Chapter 1

Time-series analysis

The many applications of least squares to the one-dimensional convolution operator constitute the subject known as "**time-series analysis**." The **autoregression** filter, also known as the **prediction-error filter** (**PEF**), gathers statistics for us, not the autocorrelation or the spectrum directly but it gathers them indirectly as the inverse of the amplitude spectrum of its input. The PEF plays the role of the so-called "inverse-covariance matrix" in statistical estimation theory. Given the PEF, we use it to find missing portions of signals.

1.0.1 Time domain versus frequency domain

In the simplest applications, solutions can be most easily found in the frequency domain. When complications arise, it is better to use the time domain, directly applying the convolution operator and the method of least squares.

A first complicating factor in the frequency domain is a required boundary in the time domain, such as that between past and future, or requirements that a filter be nonzero in a stated time interval. Another factor that attracts us to the time domain rather than the Fourier domain is weighting functions.

Weighting functions are appropriate whenever a signal or image amplitude varies from place to place. Most of the literature on **time-series analysis** applies to the limited case of uniform weighting functions. Such time series are said to be "stationary." This means that their statistical properties do not change in time. In real life, particularly in the analysis of echos, signals are never stationary in time and space. A **stationarity** assumption is a reasonable starting assumption, but we should know how to go beyond it so we can take advantage of the many opportunities that do arise. In order of increasing difficulty in the frequency domain are the following complications:

1. A time boundary such as between past and future.

2. More time boundaries such as delimiting a filter.

3. More time boundaries such erratic locations of missing data.
4. Nonstationary signal, i.e., time-variable weighting.
5. Time-axis stretching such as normal moveout.

We will not have difficulty with any of these complications here because we will stay in the time domain and set up and solve optimization problems using the conjugate-gradient method. Thus we will be able to cope with great complexity in problem formulation and get the right answer without approximations. By contrast, analytic or partly analytic methods can be more economical, but they generally solve somewhat different problems than those given to us by nature.

1.1 SOURCE WAVEFORM, MULTIPLE REFLECTIONS

Here we devise a simple mathematical model for deep **water bottom** multiple reflections.[1] There are two unknown waveforms, the source waveform $S(\omega)$ and the ocean-floor reflection $F(\omega)$. The water-bottom primary reflection $P(\omega)$ is the convolution of the source waveform with the water-bottom response; so $P(\omega) = S(\omega)F(\omega)$. The first multiple reflection $M(\omega)$ sees the same source waveform, the ocean floor, a minus one for the free surface, and the ocean floor again. Thus the observations $P(\omega)$ and $M(\omega)$ as functions of the physical parameters are

$$
\begin{aligned}
P(\omega) &= S(\omega)\,F(\omega) && (1.1)\\
M(\omega) &= -S(\omega)\,F(\omega)^2 && (1.2)
\end{aligned}
$$

Algebraically the solutions of equations (1.1) and (1.2) are

$$
\begin{aligned}
F(\omega) &= -M(\omega)/P(\omega) && (1.3)\\
S(\omega) &= -P(\omega)^2/M(\omega) && (1.4)
\end{aligned}
$$

These solutions can be computed in the Fourier domain by simple division. The difficulty is that the divisors in equations (1.3) and (1.4) can be zero, or small. This difficulty can be attacked by using a positive number ϵ to **stabilize** it. For example, multiply equation (1.3) on top and bottom by $P'(\omega)$ and add $\epsilon > 0$ to the denominator getting

$$
F(\omega) \quad = \quad -\,\frac{M(\omega)P'(\omega)}{P(\omega)P'(\omega) + \epsilon} \tag{1.5}
$$

where $P'(\omega)$ is the complex conjugate of $P(\omega)$. Although the ϵ stabilization seems nice, it apparently produces a nonphysical model. For ϵ large or small, the time-domain response could turn out to be of much greater duration than is physically

[1] For this short course I am omitting here many interesting examples of multiple reflections shown in my 1992 book, PVI.

reasonable. This should not happen with perfect data, but in real life, data always has a limited spectral band of good quality.

Functions that are rough in the frequency domain will be long in the time domain. This suggests making a short function in the time domain by local smoothing in the frequency domain. Let the notation $< \cdots >$ denote smoothing by local averaging. Thus we can specify filters whose time duration is not unreasonably long by revising equation (1.5) to

$$F(\omega) \quad = \quad - \frac{< M(\omega) P'(\omega) >}{< P(\omega) P'(\omega) + \epsilon >} \tag{1.6}$$

where it remains to specify the method and amount of smoothing. The goal of finding the filters $F(\omega)$ and $S(\omega)$ is to best model the multiple reflections so that they can be subtracted from the data, enabling us to see what primary reflections have been hidden by the multiples.

These frequency-duration difficulties do not arise in a time-domain formulation. Unlike the frequency domain, in the time domain it is easy and natural to limit the duration and location of the nonzero time range of $F(\omega)$ and $S(\omega)$. First express (1.3) and (1.4) as

$$P(\omega) F(\omega) \quad \approx \quad -M(\omega) \tag{1.7}$$

$$M(\omega) S(\omega) \quad \approx \quad -P(\omega)^2 \tag{1.8}$$

To imagine the regression (1.7) in the time domain, think of $\mathbf{f}$ as a column vector containing the unknown sea-floor filter, $\mathbf{m}$ containing the "multiple" portion of a seismogram, and $\mathbf{P}$ as a matrix of down-shifted columns, each column being the "primary", say

$$\mathbf{m} \quad = \quad \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \\ m_7 \\ m_8 \end{bmatrix} \quad = \quad \begin{bmatrix} p_1 & 0 & 0 \\ p_2 & p_1 & 0 \\ p_3 & p_2 & p_1 \\ p_4 & p_3 & p_2 \\ p_5 & p_4 & p_3 \\ p_6 & p_5 & p_4 \\ 0 & p_6 & p_5 \\ 0 & 0 & p_6 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \\ f_3 \end{bmatrix} \quad = \quad \mathbf{Pf} \tag{1.9}$$

To code the solution of (1.9) we could use the conjugate-gradient subroutine `cgmeth()` (page ??), but we should remove its call to the matrix multiply subroutine and replace it by a convolution subroutine with boundary conditions of our choice.

1.2 AUTOREGRESSION

Given y_t and y_{t-1}, you might like to predict y_{t+1}. The prediction could be a scaled sum or difference of y_t and y_{t-1}. This is called "**autoregression**" because a signal is regressed on itself. To find the scale factors you would solve the regression below for the **prediction filter** (f_1, f_2)

$$\begin{bmatrix} y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \approx \begin{bmatrix} y_1 & y_0 \\ y_2 & y_1 \\ y_3 & y_2 \\ y_4 & y_3 \\ y_5 & y_4 \end{bmatrix} \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \tag{1.10}$$

(In practice, of course the system of equations would be much taller, and perhaps somewhat wider.) A typical row in the regression (1.10) says that $y_{t+1} = y_t f_1 + y_{t-1} f_2$ hence the description of f as a "prediction" filter. The error in the prediction is simply the left side minus the right side. Change the sign of all the terms, and move those on the left to the right getting

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \approx \begin{bmatrix} y_2 & y_1 & y_0 \\ y_3 & y_2 & y_1 \\ y_4 & y_3 & y_2 \\ y_5 & y_4 & y_3 \\ y_6 & y_5 & y_4 \end{bmatrix} \begin{bmatrix} 1 \\ -f_1 \\ -f_2 \end{bmatrix} \tag{1.11}$$

which is a standard form for autoregressions and prediction errors.

