Chapter 7

Nasty noise and robust estimation

Systematic noises on data should be handled by systematic choices of weighting functions. On the other hand, we often encounter nasty unsystematic noises. To handle these, we need "robust" estimation procedures.

7.1 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

$$m_2 \quad = \quad \frac{1}{N} \sum_{i=1}^{N} d_i \tag{7.1}$$

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

$$0 \quad = \quad \frac{d}{dm_2} \sum_{i=1}^{N} (m_2 - d_i)^2 \tag{7.2}$$

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

$$\min_{m_1} \sum_{i=1}^{N} |m_1 - d_i| \tag{7.3}$$

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

$$\operatorname{sgn}(x) \quad = \quad \lim_{\epsilon \longrightarrow 0} \frac{x}{|x| + \epsilon} \tag{7.4}$$

The gradient vanishes at the minimum.

$$0 \quad = \quad \frac{d}{dm_1} \sum_{i=1}^{N} |m_1 - d_i| \tag{7.5}$$

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

$$0 \quad = \quad \sum_{i=1}^{N} \operatorname{sgn}(m_1 - d_i) \tag{7.6}$$

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.

7.1.1 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.

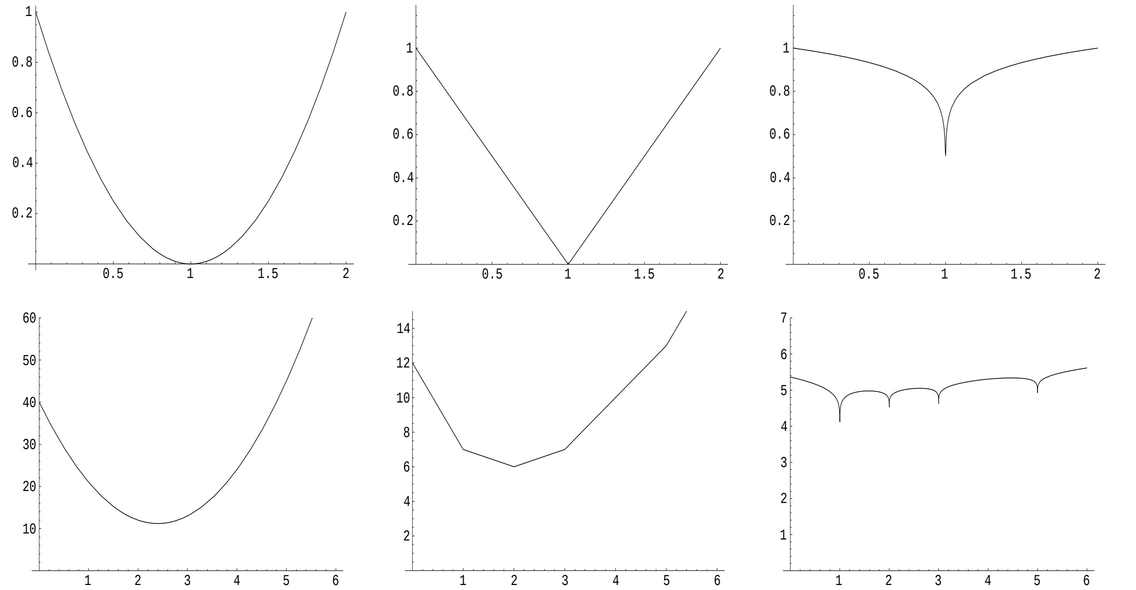

Figure 7.1: The coordinate is m. Top is the L_2, L_1, and $L_{1/10} \approx L_0$ measures of $m-1$. Bottom is the same measures of the data set $(1, 1, 2, 3, 5)$. $\boxed{\text{rbst-norms}}$ [CR]

At SEP the default 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 8.9.

