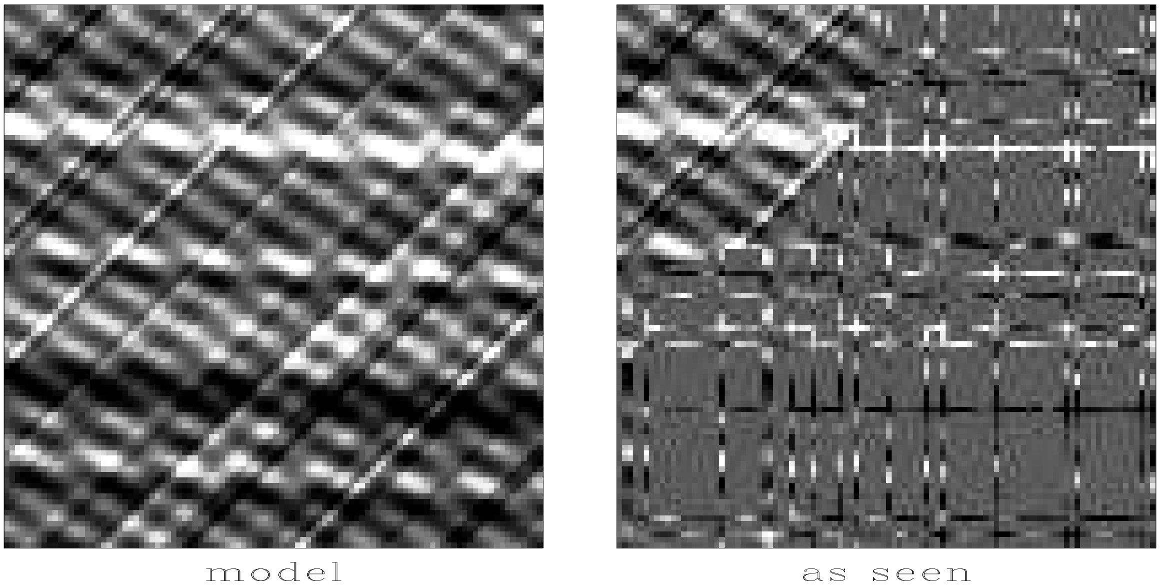


Figure 1.7: Synthetic wavefield (left) and as observed over survey lines (right). The wavefield is a superposition of waves from three directions. noiz-duelin90 [ER]

The set of test data shown in Figure 1.7 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 $\mathbf{U}$ be the east-west PE operator and $\mathbf{V}$ be the north-south operator and let the signal or image be $\mathbf{h} = h(x,y)$. The fitting residuals are

$$\begin{array}{lcl} \mathbf{0} & \approx & (\mathbf{I}-\mathbf{J})(\mathbf{h}-\mathbf{d}) \\ \mathbf{0} & \approx & \mathbf{U}\,\mathbf{h} \\ \mathbf{0} & \approx & \mathbf{V}\,\mathbf{h} \end{array} \tag{1.19}$$

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

Figure 1.8 shows the result of using a single one-dimensional PEF along either the vertical or the horizontal axis.

Figure 1.9 compares the use of a pair of 1-D PEFs versus a single 2-D PEF (which needs the "cheat" corner in Figure 1.7. Studying Figure 1.9 we conclude (what theory predicts) that

- These waves are predictable with a pair of 1-D filters:

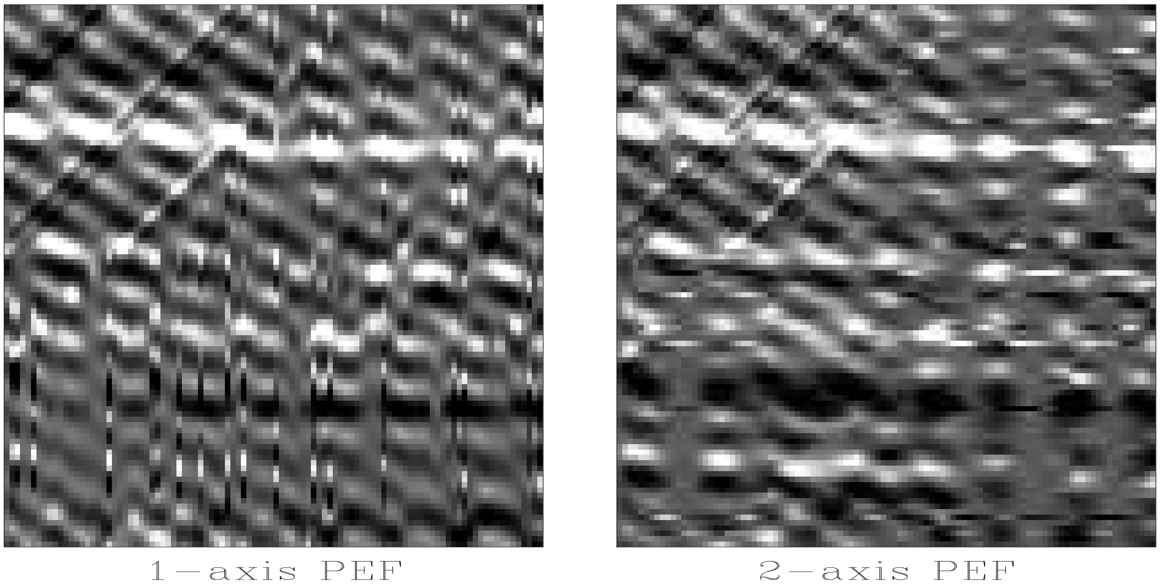


Figure 1.8: Interpolation by 1-D PEF along the vertical axis (left) and along the horizontal axis (right). noiz-dueleither90 [ER]

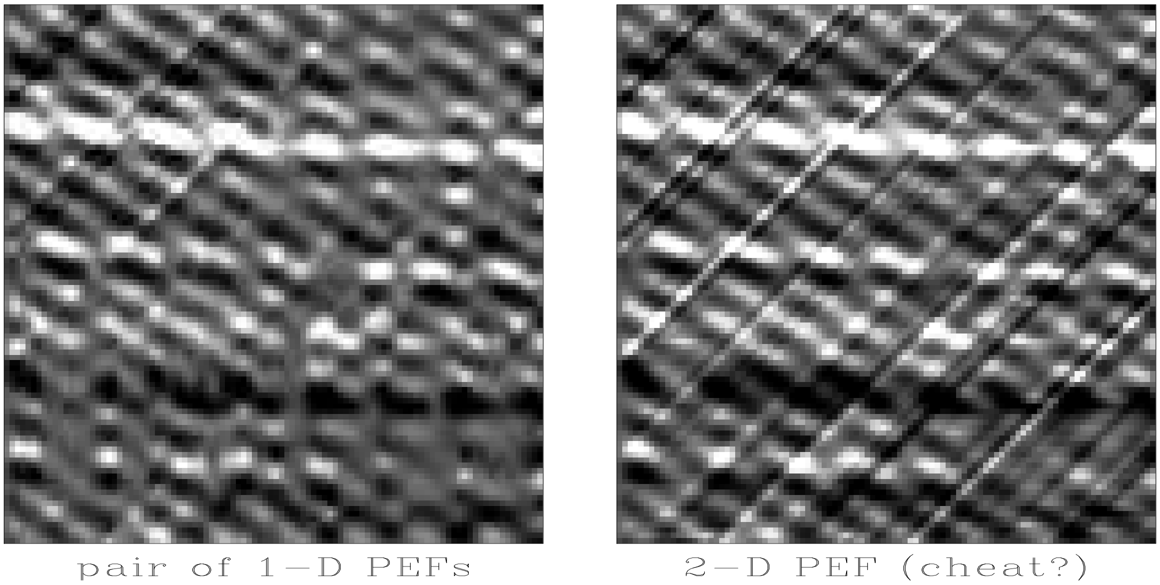


Figure 1.9: Data infilled by a pair of 1-D PEFs (left) and by a single 2-D PEF (right). noiz-duelversus90 [ER,M]

  - 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

1.7 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

$$\mathbf{0} \approx \frac{d}{dt}\,(\mathbf{h}-\mathbf{d}) \tag{1.20}$$

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.