Multiple reflections are predictable. It is the unpredictable part of a signal, the prediction residual, that contains the primary information. The output of the filter $(1, -f_1, -f_2) = (a_0, a_1, a_2)$ is the unpredictable part of the input. This filter is a simple example of a "prediction-error" (PE) filter. It is one member of a family of filters called "error filters."

The error-filter family are filters with one coefficient constrained to be unity and various other coefficients constrained to be zero. Otherwise, the filter coefficients are chosen to have minimum power output. Names for various error filters follow:

$(1, a_1, a_2, a_3, \cdots, a_n)$	**prediction-error (PE) filter**
$(1, 0, 0, a_3, a_4, \cdots, a_n)$	gapped PE filter with a gap of 2
$(a_{-m}, \cdots, a_{-2}, a_{-1}, 1, a_1, a_2, a_3, \cdots, a_n)$	**interpolation-error (IE) filter**
$(a_{-m}, \cdots, a_{-4}, a_{-3}, 0, 0, 1, 0, 0, a_3, a_4, \cdots, a_n)$	a gapped IE filter

To compute a simple prediction error filter $\mathbf{a} = (1, a_1, a_2)$ with the CG method it is better to rewrite (1.10) or (1.11) as

$$-\begin{bmatrix} y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \end{bmatrix} \approx \begin{bmatrix} y_2 & y_1 & y_0 \\ y_3 & y_2 & y_1 \\ y_4 & y_3 & y_2 \\ y_5 & y_4 & y_3 \\ y_6 & y_5 & y_4 \end{bmatrix} \begin{bmatrix} 0 & \cdot & \cdot \\ \cdot & 1 & \cdot \\ \cdot & \cdot & 1 \end{bmatrix} \begin{bmatrix} 1 \\ a_1 \\ a_2 \end{bmatrix} \tag{1.12}$$

where we have introduced a **free-mask matrix K** which "passes" the freely variable coefficients in the filter and "rejects" the constrained coefficients which in this simple case is merely the first coefficient $a_0 = 1$.

$$\mathbf{K} \quad = \quad \begin{bmatrix} 0 & . & . \\ . & 1 & . \\ . & . & 1 \end{bmatrix} \tag{1.13}$$

Now that you have seen a simple case, let us proceed in general. (Notice the similarity of the free-mask matrix **K** in this filter estimation problem with the free-mask matrix **J** in missing data equation (??).) The regression is:

$$\mathbf{0} \quad \approx \quad \mathbf{Ya} \tag{1.14}$$

$$\mathbf{0} \quad \approx \quad \mathbf{Y(I - K + K)a} \tag{1.15}$$

$$\mathbf{0} \quad \approx \quad \mathbf{YKa + Y(I - K)a} \tag{1.16}$$

$$\mathbf{0} \quad \approx \quad \mathbf{YKa + Ya_0} \tag{1.17}$$

$$\mathbf{0} \quad \approx \quad \mathbf{YKa + y} \tag{1.18}$$

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{YKa + r_0} \tag{1.19}$$

which means we initialize the residual with $\mathbf{r}_0 = \mathbf{y}$ and then iterate with

$$\Delta\mathbf{a} \quad \longleftarrow \quad \mathbf{K'Y'\,r} \tag{1.20}$$

$$\Delta\mathbf{r} \quad \longleftarrow \quad \mathbf{YK}\,\Delta\mathbf{a} \tag{1.21}$$

Bringing this all together gives us subroutine `gdecon()` (page 5).

```
# gapped deconvolution.  input=yy(n)  output=rr(n)               UNTESTED.
#
subroutine gdecon( na,  yy,n,  rr,  niter, gap)
integer i, iter,   na,     n,       niter, gap
real                    yy(n), rr(n)
temporary real          dr(n), sr(n), aa(na), da(na), sa(na)
do i= 1, na
        aa(i) = 0.
aa(1) = 1.                                               # set output lag
call         icaf1(   0,0,   1, yy,n,  aa,na, rr)        # filter data yy
do iter= 0, niter {
        call icaf1(   1,0,   1, yy,n,  da,na, rr)        # da=yy*rr
        do i= 1, gap
                da(i) = 0.                               # constrained lags
        call icaf1(   0,0,   1, yy,n,  da,na, dr)        # dr=yy*da
        call cgplus( iter,  na,aa,da,sa,
                            n ,rr,dr,sr )                # f=f+da; r=r+dr
        }
return; end
```

I am planning to make a demonstration of the gapped decon subroutine `gdecon()` (page 5).

1.2.1 Prediction-error filter output is white.

The most important property of a **prediction-error filter** or **PEF** is that its output tends to a white spectrum. No matter what the input to this filter, its output tends to whiteness as the number of the coefficients $n \to \infty$ tends to infinity. Thus, the **PE filter** adapts itself to the input by absorbing all its **color**. This has important mathematical implications and important geophysical implications.

Mathematically, PEF output being white means that a spectrum can be specified by giving it, giving the input's autocorrelation or giving the PEF. Each is transformable to the other two. Indeed, an effective mechanism of spectral estimation, developed by John P. **Burg** and described in **FGDP**, is to compute a PE filter and look at the inverse of its spectrum.

That the output spectrum of a PE filter is **white** is also useful geophysically. Imagine the reverberation of the **soil** layer, highly variable from place to place, as the resonance between the surface and deeper consolidated rocks varies rapidly with surface location as a result of geologically recent fluvial activity. The spectral **color** of this erratic variation on surface-recorded seismograms is compensated for by a PE filter. Of course, we do not want PE-filtered seismograms to be white, but once they all have the same spectrum, it is easy to postfilter them to any desired spectrum.

The power of a PE filter is that a short filter can often extinguish, and thereby represent, the information in a long resonant filter. If the input to the PE filter is a sinusoid, it is exactly predictable by a three-term recurrence relation, and all the color is absorbed by a three-term PE filter (see exercises). Burg's spectral estimation is especially effective for truncated data series.

1.2.2 Deconvolution

Prediction-error filtering is also called "**deconvolution**". This word goes back to very basic models and concepts. In this model one envisions a random white-spectum excitation function $\mathbf{x}$ existing in nature, and this excitation function is somehow filtered by unknown natural processes, with a filter operator $\mathbf{B}$ producing an *output* $\mathbf{y}$ in nature that becomes the *input* $\mathbf{y}$ to our computer programs. This is sketched in Figure 1.1. Then we design a prediction-error filter $\mathbf{A}$ on $\mathbf{y}$ getting a white-spectrumed output residual $\mathbf{r}$. Since $\mathbf{r}$ and $\mathbf{x}$ theoretically have the same spectrum, the tantalizing prospect is that maybe $\mathbf{r}$ equals $\mathbf{x}$, meaning that the PEF $\mathbf{A}$ has *deconvolved* the unknown convolution $\mathbf{B}$. Theoretically, this prospect can be valid in the mathematical limit of large data sets if the phase of the filter operator $\mathbf{B}$ is the minimum consistent with causality. Theoretically, it happens that reflections in layered media fulfil this requirement (so we have ample reasons to expect validity in practice). This old, fascinating subject is examined in my first book, FGDP.

1.2.3 Burg's proof that PE filter output is white

The basic idea of least-squares fitting is that the residual is orthogonal to the fitting functions. Applied to the PE filter, this idea means that the output of a PE filter is orthogonal to lagged inputs. The **orthogonality** applies only for lags in the past because prediction knows only the past while it aims to the future. What we want

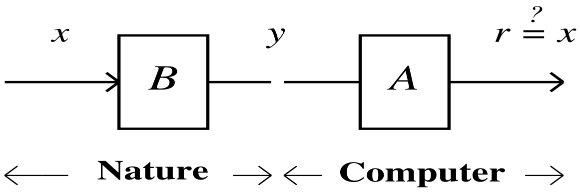

Figure 1.1: Flow of information from nature, to observation, into computer. tsa-systems [NR]

to show is different, namely, that the output is uncorrelated with *itself* (as opposed to the input) for lags in *both* directions; hence the output spectrum is **white**.