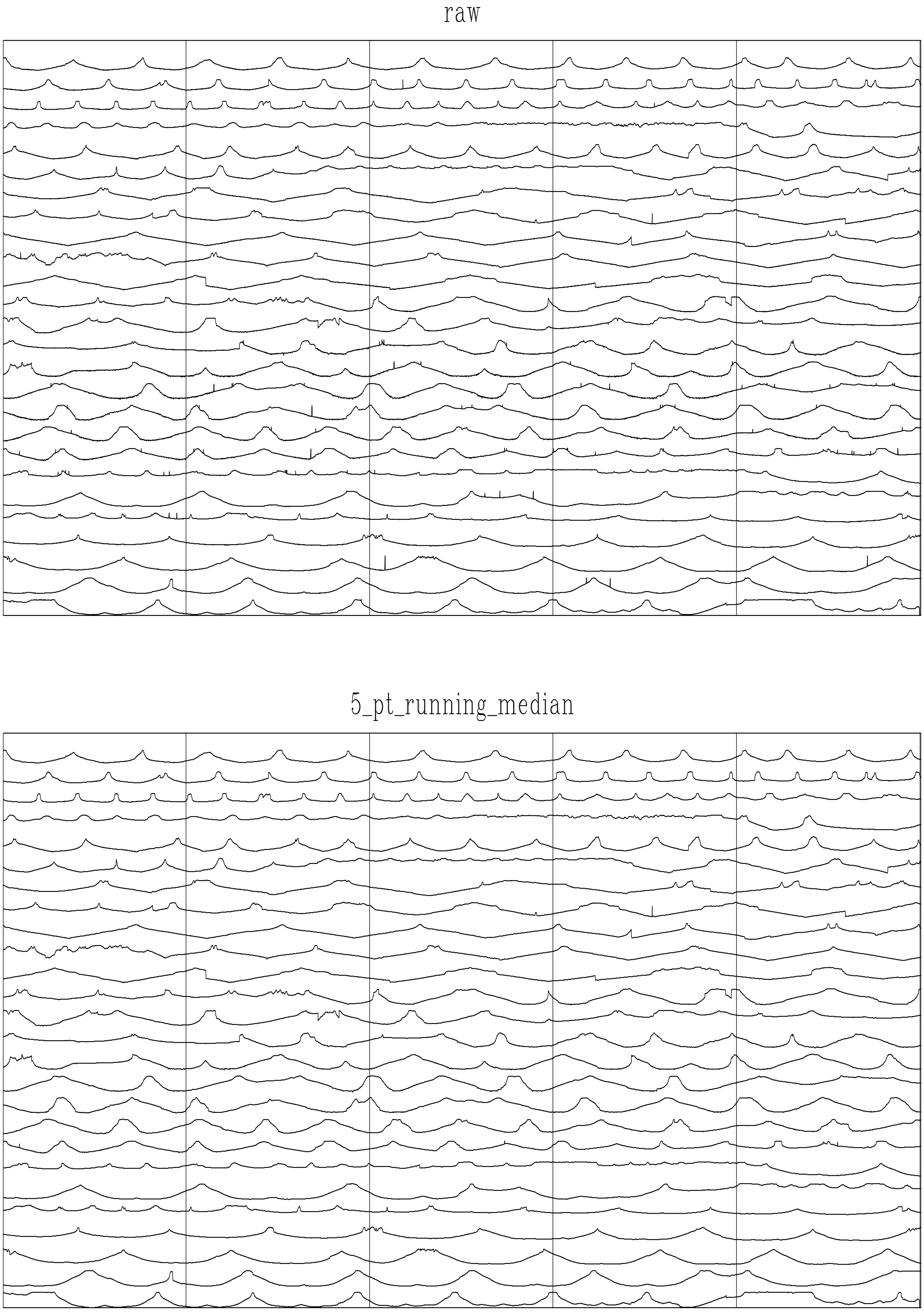

Figure 7.2: Depth of the Sea of Galilee along the vessel's track. rbst-median590 [ER]

7.1.2 The weighted mean

The **weighted mean** m is

$$m \quad = \quad \frac{\sum_{i=1}^N \; w_i^2 d_i}{\sum_{i=1}^N \; w_i^2} \tag{7.7}$$

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,

$$0 \quad = \quad \frac{d}{dm} \sum_{i=1}^N \left[w_i(m - d_i) \right]^2 \tag{7.8}$$

7.1.3 Weighted L.S. conjugate-direction template

The pseudocode for minimizing the *weighted* residual $\mathbf{0} \approx \mathbf{r} = \mathbf{W}(\mathbf{F}\mathbf{m} - \mathbf{d})$ 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 $\mathbf{W} \neq \mathbf{W}'$).

