Chapter 1

Empty bins and inverse interpolation

Let us review the big picture. In Chapter **??** we developed adjoints and in Chapter **??** we developed inverse operators. Logically, correct solutions come only through inversion. Real life, however, seems nearly the opposite. This is puzzling but intriguing.

Every time you fill your car with gasoline, it derives much more from the adjoint than from inversion. I refer to the fact that "practical seismic data processing" relates much more to the use of adjoints than of inverses. It has been widely known for about the last 15 years that medical imaging and all basic image creation methods are like this. It might seem that an easy path to fame and profit would be to introduce the notion of inversion, but it is not that easy. Both cost and result quality enter the picture.

First consider cost. For simplicity, consider a data space with N values and a model (or image) space of the same size. The computational cost of applying a dense adjoint operator increases in direct proportion to the number of elements in the matrix, in this case N^2. To achieve the minimum discrepancy between theoretical data and observed data (inversion) theoretically requires N iterations raising the cost to N^3.

Consider an image of size $m \times m = N$. Continuing, for simplicity, to assume a dense matrix of relations between model and data, the cost for the adjoint is m^4 whereas the cost for inversion is m^6. We'll consider computational costs for the year 2000, but noticing that costs go as the sixth power of the mesh size, the overall situation will not change much in the foreseeable future. Suppose you give a stiff workout to a powerful machine; you take an hour to invert a 4096×4096 matrix. The solution, a vector of 4096 components could be laid into an image of size $64 \times 64 = 2^6 \times 2^6 = 4096$. Here is what we are looking at for costs:

adjoint cost	$(m \times m)^2$	$(512 \times 512)^2$	$(2^9 2^9)^2$	2^{36}
inverse cost	$(m \times m)^3$	$(64 \times 64)^3$	$(2^6 2^6)^3$	2^{36}

These numbers tell us that for applications with dense operators, the biggest images that we are likely to see coming from inversion methods are 64×64 whereas those from adjoint methods are 512×512. For comparison, the retina of your eye is comparable to your computer screen at

1000×1000. We might summarize by saying that while adjoint methods are less than perfect, inverse methods are "legally blind" :-)

http://sepwww.stanford.edu/sep/jon/family/jos/gifmovie.html holds a movie blinking between Figures 1.1 and 1.2.

Figure 1.1: Jos greets Andrew, "Welcome back Andrew" from the Peace Corps. At a resolution of 512×512, this picture is about the same as the resolution as the paper it is printed on, or the same as your viewing screen, if you have scaled it to 50% of screen size. $\boxed{\text{iin-512x512}}$ [NR]

This cost analysis is oversimplified in that most applications do not require dense operators. With sparse operators, the cost advantage of adjoints is even more pronounced since for adjoints, the cost savings of operator sparseness translate directly to real cost savings. The situation is less favorable and much more muddy for inversion. The reason that Chapter 2 covers iterative methods and neglects exact methods is that in practice iterative methods are not run to their theoretical completion but they run until we run out of patience.

Cost is a big part of the story, but the story has many other parts. Inversion, while being the only logical path to the best answer, is a path littered with pitfalls. The first pitfall is that the data is rarely able to determine a complete solution reliably. Generally there are aspects of the image that are not learnable from the data.

In this chapter we study the simplest, most transparant example of data insufficiency. Data exists at irregularly spaced positions in a plane. We set up a cartesian mesh and we discover that some of the bins contain no data points. What then?

1.1 MISSING DATA IN ONE DIMENSION

A method for restoring **missing data** is to ensure that the restored data, after specified filtering, has minimum energy. Specifying the filter chooses the interpolation philosophy. Generally

Figure 1.2: Jos greets Andrew, "Welcome back Andrew" again. At a resolution of 64×64 the pixels are clearly visible. From far the pictures are the same. From near, examine their glasses. iin-64x64 [NR]

the filter is a **roughening** filter. When a roughening filter goes off the end of smooth data, it typically produces a big end transient. Minimizing energy implies a choice for unknown data values at the end, to minimize the transient. We will examine five cases and then make some generalizations.

> A method for restoring missing data is to ensure that the restored data, after specified filtering, has **minimum energy**.