We are given a signal y_t and filter it by

$$x_t \quad = \quad y_t + \sum_{\tau > 0} a_\tau y_{t-\tau} \tag{1.22}$$

We found each a_τ by minimizing the energy $\sum_t x_t^2$ by setting to zero its derivative by a_τ:

$$\frac{1}{2} \frac{\partial}{\partial a_\tau} \sum_t x_t^2 \quad = \quad \sum_t x_t \frac{\partial x_t}{\partial a_\tau} \quad = \quad \sum_t x_t \, y_{t-\tau} \quad = \quad 0 \qquad \text{for } \tau > 0 \tag{1.23}$$

We interpret this to mean that the residual x_t is orthogonal to each of the fitting functions $y_{t-\tau}$. (One fitting function is y_{t-7}.) In other words, the present PE filter output is orthogonal to its past inputs. In still other words, one side of the crosscorrelation vanishes. Taking an unlimited number of time lags and filter coefficients, the crosscorrelation vanishes not only for $\tau > 0$ but for larger values, say $\tau + s$ where $\tau \geq 0$ and $s > 0$. In other words, the future PE filter outputs are orthogonal to present and past inputs:

$$\sum_t x_{t+s} y_{t-\tau} \quad = \quad 0 \qquad \text{for } \tau \geq 0 \text{ and } s > 0 \tag{1.24}$$

Recall that if $\mathbf{r} \cdot \mathbf{u} = 0$ and $\mathbf{r} \cdot \mathbf{v} = 0$, then $\mathbf{r} \cdot (a_1 \mathbf{u} \pm a_2 \mathbf{v}) = 0$ for any a_1 and a_2. So for any a_τ we have

$$\sum_t x_{t+s}(y_t \pm a_\tau y_{t-\tau}) \quad = \quad 0 \qquad \text{for } \tau \geq 0 \text{ and } s > 0 \tag{1.25}$$

and since this applies to any linear combination we have

$$\sum_t x_{t+s}(y_t + \sum_{\tau > 0} a_\tau y_{t-\tau}) \quad = \quad 0 \qquad \text{for } s > 0 \tag{1.26}$$

Therefore, substituting from (1.22), we get

$$\sum_t x_{t+s} x_t \quad = \quad 0 \qquad \text{for } s > 0 \tag{1.27}$$

which is an **autocorrelation** function and must be symmetric. Thus,

$$\sum_t x_{t+s} x_t \quad = \quad 0 \qquad \text{for } s \neq 0 \tag{1.28}$$

Since the autocorrelation of the prediction-error output is an impulse, its spectrum is **white**.

1.3 TIME VARIABLE FILTERING

Filters sometimes change with time. We sometimes observe signals whose spectrum changes with time. Simple modifications to earlier convolution subroutines such as `icaf1()` (page ??) lead to subroutine `vfilt1()` (page 8) which allows for a different filter on each output data point.

```
# Nonstationary filtering. 1-dimensional                        UNTESTED.
#
subroutine vfilt1( adj, add, yy,n,  aa,  na,  rr  )
integer            adj, add,  y,n,   a,  na,   r
real                         yy(n), aa(n,na), rr(n)
call adjnull(      adj, add,        aa,n*na,  rr,n )
do a = 1, na {
do r = na, n {
                                                        y = r - a + 1
        if( adj == 0 )
                        rr(r)   = rr(r)   + aa(r,a) * yy(y)
        else
                        aa(r,a) = aa(r,a) + rr(y)   * yy(y)
        } }
return; end
```

The filter coefficients take much computer memory because there is a separate filter for every data point. The trouble with all these filter coefficients is that someone must find them. If we have no theory to find them, we must find them experimentally, say by a deconvolution filter estimation process.

To do this we need an initial filter $\mathbf{a}_0 = (1,0,0,0,\cdots)$ and a filter constraint matrix $\mathbf{K}$ like equation (1.13). Unlike in equation (1.19) we need to visualize $\mathbf{K}$ at each output time point and we need to initialize $\mathbf{a}$ with copies of $\mathbf{a}_0$ at every time point. Formally, the regression (1.19) for $\mathbf{a}$ remains

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{YKa} + \mathbf{r}_0 \tag{1.29}$$

but we must think of $\mathbf{a}$ as an abstract vector into which we have packed a matrix (a filter for each output time). The difficulty with this regression is that there are many more filter unknowns than there are regression equations by about a factor equal the number of filter coefficients. We can hope to get reasonable results only if we somehow use an operator, say $\mathbf{S}$, that smooths over long windows on the time axis. One possibility is

$$\mathbf{0} \quad \approx \quad \mathbf{YSKa} \tag{1.30}$$

which I think would work well. In principle, we are still immensely underdetermined but I think the first iterations of CG would lead us where we want to go.

An alternate approach is the joint regression

$$\mathbf{r} \quad \approx \quad \mathbf{YKa} \tag{1.31}$$

$$\mathbf{0} \quad \approx \quad \epsilon\mathbf{Ra} \tag{1.32}$$

where $\mathbf{R}$ is a roughening operator, perhaps $\mathbf{I}-\mathbf{S}$. So we have two prospective methods and we need a synthetic data study to see which method works best.

I hope also to find a test case with field data, but experience in seismology is that spectral changes are slow, implying unexciting results. Many interesting examples should exist in two and three-dimensional filtering, however, because reflector dip is always changing and that changes the *spatial* spectrum.

EXERCISES:

1 Sketch the matrix corresponding to subroutine `vfilt1()` (page 8). HINTS: Do not try to write all the matrix elements. Instead draw short lines to indicate rows or columns. As a "warm up" consider a simpler case where one filter is used on the first half of the data and another filter for the other half. Then upgrade that solution from two to about ten filters.

1.4 WEIGHTED ERROR FILTERS

There are *practical* reasons for wanting **weighting functions** in regressions and there are *physical* reasons. When data points are missing for accidental reasons, they spoil the regression equations they affect and it is best to simply weight those equations by zero. We will be looking at many examples of this type in two-dimensions in chapter ??.

Physically, echos get weaker with time, though the information content is unrelated to the signal strength. Echos also vary in strength as different materials are encountered by the outgoing wave. Programs for echo analysis typically divide the data by a scaling factor that is a smoothed average of the signal strength. This practice is nearly universal, although it is fraught with hazards. An example of **automatic gain control** (**AGC**) is to compute the divisor by forming the absolute value of the signal strength and then smoothing by convolution with a triangle or damped exponential. Pitfalls are strange amplitude behavior surrounding the water bottom, and the overall loss of information contained in amplitudes. Personally, I have found that the gain function t^2 nearly always eliminates the need for AGC on raw field data, but I have no doubt that AGC is occasionally needed. (A theoretical explanation for t^2 is given in **IEI**.)

Equation (1.33) shows a tiny example of a weighted least-squares filtering regression. In real-life applications, the output is typically 1000-2000 points and the filter 5-50 points.

$$\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \approx \quad \begin{bmatrix} w_1 & 0 & 0 & 0 \\ 0 & w_2 & 0 & 0 \\ 0 & 0 & w_3 & 0 \\ 0 & 0 & 0 & w_4 \end{bmatrix} \left(\begin{bmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \end{bmatrix} - \begin{bmatrix} y_1 & 0 \\ y_2 & y_1 \\ y_3 & y_2 \\ 0 & y_3 \end{bmatrix} \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \right) \tag{1.33}$$

The weights w_t are any positive numbers we choose. In "accidental" applications, we set weights to zero where data points are missing or bad. In physical applications, we choose the w_t so that the residual components are about equal in magnitude. (A generalization of weighting is where the weighting matrix is a filtering matrix or a product of weighting and filtering.)