1.7.1 Using a PEF on the Galilee residual

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

$$\begin{aligned} 0 &\approx \mathbf{A}(\mathbf{Bh}-\mathbf{d}) \\ 0 &\approx \epsilon\nabla\mathbf{h} \end{aligned} \tag{1.21}$$

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 weighting functions. Including such a weighting function gives us the map-fitting goals,

$$\begin{aligned} 0 &\approx \mathbf{WA}(\mathbf{Bh}-\mathbf{d}) \\ 0 &\approx \epsilon\nabla\mathbf{h} \end{aligned} \tag{1.22}$$

Results are in Figure 1.10. It is pleasing to see the ship's tracks gone at last.

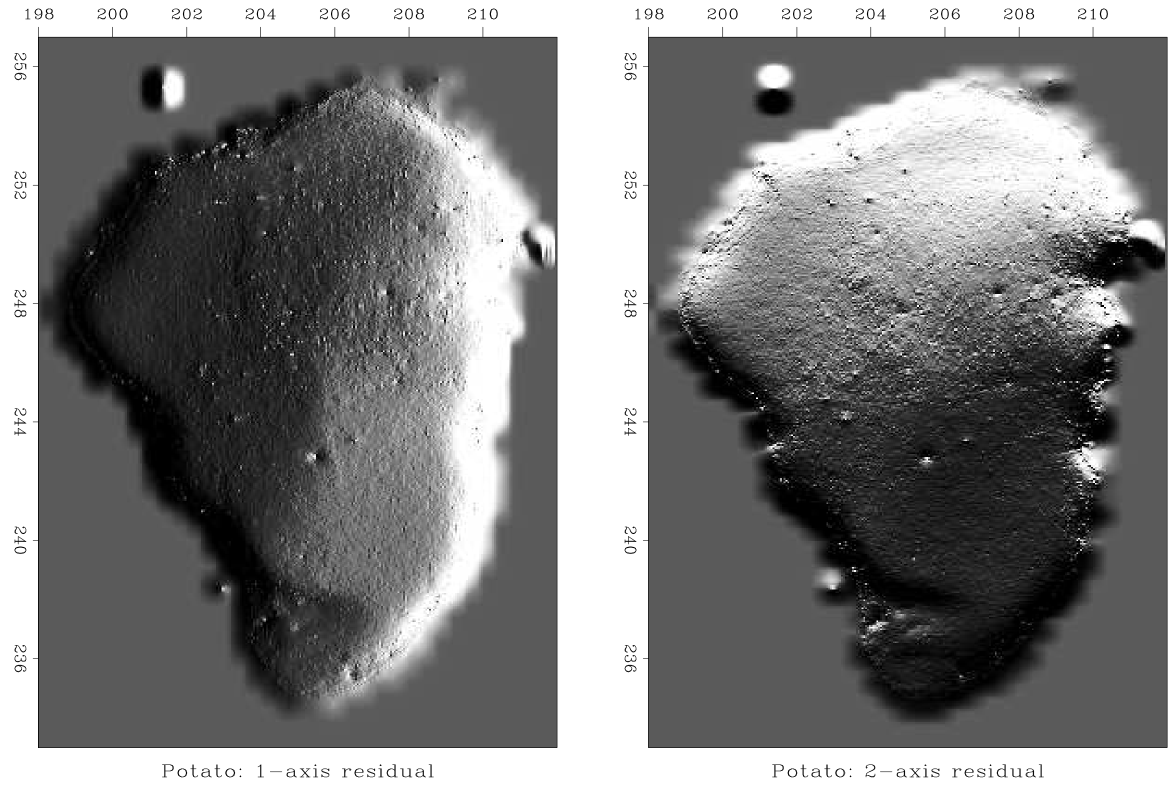


Figure 1.10: Gradient of the Galilee map from (1.22). $\boxed{\text{noiz-potato}}$ [ER,M]