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

7.1.4 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

$$E(\mathbf{r}) \quad = \quad \sum_i \left(\sqrt{1 + r_i^2/\bar{r}^2} - 1 \right) \tag{7.9}$$

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

$$r_i \quad = \quad \sum_j F_{ij} m_j - d_i \tag{7.10}$$

We will need

$$\frac{\partial r_i}{\partial m_k} \quad = \quad \sum_j F_{ij} \delta_{jk} \quad = \quad F_{ik} \tag{7.11}$$

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

$$g_k \quad = \quad \frac{\partial E}{\partial m_k} \quad = \quad \sum_i \frac{1}{\sqrt{1 + r_i^2/\bar{r}^2}} \frac{r_i}{\bar{r}^2} \frac{\partial r_i}{\partial m_k} \tag{7.12}$$

$$\mathbf{g} \quad = \quad \Delta\mathbf{m} \quad = \quad \mathbf{F}' \; \mathbf{diag} \left(\frac{1}{\sqrt{1 + r_i^2/\bar{r}^2}} \right) \mathbf{r} \tag{7.13}$$

where we have use the notation $\mathbf{diag}()$ 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.

7.1.5 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 {

$$\begin{aligned} \mathbf{r} \quad &\longleftarrow \quad \mathbf{Fm} - \mathbf{d} \\ \mathbf{W} \quad &\longleftarrow \quad \mathbf{diag}[w(\mathbf{r})] \end{aligned}$$

$$
\begin{array}{lll}
\mathbf{r} & \longleftarrow & \mathbf{W}\mathbf{r} \\
\Delta\mathbf{m} & \longleftarrow & \mathbf{F}'\mathbf{W}'\,\mathbf{r} \\
\Delta\mathbf{r} & \longleftarrow & \mathbf{W}\mathbf{F}\,\Delta\mathbf{m} \\
(\mathbf{m},\mathbf{r}) & \longleftarrow & \mathrm{cgstep}(\mathbf{m},\mathbf{r},\Delta\mathbf{m},\Delta\mathbf{r}) \\
\}
\end{array}
$$

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 $\mathbf{\Delta m}$. Thus $\mathbf{W}$ is the square root of the diagonal operator in equation (1.13).

$$
\mathbf{W} \quad = \quad \mathbf{diag}\left(\frac{1}{\sqrt{\sqrt{1+r_i^2/\bar{r}^2}}}\right) \tag{7.14}
$$

Module `weight_solver` on page 8 implements the computational template above. In addition to the usual set of arguments from the `solver()` subroutine on page 47, 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

$$\alpha \quad = \quad - \frac{\Delta \mathbf{r} \cdot \mathbf{r}}{\Delta \mathbf{r} \cdot \Delta \mathbf{r}} \tag{7.15}$$

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.

7.1.6 Minimizing the Cauchy function

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

$$\mathbf{W} \quad = \quad \mathbf{diag}\left(\frac{1}{\sqrt{1 + r_i^2/\bar{r}^2}} \right) \tag{7.16}$$