Equation (1.33) is of the form $\mathbf{0} \approx \mathbf{W}(\mathbf{d} - \mathbf{Y}\mathbf{a})$. This can be converted to a new problem without weights by defining a new data vector $\mathbf{Wd}$ and a new operator $\mathbf{WY}$ simply by carrying $\mathbf{W}$ through the parentheses to $\mathbf{0} \approx \mathbf{Wd} - (\mathbf{WY})\mathbf{a}$. A diagonal matrix multiply operation is implemented in subroutine `diag()` (page 10)

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

Convolution followed by weighting is implemented in subroutine `wicaf1()` (page 10).

```
# filter and weight.
#
subroutine wicaf1( adj, add, ww,    lag, xx,nn,   ff,nf,  yy    )
integer            adj, add,        lag,    nn,      nf
real                         ww(nn),     xx(nn),  ff(nf), yy(nn)
temporary real               ss(nn)
call adjnull(      adj, add,                      ff,nf,  yy,nn)

if( adj == 0) {  call icaf1(    0,  0,  lag,  xx,nn, ff,nf,     ss)
                 call diag(     0, add, ww,      nn,            ss, yy)
                 }
else          {  call diag(     1,  0,  ww,      nn,            ss, yy)
                 call icaf1(    1,  0,  lag,  xx,nn, ff,nf,     ss)
                 }
return;  end
```

These steps are assembled in subroutine `pef1()` (page 10) along with weighting functions using subroutine `wicaf1()` (page 10).

```
# weighted prediction-error filter
#
subroutine pef1( aa,na,  yy,nr,rr, ww, niter)
integer i, iter,    na,     nr,        niter
real             aa(na), yy(nr), rr(nr), ww(nr)
temporary real   da(na), sa(na), dr(nr), sr(nr)
do i=1,na { aa(i) = 0.};        aa(1)=1.;
call         wicaf1( 0,0, ww,  1, yy,nr,  aa,na, rr)      # r = W Y a
do iter= 0, niter {
        call wicaf1( 1,0, ww,  1, yy,nr,  da,na, rr)      # da = Y' r
        da(1) = 0.                                        # constrained point
        call wicaf1( 0,0, ww,  1, yy,nr,  da,na, dr)      # dr = Y da
        call cgplus( iter,  na,aa,da,sa,
                            nr,rr,dr,sr )                 # aa=aa+da
        }
return; end
```

EXERCISES:

1 An exercise in chapter ?? shows how to remove the mean value of a signal. This chapter shows how to do deconvolution with a prediction-error filter. Now show how to do both of these operations simultaneously by specifying the regression matrix and its adjoint. Describe an example where simultaneous should be better.

2 Write the subroutine needed to implement the previous exercise.

1.5 TWO-STAGE LINEAR LEAST SQUARES

To solve a missing data regression, we need a filter. That filter should be a prediction-error filter (PEF). Thus, finding missing data is a two-stage process using linear least squares (LS) at each stage.

Likewise, fitting data to a model, we need a weighting function. The first stage is to fit a model with any weighting function. Then we examine the residual and define a weighting function inverse to it in the sense of amplitude in physical space and in the sense of amplitude in Fourier space. Nothing says we need to stop after two stages.

Let us examine some details in the small problem of data restoration in one dimension. Recall the theory and discussion leading up to Figure ??. The best method we came up with there was to use the regression $0 \approx w_i(y_{i+1} - 2y_i + y_{i-1})$ 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 replace the second derivative (Laplacian in 1-D) by the "best" operator which we will learn by studying the data itself. The result of this "PEF-deburst" processing is shown in Figure 1.2.

Subroutine `pefest1()` (page 11) finds the operator.

```
# Estimate a PEF avoiding zeros and bursty noise on input.
#
subroutine pefest1( aa,na,  yy,nr,  rr, niter)
integer i,              na,     nr,      niter
real                aa(na), yy(nr), rr(nr),   rbar, easilygood
temporary real      da(na), sa(na), ww(nr), dr(nr), sr(nr), afre(na), yfre(nr)
call null(aa,na);  aa(1)=1; aa(2)=-2; aa(3)=1   # starting guess.
call  icai1( 0,0,1, aa  , 3, yy,  nr,rr)        # rr=misfits
easilygood=.33;   i= easilygood*nr
call quantabs(    i, nr, rr, rbar)              # rbar={ r safe below rbar}
do i = 1, nr
        if( yy(i)==0.| abs(rr(i))>5*rbar | i<3)  { yfre(i) = 1 }  # risky
        else                                     { yfre(i) = 0 }  # safe enuf
do i = 1,na
        afre(i) = 1.
call  icai1( 0,0,1, afre,na, yfre,nr,rr)        # likely bad outputs.
do i = 1, nr
        if( rr(i) > 0) { ww(i) = 0 }            # unsafe regressors
        else           { ww(i) = 1 }            # safe   regressors
call pef1( aa,na,  yy,nr,rr, ww, niter)
return; end
```

Given the filter that comes out of `pefest1()` (page 11), subroutine `fixbad1()`

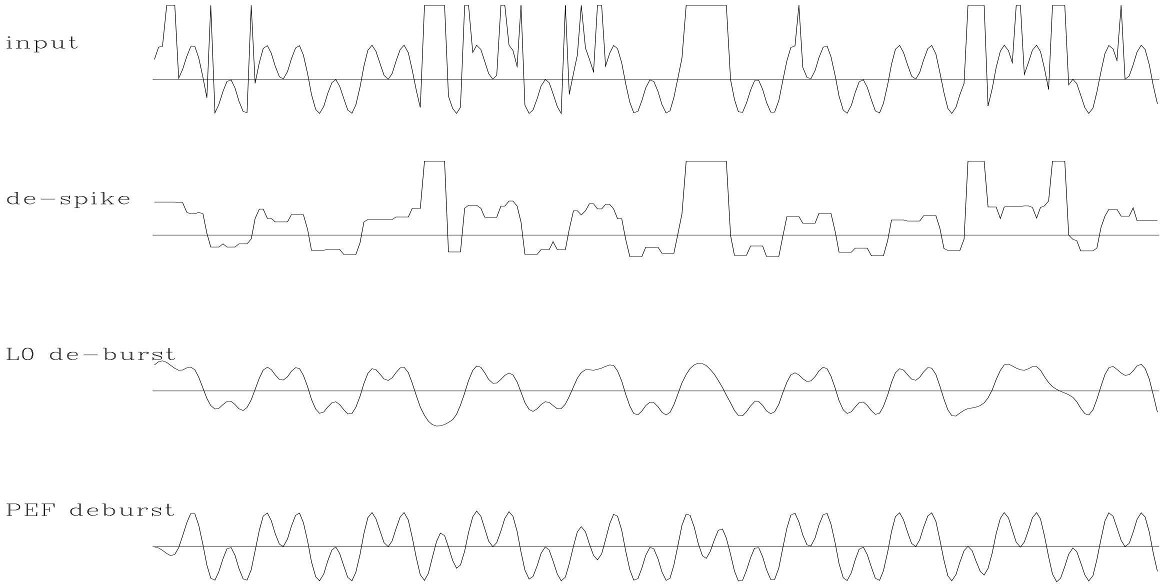

Figure 1.2: Top is synthetic data with noise spikes and bursts. (Some bursts are fifty times larger than shown.) Next is after running medians. Next is Laplacian L_0 weighted deburst processing. Last is PEF-deburst processing. tsa-pefdeburst [NR]

(page 12) 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() (page ??) is invoked to replace the zeroed values by reasonable ones.

```
# Given a PEF, finds bad data and restores it.
#
subroutine fixbad1(  niter, aa,na, yy,n)
integer i,           niter,    na,    n
real                        aa(na),yy(n), rbar
temporary real                            rr(n), yfre(n)
call    icai1( 0,0,1,       aa,na, yy, n, rr)
call quantabs( n/2,                    n, rr, rbar)
do i=1,n
        if( yy(i)==0.| abs(rr(i))>4*rbar | i<na)  { yfre(i) = 1.}
        else                                      { yfre(i) = 0.}
do i=1,n
        yy(i) = yy(i) * (1. - yfre(i))
call mis1( yfre, yy, n, niter, aa,na)
return; end
```

1.6 MISSING DATA AND UNKNOWN FILTER

Recall the missing-data figures beginning with Figure ??. There the filters were taken as known, and the only unknowns were the missing data. Now, instead of having a predetermined filter, we will solve for the filter along with the missing data. The principle we will use is that the output power is minimized while the filter is constrained to have one nonzero coefficient (else all the coefficients would go to zero). We will look first at some results and then see how they were found.