1.7.2 PEFs on both model space and residual space

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

$$\begin{aligned} 0 &\approx \mathbf{WA}(\mathbf{Bh}-\mathbf{d}) \\ 0 &\approx \epsilon\mathbf{Uh} \\ 0 &\approx \epsilon\mathbf{Vh} \end{aligned} \tag{1.23}$$

I omit the display of my subroutine for the goals (1.23) 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 1.10 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 1.11 we see that this is no longer true. Thus although the topographic PEFs look similar to a gradient, the difference is substantial.

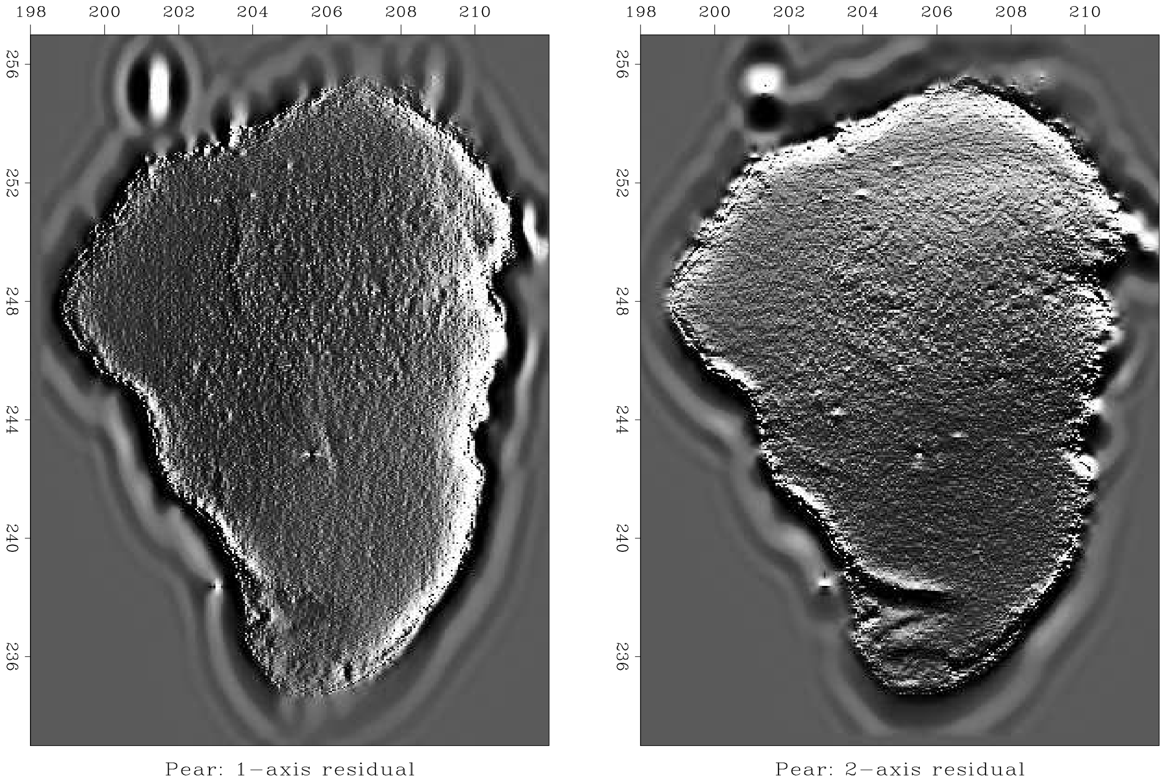


Figure 1.11: Galilee residuals estimated by (1.23). noiz-pear [ER,M]

Subjectively comparing Figures 1.10 and 1.11 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.