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

$$E(\mathbf{r}) \quad = \quad \sum_i \log(1 + r_i^2/\bar{r}^2) \tag{7.17}$$

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

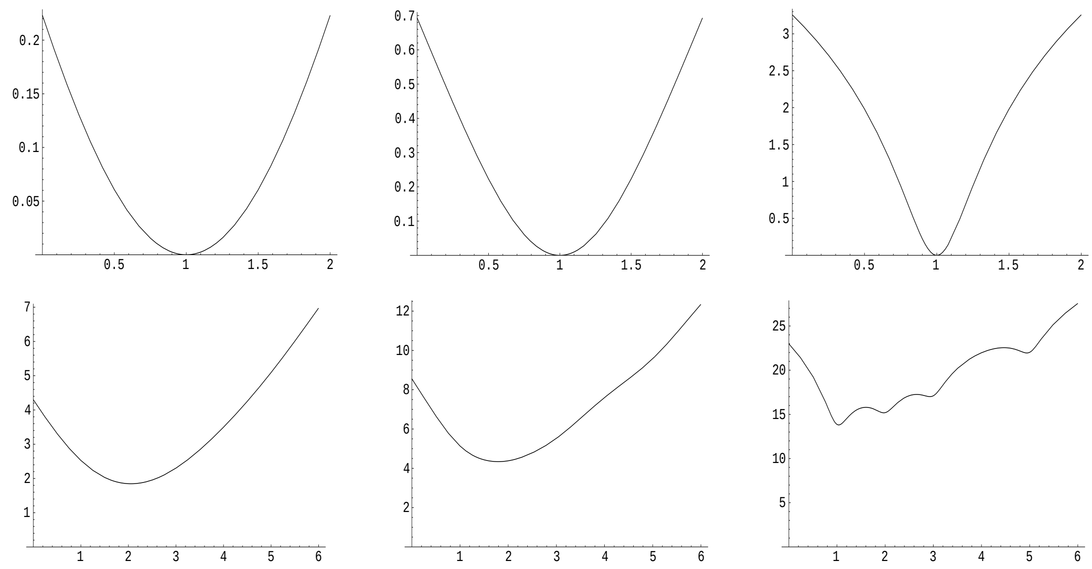

Figure 7.3: The coordinate is m. Top is Cauchy measures of $m-1$. Bottom is the same measures of the data set $(1, 1, 2, 3, 5)$. Left, center, and right are for $\bar{r} = (2, 1, .2)$. $\boxed{\text{rbst-cauchy}}$ [CR]

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 page 10 supplies two useful weighting functions that can be interchanged as arguments to the reweighted scheme on page 8.

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

7.2 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 chapter 6, Figure **??**.)

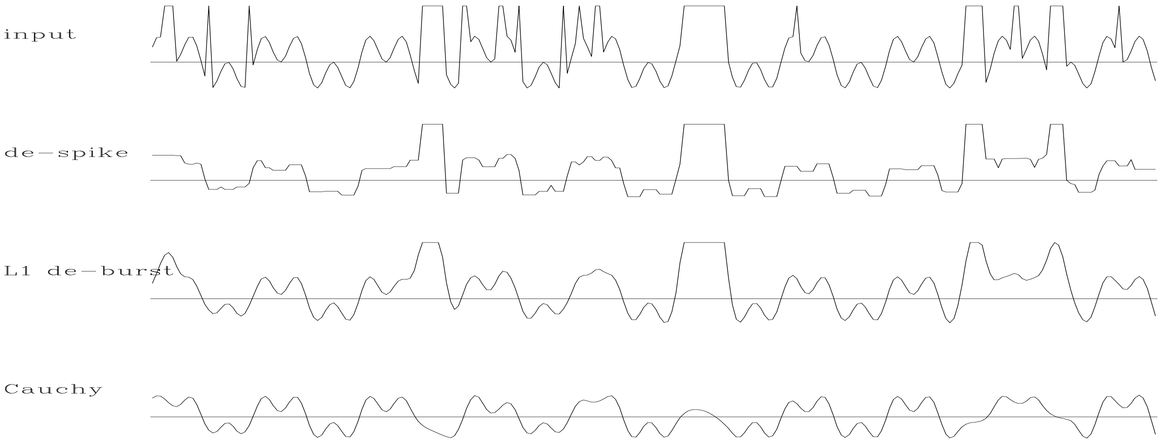

Figure 7.4: Top is synthetic data with noise spikes and bursts. (Most bursts are a hundred times larger than shown.) Next is after running medians. Bottom is after the two processes described here. rbst-burst90 [ER]

7.2.1 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.