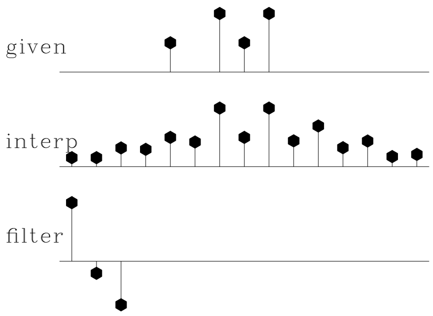

Figure 1.3: Top is known data. Middle includes the interpolated values. Bottom is the filter with the leftmost point constrained to be unity and other points chosen to minimize output power. tsa-misif [NR]

In Figure 1.3 the filter is constrained to be of the form $(1, a_1, a_2)$. The result is pleasing in that the interpolated traces have the same general character as the given values. The filter came out slightly different from the $(1, 0, -1)$ that I suggested for Figure ?? based on a subjective analysis. Curiously, constraining the filter to be of the form $(a_{-2}, a_{-1}, 1)$ in Figure 1.4 yields the same interpolated missing data as in Figure 1.3. I understand that the sum squared of the coefficients of $A(Z)P(Z)$ is the same as that of $A(1/Z)P(Z)$, but I do not see why that would imply the same interpolated data.

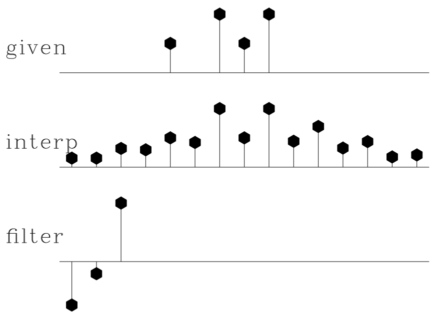

Figure 1.4: The filter here had its rightmost point constrained to be unity—i.e., this filtering amounts to backward prediction. The interpolated data seems to be identical, as with forward prediction. tsa-backwards [NR]

1.6.1 Objections to interpolation error

In any data interpolation or extrapolation, we want the extended data to behave like the original data. And, in regions where there is no observed data, the extrapolated data should drop away in a fashion consistent with its **spectrum** determined from the known region.

My basic idea is that the spectrum of the missing data should match that of the known data and this happens with the PEF (one sided filter) because its spectrum tends to the inverse of that of the known data while that of the unknown data tends to the inverse of that of the PEF. Thus the spectrum of the missing is the "inverse of the inverse" of the spectrum of the known. The idea that the missing data and the known data should have the same spectrum is the idea that the spectrum should be unchanging from one region to another. A technical word to express the idea of spectra not changing is "stationary."

On the other hand, the **interpolation-error filter**, a filter like $(a_{-2}, a_{-1}, 1, a_1, a_2)$, should fail to do the job because it has the wrong spectrum. Its spectrum tends to the inverse of the spectrum in (I am stating this fact without proof), so the spectrum of the IE filter tends to the inverse *squared* of the spectrum in.

To confirm and show these concepts, I prepared synthetic data consisting of a fragment of a damped exponential, and off to one side of it an impulse function. Most of the energy is in the damped exponential. Figure 1.5 shows that the spectrum and the extended data are about what we would expect. From the extrapolated data, it is impossible to see where the given data ends.

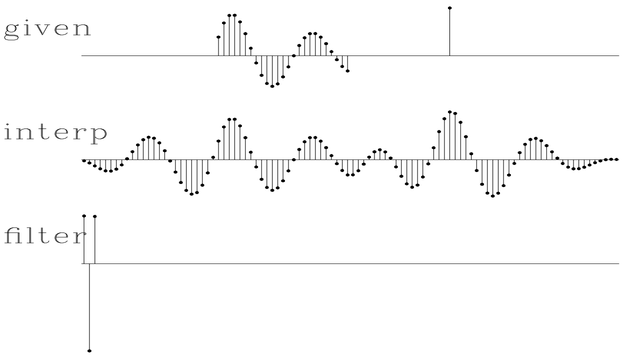

Figure 1.5: Top is synthetic data with missing portions. Middle includes the interpolated values. Bottom is the filter, a *prediction*-error filter which may look symmetric but is not quite. tsa-exp [NR]

For comparison, I prepared Figure 1.6. It is the same as Figure 1.5, except that the filter is constrained in the middle. Notice that the extended data does *not* have the spectrum of the given data—the wavelength is much shorter. The boundary between real data and extended data is not nearly as well hidden as in Figure 1.5.

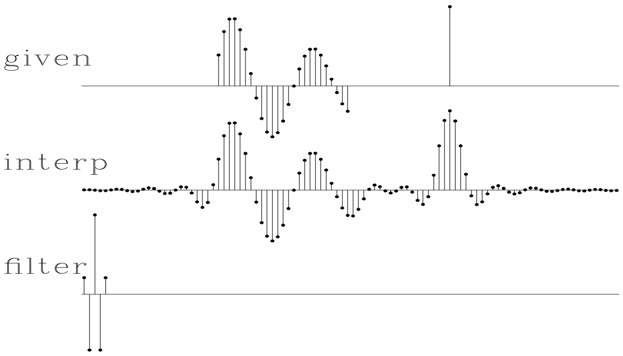

Figure 1.6: Top is the same synthetic data. Middle includes the interpolated values. Bottom is the filter, an *interpolation*-error filter. tsa-center [NR]

1.6.2 Packing both missing data and filter into a CG vector

Now let us examine the theory and coding behind the above examples. Define a roughening filter $A(\omega)$ and a data signal $Y(\omega)$ at some stage of interpolation. The regression is $0 \approx A(\omega)Y(\omega)$ where the filter $A(\omega)$ has at least one time-domain coefficient constrained to be nonzero and the data contains both known and missing values. Think of perturbations ΔA and ΔY. We neglect the nonlinear term $\Delta A\, \Delta Y$ as follows:

$$0 \;\approx\; (A \,+\, \Delta A)(Y \,+\, \Delta Y) \tag{1.34}$$

$$0 \;\approx\; A\,\Delta Y \,+\, Y\,\Delta A \,+\, AY \,+\, \Delta A\,\Delta Y \tag{1.35}$$

$$0 \;\approx\; A\,\Delta Y \,+\, Y\,\Delta A \,+\, AY \tag{1.36}$$

Let us give these ideas concrete form by writing the regression (1.36) using matrix algebraic notation. For this we need mask matrices (diagonal matrices with ones on the diagonal where variables are free and zeros where they are constrained i.e., where $\Delta a_i = 0$ and $\Delta y_i = 0$). The free-mask matrix for missing data is denoted $\mathbf{J}$ and that for the PE filter is $\mathbf{K}$. The regression (1.36) becomes

$$\mathbf{0} \quad \approx \quad \mathbf{AJ}\Delta\mathbf{y} + \mathbf{YK}\Delta\mathbf{a} + (\mathbf{Ay} \text{ or } \mathbf{Ya}) \tag{1.37}$$

or

$$\mathbf{0} \quad \approx \quad \begin{bmatrix} \mathbf{AJ} & \mathbf{YK} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{y} \\ \Delta\mathbf{a} \end{bmatrix} + \mathbf{Ay} \tag{1.38}$$