Let u denote an unknown (missing) value. The dataset on which the examples are based is $(\cdots, u, u, \; 1, u, 2, 1, 2, u, u, \cdots)$. Theoretically we could adjust the missing u values (each different) to minimize the energy in the unfiltered data. Those adjusted values would obviously turn out to be all zeros. The unfiltered data is data that has been filtered by an impulse function. To find the missing values that minimize energy out of other filters, we can use subroutine `mis1()` on page 8. Figure 1.3 shows interpolation of the dataset with $(1, -1)$ as a roughening filter. The interpolated data matches the given data where they overlap.

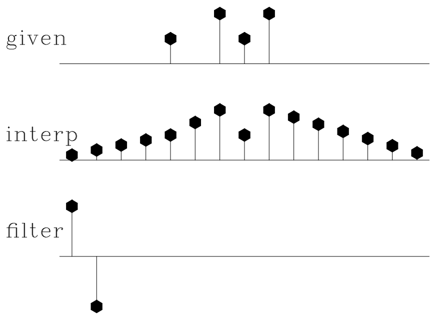

Figure 1.3: Top is given data. Middle is given data with interpolated values. Missing values seem to be interpolated by straight lines. Bottom shows the filter $(1, -1)$, whose output has minimum energy. iin-mlines90 [ER]

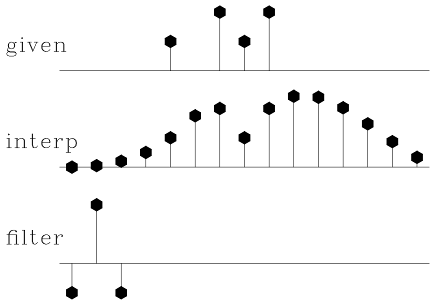

Figure 1.4: Top is the same input data as in Figure 1.3. Middle is interpolated. Bottom shows the filter $(-1,2,-1)$. The missing data seems to be interpolated by parabolas. $\boxed{\text{iin-mparab90}}$ [ER]

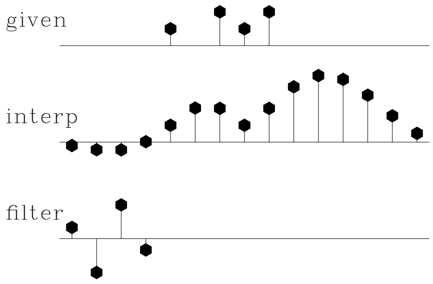

Figure 1.5: Top is the same input. Middle is interpolated. Bottom shows the filter $(1,-3,3,-1)$. The missing data is very smooth. It shoots upward high off the right end of the observations, apparently to match the data slope there. $\boxed{\text{iin-mseis90}}$ [ER]

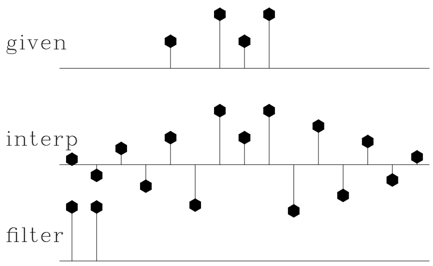

Figure 1.6: Bottom shows the filter $(1,1)$. The interpolation is rough. Like the given data itself, the interpolation has much energy at the Nyquist frequency. But unlike the given data, it has little zero-frequency energy. $\boxed{\text{iin-moscil90}}$ [ER]

Figures 1.3–1.6 illustrate that the rougher the filter, the smoother the interpolated data, and vice versa. Let us switch our attention from the residual spectrum to the residual itself. The residual for Figure 1.3 is the *slope* of the signal (because the filter $(1,-1)$ is a *first derivative*), and the slope is constant (uniformly distributed) along the straight lines where the least-squares procedure is choosing signal values. So these examples confirm the idea that the **least-squares method** abhors large values (because they are squared). Thus, least squares tends to distribute residuals uniformly in both time and frequency to the extent allowed by the **constraint**s.