7.2.2 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

$$\begin{array}{rcl} 0 & \approx & \mathbf{W}(\mathbf{h} - \mathbf{d}) \\ 0 & \approx & \epsilon \, \nabla^2 \mathbf{h} \end{array} \tag{7.18}$$

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 **??**, where the ∇^2 operator is replaced by an operator designed to predict this very predictable signal.

7.3 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 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.

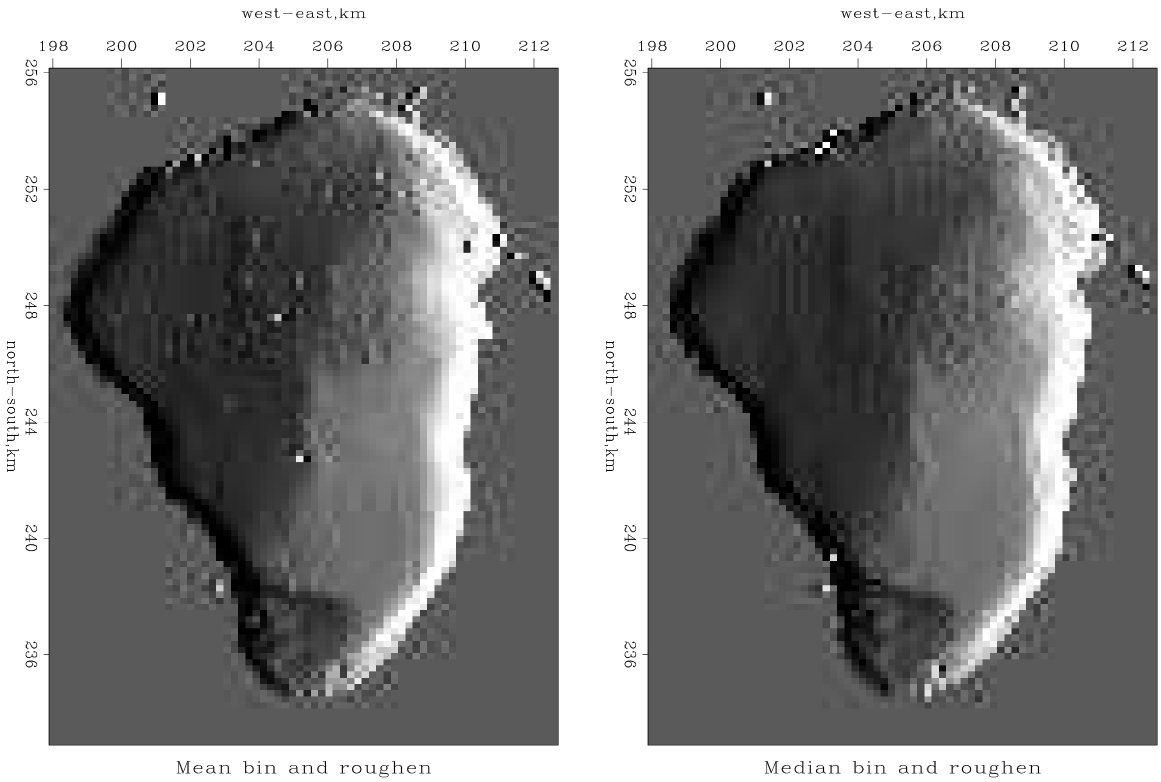

Figure 7.5: Galilee water depth binned and roughened. Left is binning with the mean, right with the median. rbst-medbin90 [ER]

Index

bursts, 11

Cauchy function, 9

deburst, 11
despike, 11

Hoare's algorithm, 3

irls module, 10

L-0 norm, 2
L-1 norm, 2
L-2 norm, 2
L1 or L_1, 6
L2 or L_2, 6

mean, 1
median, 1
median smoothing, 11
mode, 1, 2
module
    `irls`, weighting functions for iterative reweighting, 10
    `quantile`, percentile, 4
    `weightsolver`, iteratively reweighted optimization, 8

noise spikes, 4
nonlinear optimization, 7

percentile, 2

quantile module, 4

running median, 4

spikes, 11

weighted mean, 6
weightsolver module, 8