For a 3-term filter and a 7-point data signal the regression (1.37) becomes

$$\left[\begin{array}{cccccccccc}
a_0 & . & . & . & . & . & . & y_0 & . & . \\
a_1 & a_0 & . & . & . & . & . & y_1 & y_0 & . \\
a_2 & a_1 & a_0 & . & . & . & . & y_2 & y_1 & y_0 \\
. & a_2 & a_1 & a_0 & . & . & . & y_3 & y_2 & y_1 \\
. & . & a_2 & a_1 & a_0 & . & . & y_4 & y_3 & y_2 \\
. & . & . & a_2 & a_1 & a_0 & . & y_5 & y_4 & y_3 \\
. & . & . & . & a_2 & a_1 & a_0 & y_6 & y_5 & y_4 \\
. & . & . & . & . & a_2 & a_1 & . & y_6 & y_5 \\
. & . & . & . & . & . & a_2 & . & . & y_6 \\
. & . & . & . & . & . & . & . & . & .
\end{array}\right]
\begin{bmatrix} \mathbf{J} & \mathbf{0} \\ \mathbf{0} & \mathbf{K} \end{bmatrix}
\left[\begin{array}{c}
\Delta y_0 \\ \Delta y_1 \\ \Delta y_2 \\ \Delta y_3 \\ \Delta y_4 \\ \Delta y_5 \\ \Delta y_6 \\ \hline \Delta a_0 \\ \Delta a_1 \\ \Delta a_2
\end{array}\right]
+
\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ r_3 \\ r_4 \\ r_5 \\ r_6 \\ r_7 \\ r_8 \\ 0 \end{bmatrix}
\approx \mathbf{0} \tag{1.39}$$

where $\mathbf{r} = \mathbf{Ay}$, in other words, r_t is the convolution of a_t with y_t, namely, $r_0 = y_0 a_0$ and $r_1 = y_0 a_1 + y_1 a_0$, etc. To solve this regression we first initialize $\mathbf{a} = (1, 0, 0, \cdots)$ and the put zeros in for missing data in $\mathbf{y}$. Then we iterate over equations (1.40) to (1.44). This is the same idea as all the linear regressions we have been solving except that now we recompute the residual $\mathbf{r}$ inside the iteration loop so that as convergence is achieved (*if* it is achieved) the neglected nonlinear term $\Delta A \Delta Y$ tends to zero.

$$\mathbf{r} \quad \longleftarrow \quad \mathbf{Ay} \tag{1.40}$$

$$\begin{bmatrix} \Delta\mathbf{y} \\ \Delta\mathbf{a} \end{bmatrix} \quad \longleftarrow \quad \begin{bmatrix} \mathbf{J}'\mathbf{A}' \\ \mathbf{K}'\mathbf{Y}' \end{bmatrix} \mathbf{r} \tag{1.41}$$

$$\Delta\mathbf{r} \quad \longleftarrow \quad \begin{bmatrix} \mathbf{AJ} & \mathbf{YK} \end{bmatrix} \begin{bmatrix} \Delta\mathbf{y} \\ \Delta\mathbf{a} \end{bmatrix} \tag{1.42}$$

$$\mathbf{y} \longleftarrow \text{cgplus}(\mathbf{y}, \Delta\mathbf{y}) \tag{1.43}$$

$$\mathbf{a} \longleftarrow \text{cgplus}(\mathbf{a}, \Delta\mathbf{a}) \tag{1.44}$$

A solver subroutine needs to **pack** both unknowns into a single vector `x() =` $(\Delta Y, \Delta A)$ before calling the conjugate-gradient program. Likewise, the resulting filter and data coming out must be unpacked. Also, the **gradient** now has two contributions, one from $A\,\Delta Y$ and one from $Y\,\Delta A$, and these must be combined. The subroutine `misif()` (page 16) which makes Figures 1.3 through 1.6, effectively combines `pef1()` (page 10) and `mis1()` (page ??). A new aspect is that, to avoid accumulation of errors from the neglect of the **nonlinear** product $\Delta A\,\Delta Y$, the residual is recalculated inside the iteration loop instead of only once at the beginning. We gather the above ideas into subroutine `misif()` (page 16).

```
# MISSIF -- find MISSing Input data and Filter on 1-axis by min power out.
#
subroutine misif( na, lag, aa, ny, yy, yfree, niter)
integer iter,     na, lag,     ny,           niter, nx, ax, yx, iy, nr
real    yy(ny)            # input: known data with zeros for missing values.
                          # output: known plus missing data.
real    yfree(ny)         # yfree(iy) =0 where y(iy) is a known value, =1 otherwise
real    aa(na)            # input and output: roughening filter
temporary real  xx(ny+na),   rr(ny+na)
temporary real  dx(ny+na),   dr(ny+na)
temporary real  sx(ny+na),   sr(ny+na)
nr= ny+na;      nx= ny+na;      yx=1;   ax=1+ny;
call copy( ny,  yy, xx(yx));      if( aa(lag)==0.)   call erexit('misif: a(lag)==0.')
call copy( na,  aa, xx(ax))
do iter= 0, niter {
        call tcai1( 0, 0, aa,na, yy    ,ny, rr)         # rebuild residual
        call tcai1( 1, 0, aa,na, dx(yx),ny, rr)
        call tcaf1( 1, 0, yy,ny, dx(ax),na, rr)
        do iy= 1, ny
                dx( iy) = dx(iy) * yfree(iy)
        dx( lag+ny) = 0.                                # aa(lag)=1 and da(lag)=0
        call tcai1( 0, 0, aa,na, dx(yx),ny, dr)
        call tcaf1( 0, 1, yy,ny, dx(ax),na, dr)
        call cgplus( iter, nx, xx, dx, sx,
                           nr, rr, dr, sr )
        call copy( ny, xx(yx), yy)
        call copy( na, xx(ax), aa)
        }
return; end
```

There is a danger that `misif()` might converge very slowly or fail if `aa()` and `yy()` are much out of scale with each other. Till now, I have found it best to **prescale** the data to a maximum value of unity (as the filter is constrained to have its first value being $+1$).

EXERCISES:

1 Two filters, $\mathbf{a}$ and $\mathbf{b}$, are convolved on a time series $\mathbf{x}$ to get an output $\mathbf{y} = \mathbf{a} * \mathbf{b} * \mathbf{x}$ or $\mathbf{y} = \mathbf{b} * \mathbf{a} * \mathbf{x}$. Linearize to express $\Delta\mathbf{y}$ as a function of $\Delta\mathbf{a}$ and $\Delta\mathbf{b}$. Describe the adjoint relation getting $\Delta\mathbf{a}$ and $\Delta\mathbf{b}$ as a function of $\Delta\mathbf{y}$.

1.7 LEVELED INVERSE INTERPOLATION

Eighteenth and nineteenth century mathematics literature gives us many methods of interpolating functions. These classical methods are generally based on polynomials. The user specifies some order of polynomial and the theory works out the coefficients. Today our interest is in both interpolating and extrapolating wavefields (which are solutions to low order differential equations) and we use methods that are much better behaved than polynomials when extrapolating data, methods which behave acceptably when faced with contradictory data values, and methods which also apply in two and three dimensions.