This idea helps us answer the question, what is the best filter to use? It suggests choosing the filter to have an amplitude spectrum that is inverse to the spectrum we want for the interpolated data. A systematic approach is given in chapter 6, but I offer a simple subjective analysis here: Looking at the data, we see that all points are positive. It seems, therefore, that the data is rich in low frequencies; thus the filter should contain something like $(1,-1)$, which vanishes at zero frequency. Likewise, the data seems to contain Nyquist frequency, so the filter should contain $(1,1)$. The result of using the filter $(1,-1)*(1,1) = (1,0,-1)$ is shown in Figure 1.7. This is my best subjective interpolation based on the idea that the missing data should look like the given data. The **interpolation** and **extrapolation**s are so good that you can hardly guess which data values are given and which are interpolated.

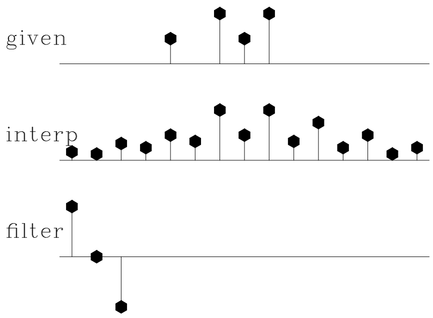

Figure 1.7: Top is the same as in Figures 1.3 to 1.6. Middle is interpolated. Bottom shows the filter $(1,0,-1)$, which comes from the coefficients of $(1,-1)*(1,1)$. Both the given data and the interpolated data have significant energy at both zero and Nyquist frequencies. iin-mbest90 [ER]

1.1.1 Missing-data program

Now let us see how Figures 1.3-1.7 could have been calculated and how they were calculated. They could have been calculated with matrices, in which matrices were pulled apart according to subscripts of known or missing data; instead I computed them with operators, and applied only operators and their adjoints. First we inspect the matrix approach because it is more conventional.

Matrix approach to missing data

Customarily, we have referred to data by the symbol $\mathbf{d}$. Now that we are dividing the data space into two parts, known and unknown (or missing), we will refer to this complete space as the model (or map) space $\mathbf{m}$.

There are 15 data points in Figures 1.3-1.7. Of the 15, 4 are known and 11 are missing. Denote the known by k and the missing by u. Then the sequence of missing and known is $(u,u,u,u,k,u,k,k,k,u,u,u,u,u,u)$. Because I cannot print 15×15 matrices, please allow me to describe instead a data space of 6 values $(m_1,m_2,m_3,m_4,m_5,m_6)$ with known values only m_2 and m_3, that is arranged like (u,k,k,u,u,u).

Our approach is to minimize the energy in the residual, which is the filtered map (model) space. We state the fitting goals $\mathbf{0}\approx \mathbf{Am}$ as

$$\begin{bmatrix} 0\\0\\0\\0\\0\\0\\0\\0 \end{bmatrix} \quad \approx \quad \mathbf{r} \quad = \quad \begin{bmatrix} a_1 & 0 & 0 & 0 & 0 & 0\\ a_2 & a_1 & 0 & 0 & 0 & 0\\ a_3 & a_2 & a_1 & 0 & 0 & 0\\ 0 & a_3 & a_2 & a_1 & 0 & 0\\ 0 & 0 & a_3 & a_2 & a_1 & 0\\ 0 & 0 & 0 & a_3 & a_2 & a_1\\ 0 & 0 & 0 & 0 & a_3 & a_2\\ 0 & 0 & 0 & 0 & 0 & a_3 \end{bmatrix} \begin{bmatrix} m_1\\m_2\\m_3\\m_4\\m_5\\m_6 \end{bmatrix} \tag{1.1}$$

We rearrange the above fitting goals, bringing the columns multiplying known data values (m_2 and m_3) to the left, getting $\mathbf{y}=-\mathbf{A}_k\mathbf{m}_k\approx\mathbf{A}_u\mathbf{m}_u$.

$$\begin{bmatrix} y_1\\y_2\\y_3\\y_4\\y_5\\y_6\\y_7\\y_8 \end{bmatrix} \quad = \quad - \begin{bmatrix} 0 & 0\\ a_1 & 0\\ a_2 & a_1\\ a_3 & a_2\\ 0 & a_3\\ 0 & 0\\ 0 & 0\\ 0 & 0 \end{bmatrix} \begin{bmatrix} m_2\\ m_3 \end{bmatrix} \quad \approx \quad \begin{bmatrix} a_1 & 0 & 0 & 0\\ a_2 & 0 & 0 & 0\\ a_3 & 0 & 0 & 0\\ 0 & a_1 & 0 & 0\\ 0 & a_2 & a_1 & 0\\ 0 & a_3 & a_2 & a_1\\ 0 & 0 & a_3 & a_2\\ 0 & 0 & 0 & a_3 \end{bmatrix} \begin{bmatrix} m_1\\m_4\\m_5\\m_6 \end{bmatrix} \tag{1.2}$$