I have wondered whether any significant improvements might result from using linear interpolation **L** instead of simple binning **B**. The initialization arguments are identical in operator `lint2` on page 22 and operator `bin2` on page ??, so they are "plug compatible" and we could easily experiment.

1.7.3 Regridding

Because of the weighting **W**, which is a function of the residual itself, the fitting problems (1.22) and (1.23) 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 operator `lint2`. It is designed for data points at irregular locations, but we can use it for regular locations too. Operator `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.

```
module refine2 { # Refine mesh.
# Input mm(m1,m2) is coarse.  Output dd(n1,n2) linear interpolated.
#
use lint2
real, dimension( :, :), pointer, private :: xy
#% _init( co1,cd1,co2,cd2, m1,m2, fo1,fd1,fo2,fd2, n1,n2)
  integer, intent( in)  :: m1,m2, n1,n2
  real,    intent( in)  :: co1,cd1,co2,cd2                 # coarse grid
  real,    intent( out) :: fo1,fd1,fo2,fd2                 # fine grid
  integer               :: i1,i2, id
  real                  :: xmin,xmax, ymin,ymax, x,y
  allocate (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, :) = (/ x, y /)
        }}
  call lint2_init( m1,m2, co1,cd1, co2,cd2, xy)
#% _lop (mm, dd)
  integer stat1
  stat1 = lint2_lop( adj, .true., mm, dd)
#% _close
deallocate (xy)
}
```

Finally, here is the 2-D linear interpolation operator `lint2`, which is a trivial extension of the 1-D version `lint1` on page ??.

```
module lint2 {                                    # (Bi)Linear interpolation in 2-
D
integer                        :: m1,m2
real                           :: o1,d1, o2,d2
real, dimension (:,:), pointer :: xy
#%  _init (     m1,m2, o1,d1, o2,d2, xy)
#%  _lop  ( mm (m1,m2), dd (:))
integer i, ix,iy, id
real    f, fx,gx, fy,gy
do id= 1, size(dd) {
   f = (xy(id,1)-o1)/d1; i=f; ix= 1+i; if( 1>ix .or. ix>=m1) cycle; fx=f-i; gx= 1.-
fx
   f = (xy(id,2)-o2)/d2; i=f; iy= 1+i; if( 1>iy .or. iy>=m2) cycle; fy=f-i; gy= 1.-
fy
                if( adj) {
                        mm(ix  ,iy  ) += gx * gy * dd(id)
                        mm(ix+1,iy  ) += fx * gy * dd(id)
                        mm(ix  ,iy+1) += gx * fy * dd(id)
                        mm(ix+1,iy+1) += fx * fy * dd(id)
                        }
                else
                        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)
   }
}
```

1.7.4 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 1.10 and 1.11 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 1.12 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.

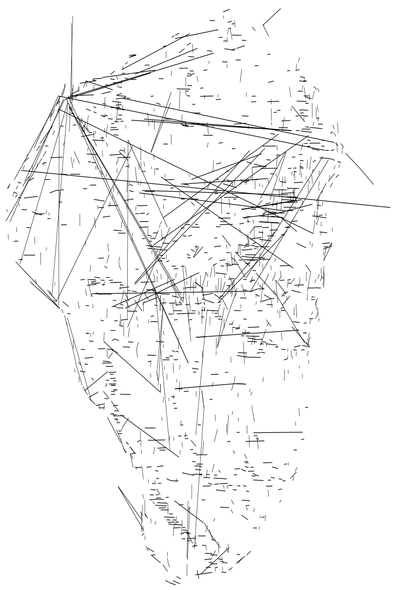


Figure 1.12: The lines are the gaps between successive vessel tracks, i.e., where the vessel turned off the depth sounder. At the end of each track is a potential glitch. noiz-seegap [ER]

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 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