In chapter ?? subroutine `invint1()` (page ??) solved the problem of inverse linear interpolation, namely, given scattered data points, find a function on a uniform mesh from which linear interpolation gives the scattered data points. To cope with regions where there are no data points, the subroutine requires an input roughening filter. This is a bit like specifying a differential equation to be satisfied between the data points. The question is, how should we choose a roughening filter? The importance of the roughening filter grows as the data gets sparser or as the mesh is refined.

Figures 1.3-1.6 suggest that the choice of the roughening filter need not be subjective, nor a priori, but that the prediction-error filter (PEF) is the ideal roughening filter. Spectrally, the PEF tends to the inverse of its input hence its output tends to be "level". Missing data that is interpolated with this "leveler" tends to have the spectrum of given data.

1.7.1 Leveled inverse interpolation test results

Figures 1.7 and 1.8 show the same example as in Figures ?? and ??. What is new here is that the proper PEF is not given but is determined from the data. Figure 1.7 was made with a three coefficient filter $(1, a_1, a_2)$ and Figure 1.8 was made with a five coefficient filter $(1, a_1, a_2, a_3, a_4)$. The main difference in the figures is where the data is sparse. The three coefficient filter can fit only a single sinusoid and it does that. The five coefficient filter can fit only two sinusoids and it does something like that, a high frequency sinsoid on the left and a lower one on the right. The data points in Figures ??, 1.7 and 1.8 are samples from a sinusoid.

> Comparing Figures ?? and ?? to Figures 1.7 and 1.8 we conclude that by finding and imposing the prediction-error filter while finding the model space, we have interpolated beyond aliasing in data space.

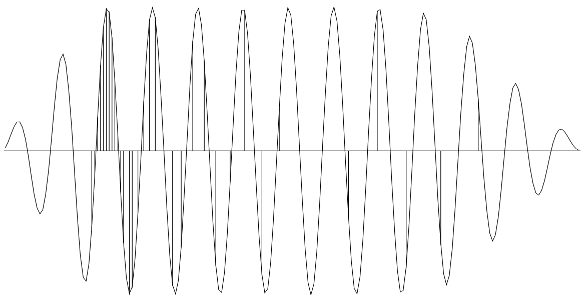

Figure 1.7: Interpolating with a three term filter. The interpolated signal is fairly monofrequency. tsa-subsine3 [NR]

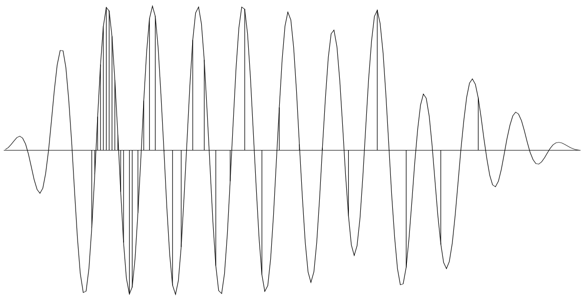

Figure 1.8: Interpolating with a five term filter. A low frequency component grows towards the right. tsa-subsine5 [NR]

1.7.2 Leveled inverse interpolation analysis

Here we see how the interpolation beyond aliasing was done. The first "statement of wishes" is that the observational data $\mathbf{d}$ should result from linear interpolation $\mathbf{L}$ of the uniformly sampled model space $\mathbf{m}$, namely, $\mathbf{0} \approx \mathbf{Lm} - \mathbf{d}$. Expressing this as a change $\Delta\mathbf{m}$ gives the regression in terms of the model change, $\mathbf{0} \approx \mathbf{L}\Delta\mathbf{m} + (\mathbf{Lm} - \mathbf{d}) = \mathbf{L}\Delta\mathbf{m} + \mathbf{r}$. The second wish is really an assertion that a good way to find missing parts of a function (the model space) is to solve for the function and its PEF at the same time. We are merging the regression (??) for irregularly sampled data with the regression (1.39) for finding the prediction-error filter.

$$\begin{aligned} \mathbf{0} &\approx \mathbf{L}\Delta\mathbf{m} + (\mathbf{Lm} - \mathbf{d}) \qquad (1.45) \\ \mathbf{0} &\approx \mathbf{A}\Delta\mathbf{m} + \mathbf{MK}\Delta\mathbf{a} + (\mathbf{Am} \text{ or } \mathbf{Ma}) \qquad (1.46) \end{aligned}$$

Writing this out in full for 3 data points and 6 model values on a uniform mesh and a PEF of 3 terms we have

$$
\left[\begin{array}{cccccc|ccc}
.8 & .2 & \cdot & \cdot & \cdot & \cdot & & & \\
\cdot & \cdot & 1 & \cdot & \cdot & \cdot & & & \\
\cdot & \cdot & \cdot & \cdot & .5 & .5 & & & \\
\hline
a_0 & \cdot & \cdot & \cdot & \cdot & \cdot & m_0 & \cdot & \cdot \\
a_1 & a_0 & \cdot & \cdot & \cdot & \cdot & m_1 & m_0 & \cdot \\
a_2 & a_1 & a_0 & \cdot & \cdot & \cdot & m_2 & m_1 & m_0 \\
\cdot & a_2 & a_1 & a_0 & \cdot & \cdot & m_3 & m_2 & m_1 \\
\cdot & \cdot & a_2 & a_1 & a_0 & \cdot & m_4 & m_3 & m_2 \\
\cdot & \cdot & \cdot & a_2 & a_1 & a_0 & m_5 & m_4 & m_3 \\
\cdot & \cdot & \cdot & \cdot & a_2 & a_1 & \cdot & m_5 & m_4 \\
\cdot & \cdot & \cdot & \cdot & \cdot & a_2 & \cdot & \cdot & m_5 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}\right]
\left[\begin{array}{cc} \mathbf{I} & \mathbf{0} \\ \mathbf{0} & \mathbf{K} \end{array}\right]
\left[\begin{array}{c} \Delta m_0 \\ \Delta m_1 \\ \Delta m_2 \\ \Delta m_3 \\ \Delta m_4 \\ \Delta m_5 \\ \Delta m_6 \\ \hline \Delta a_0 \\ \Delta a_1 \\ \Delta a_2 \end{array}\right]
+
\left[\begin{array}{c} r_{d0} \\ r_{d1} \\ r_{d2} \\ \hline r_{m0} \\ r_{m1} \\ r_{m2} \\ r_{m3} \\ r_{m4} \\ r_{m5} \\ r_{m6} \\ r_{m7} \\ r_{m8} \\ 0 \end{array}\right]
\approx \mathbf{0} \tag{1.47}
$$

where r_m is the convolution of the filter a_t and the model m_t, where r_d is the data misfit $\mathbf{r} = \mathbf{Lm} - \mathbf{d}$, and where $\mathbf{K}$ was defined in equation (1.13).