This is the familiar form of an overdetermined system of equations $\mathbf{y}\approx\mathbf{A}_u\mathbf{m}_u$ which we could solve for $\mathbf{m}_u$ as illustrated earlier by conjugate directions, or by a wide variety of well-known methods.

The trouble with this matrix approach is that it is awkward to program the partitioning of the operator into the known and missing parts, particularly if the application of the operator uses arcane techniques, such as those used by the fast–Fourier-transform operator or various numerical approximations to differential or partial differential operators that depend on regular data sampling. Even for the modest convolution operator, we already have a library of convolution programs that handle a variety of end effects, and it would be much nicer to use the library as it is rather than recode it for all possible geometrical arrangements of missing data values.

Note: Here I take the main goal to be the clarity of the code, not the efficiency or accuracy of the solution. So, if your problem consumes too many resources, and if you have many more known points than missing ones, maybe you should fit $\mathbf{y}\approx\mathbf{A}_u\mathbf{m}_u$ and ignore the suggestions below.

Operator approach to missing data

For the operator approach to the fitting goal $-\mathbf{A}_k\mathbf{m}_k \approx \mathbf{A}_u\mathbf{m}_u$ we rewrite it as $-\mathbf{A}_k\mathbf{m}_k \approx \mathbf{AJm}$ where

$$-\mathbf{A}_k\mathbf{m}_k \quad \approx \quad \begin{bmatrix} a_1 & 0 & 0 & 0 & 0 & 0 \\ a_2 & a_1 & 0 & 0 & 0 & 0 \\ a_3 & a_2 & a_1 & 0 & 0 & 0 \\ 0 & a_3 & a_2 & a_1 & 0 & 0 \\ 0 & 0 & a_3 & a_2 & a_1 & 0 \\ 0 & 0 & 0 & a_3 & a_2 & a_1 \\ 0 & 0 & 0 & 0 & a_3 & a_2 \\ 0 & 0 & 0 & 0 & 0 & a_3 \end{bmatrix} \begin{bmatrix} 1 & . & . & . & . & . \\ . & 0 & . & . & . & . \\ . & . & 0 & . & . & . \\ . & . & . & 1 & . & . \\ . & . & . & . & 1 & . \\ . & . & . & . & . & 1 \end{bmatrix} \begin{bmatrix} m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \end{bmatrix} \quad = \quad \mathbf{AJm} \tag{1.3}$$

Notice the introduction of the new diagonal matrix $\mathbf{J}$, called a **mask**ing matrix or a **constraint-mask** matrix because it multiplies constrained variables by zero leaving freely adjustable variables untouched. Experience shows that better name than “mask matrix” is “**selector** matrix” because what comes out of it, that which is selected, is a less-confusing name for it than which is rejected. With a selector matrix the whole data space seems freely adjustable, both the missing data values and known values. We see that the CD method does not change the known (constrained) values. In general, we derive the fitting goal (1.3) by

$$\mathbf{0} \quad \approx \quad \mathbf{Am} \tag{1.4}$$
$$\mathbf{0} \quad \approx \quad \mathbf{A}(\mathbf{J}+(\mathbf{I}-\mathbf{J}))\mathbf{m} \tag{1.5}$$
$$\mathbf{0} \quad \approx \quad \mathbf{AJm}+\mathbf{A}(\mathbf{I}-\mathbf{J})\mathbf{m} \tag{1.6}$$
$$\mathbf{0} \quad \approx \quad \mathbf{AJm}+\mathbf{Am}_{\text{known}} \tag{1.7}$$
$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{AJm}+\mathbf{r}_0 \tag{1.8}$$

As usual, we find a direction to go $\Delta\mathbf{m}$ by the gradient of the residual energy.