Before you begin this **nonlinear regression** you need some starting guesses for $\mathbf{m}$ and $\mathbf{a}$. The guess $\mathbf{m} = 0$ is satisfactory (as explained later). For the first guess of the filter, I suggest you load it up with $\mathbf{a} = (1, -2, 1)$ as I did for the examples here. Then we begin the iteration loop in subroutine `levint1()` (page 19) by computing the two components of the residual vector, $\mathbf{r}_d = \mathbf{Lm} - \mathbf{d}$ and $\mathbf{r}_m = \mathbf{Am}$. Next, the steps are the ones of usual CG, using the adjoint operator on the residuals to find the gradient direction

$$
\left[\begin{array}{c} \Delta\mathbf{m} \\ \Delta\mathbf{a} \end{array}\right] \quad \longleftarrow \quad \left[\begin{array}{cc} \mathbf{L}' & \mathbf{A}' \\ \mathbf{0} & \mathbf{K}'\mathbf{M}' \end{array}\right] \left[\begin{array}{c} \mathbf{r}_d \\ \mathbf{r}_m \end{array}\right] \tag{1.48}
$$

Besides the usual constraints on filters and known data, I do not permit the filter to change in an early group of iterations while the missing data is roughed in. Since this is a nonlinear problem, you might fear significant dependence on the starting value of $\mathbf{m}$. Restraining the filter to be unchanging $\Delta\mathbf{a} = \mathbf{0}$ in early iterations changes the problem to a linear one in those early iterations and I found the bad effects of very poor starting guesses were strongly diminished by this restraint.

Then using $\Delta\mathbf{m}$ and $\Delta\mathbf{a}$ we find the implied changes in the residual:

$$
\left[\begin{array}{c} \Delta\mathbf{r}_d \\ \Delta\mathbf{r}_m \end{array}\right] \quad \longleftarrow \quad \left[\begin{array}{cc} \mathbf{L} & \mathbf{0} \\ \mathbf{A} & \mathbf{MK} \end{array}\right] \left[\begin{array}{c} \Delta\mathbf{m} \\ \Delta\mathbf{a} \end{array}\right] \tag{1.49}
$$

Finally, as ever, we use subroutine `cgplus()` (page ??) for the selecting the step size and updating the solution.

```
# levint1 -- LEVeler INverse INTerpolation in 1-D while finding PEF.
#
subroutine levint1( aa, a1, ord, dd,   nd, o1,d1, mm,m1, niter)
integer iter, i,        a1,             nd,           m1, niter
real                  aa(a1),ord(nd),dd(nd),o1,d1, mm(m1)
integer nx, ax, mx, nr, rd,rm
temporary real  xx(m1+a1),   rr( nd+m1+a1)
```

```
temporary real  dx(m1+a1),    dr( nd+m1+a1)
temporary real  sx(m1+a1),    sr( nd+m1+a1)
                nx=m1+a1;     nr= nd+m1+a1;   mx=1;   ax=1+m1;    rd=1;  rm=1+nd
#       call null( aa, a1); aa(1)=1;  aa(2)=-2.; aa(3)=1.       #  a = (1,-2,1)
        call copy( m1, mm, xx(mx))
        call copy( a1, aa, xx(ax))
do iter= 0, niter {                   call snap('movie.H', m1, 1, mm) # output movie
        call negcopy(        nd,          dd,        rr(rd))     #        -d
        call lint1( 0, 1, o1,d1, ord,    mm, m1, rr(rd), nd)     #  r_d = -d+Lm
        call tcai1( 0, 0, aa,a1,         mm, m1, rr(rm))         #  r_m =    Am
        call lint1( 1, 0, o1,d1, ord, dx(mx),m1, rr(rd), nd)     #     L'r_d
        call tcai1( 1, 1, aa,a1,      dx(mx),m1, rr(rm))         # dm= L'r_d +A'r_m
        call tcaf1( 1, 0, mm,m1,      dx(ax),a1, rr(rm))         # da=        M'r_m
        if( iter < 20)   do i=1,a1 { dx(m1+i) = 0. }             # da = 0
                                     dx(m1+1) = 0.               # aa(1)=1.
        call lint1( 0, 0, o1,d1, ord, dx(mx),m1, dr(rd), nd)     # d r_d = L dm
        call tcai1( 0, 0, aa,a1,      dx(mx),m1, dr(rm))         # d r_m = A dm
        call tcaf1( 0, 1, mm,m1,      dx(ax),a1, dr(rm))         # d r_m = A dm+M da
        call cgplus( iter, nx, xx, dx, sx,
                               nr, rr, dr, sr )
        call copy( m1, xx(mx), mm)
        call copy( a1, xx(ax), aa)
        }
return; end
```

1.7.3 Unresolved computational issues

Theory offers little assurance about the behavior of nonlinear optimization problems such as we solve here. I breathed a sigh of relief when my first code gave satisfactory results. After it worked, I stressed it with inappropriate scaling, too long filters, insufficient data, etc, to see what happens.

First, (limited) experience showed that the data should be scaled to about a maximum value of unity (not unlike the constrained filter coefficient). Second, notice that the linear interpolation operator **L** is devoid of physical dimensions and is about unit scale. Likewise, the PEF is dimensionless and begins with a "1.0". Thus the operators seem about the same scale. Third, I investigated revising operator scales by beginning the PEF with an ϵ instead of a "1.0". I found unsatisfactory convergence when ϵ got outside the range $0.2 < \epsilon < 2.0$, and fairly consistent behavior inside that range, so I set $\epsilon = 1.0$ and turned my attention elsewhere.

I experimented with variable numbers of filter coefficients taking values of `na` between 3 and 20. I was pleased to discover only moderate sensitivity of the solution to the filter size and I plan to do most future work with 3-5 coefficients.

Since the problem is nonlinear, there is no assurance that the recursion will converge to a global solution and I cannot know whether the solutions shown in the figures correspond to global solutions. My experience, however, was that where there were "enough" data samples, the convergence was unequivocal, and seemed insensitive to `niter`, `na`, and ϵ, Where there were not "enough" data samples, results seemed dependent on those variables.

Also since the problem is nonlinear, the required **iteration count** is not known. I started by using double that required by the linear theory, but as I watched movies of the interpolated signal as a function of iteration, I saw convergence was generally much more rapid, especially where there were enough data points. I changed the default number of iterations one and a half iterations per unknown. These plots required less than a second of computer time so I had little motivation to take advantage of the fact that most of the "action" is in the first ten iterations. Efficiency will be a much bigger issue with 2-D and 3-D models, and then we may return to 1-D to gain rapid experience with convergence issues.

Since "most physical functions are smooth" I tried initializing the PE filter to $(1, -2, 1)$ but this improved first guess had little effect beyond the first iteration. To successfully impose this prior knowledge, I added the constraint that the PE filter begin at $(1, -2, 1)$ and not change $\Delta \mathbf{a} = \mathbf{0}$ in the early iterations while the model $\mathbf{m}$ accommodates itself to this prior filter. Since linear theory applies during these early iterations, we can hope it brings us to a reasonably good position before nonlinear theory takes over.

I experimented by running first to completion with $(1, -1)$ or $(1, -.5, -.5)$. This failed because these filters introduced high frequencies (such as corners) that surprisingly would not melt away when the filter coefficients were set free. This reminds us that local minima are a reality and could mean (1) it got stuck in a local minimum, or (2) that we really do not seek the global minimum sum of squares, but we want a local minimum near a smooth solution. This phenomena appears again in experiments in 2-D in chapter **??** where after many iterations with the Laplacian filter, the wavefield "froze" and would not move when the filter was restarted from an impulse. This puzzle requires further study. Perhaps ϵ holds the key.

1.7.4 Risky ways to do nonlinear optimization

I have noticed that some geophysicists have adopted a risky method of nonlinear optimization, which is not advocated in the professional optimization literature. This risky method is to linearize a problem (with a multiparameter model space), then solve the linearized problem, then relinearize, etc. The safer method is to relinearize after each step of CG.

An instructive example I learned about many years ago was earthquake epicenter location. Model space is latitude, longitude, and origin time. When people added a new variable, the depth, the solutions went wild until they learned to restrict the depth to zero until the other three parameters were stabilized. Apparently the instability stems from the fact that depth and origin time affect distant receivers in a similar way. In another case, one of my students had difficulty when trying to remake Figure 1.5 by the risky method. Now you have been warned.