$$\Delta\mathbf{m} \quad = \quad \frac{\partial}{\partial\mathbf{m}'}\mathbf{r}'\mathbf{r} \quad = \quad \left(\frac{\partial}{\partial\mathbf{m}'}\mathbf{r}'\right)\mathbf{r} \quad = \quad \left(\frac{\partial}{\partial\mathbf{m}'}(\mathbf{m}'\mathbf{J}'\mathbf{A}'+\mathbf{r}_0')\right)\mathbf{r} \quad = \quad \mathbf{J}'\mathbf{A}'\mathbf{r} \tag{1.9}$$

We begin the calculation with the known data values where missing data values are replaced by zeros, namely $(\mathbf{I}-\mathbf{J})\mathbf{m}$. Filter this data, getting $\mathbf{A}(\mathbf{I}-\mathbf{J})\mathbf{m}$, and load it into the residual $\mathbf{r}_0$. With this initialization completed, we begin an iteration loop. First we pass the residual through the adjoint

$$\Delta\mathbf{m} \quad \longleftarrow \quad \mathbf{J}'\mathbf{A}'\mathbf{r} \tag{1.10}$$

$\mathbf{A}'$ applies a *crosscorrelation* of the filter to the residual and then $\mathbf{J}'$ sets to zero any changes proposed to known data values. Next, compute the change in residual $\Delta\mathbf{r}$ from the proposed change in the data $\Delta\mathbf{m}$.

$$\Delta\mathbf{r} \quad \longleftarrow \quad \mathbf{AJ}\Delta\mathbf{m} \tag{1.11}$$

This applies the filtering again. Then use the method of steepest descent (or conjugate direction) to choose the appropriate scaling (or inclusion of previous step) of $\Delta\mathbf{m}$ and $\Delta\mathbf{r}$, and update $\mathbf{m}$ and $\mathbf{r}$ accordingly and iterate.

I could have passed a new operator **AJ** into the old solver, but found it worthwhile to write a new, more powerful solver having built-in constraints. To introduce the masking operator **J** into the `solver` subroutine on page ??, I introduce an optional input parameter `known`, which contains a logical array of the model size. The values of `known` are `.true.` for the known data locations, and `.false.` for the unknown data. Two lines in the `simple_solver` module on page ??

```
stat = oper( T, F, g, rr)                                # dm = A' r
stat = oper( F, F, g, gg)                                # dr = A  dm
```

become three lines in the standard library module `solver_mod`. (Here I changed the comment lines to correspond to the notation of the missing-data application; $\Delta\mathbf{m}$ is `g` and $\Delta\mathbf{r}$ is `gg`.)

```
stat = oper( T, F, g, rr)                                # dm  = A' r
if ( present( known))  where(known)   g = 0.             # dm  = JJ' dm
stat = oper( F, F, g, gg)                                # dr =  A   dm
```

In Fortran77, I simply multiplied by **J**. In Fortran90, subroutines can have optional arguments. The expression `if( present( known))` says, "if the argument `known`, a logical array, is an argument of the call." The expression, `where( known) dm = 0` means that each component of the logical array `known` is examined; if the value of that component is `.true.` then the corresponding component of $\Delta\mathbf{m}$ is set to zero. In other words, we are not going to change the given, known data values.

The subroutine to find missing data is `mis1()`. It assumes that zero values in the input data correspond to missing data locations. It uses our convolution operator `tcai1()` on page ??. You can also check the Index for other **operators** and **modules**.

```
module mis_mod {
  use tcai1+cgstep_mod+solver_mod
contains
# fill in missing data on 1-axis by minimizing power out of a given filter.
  subroutine mis1 ( niter, mm, aa) {
    integer,              intent (in)      :: niter   # number of iterations
    real, dimension (:), pointer           :: aa      # roughening filter
    real, dimension (:), intent (in out)   :: mm      # in - data with zeroes
                                                      # out - interpolated
    real, dimension (size(mm) + size(aa)) :: zero    # filter output
    call tcai1_init( aa);                       zero = 0.
    call solver (tcai1_lop,cgstep,mm, zero, niter,x0 =mm,known =(mm/=0.))
    call cgstep_close ()
    }
}
```

Here `known` is declared in `solver_mod` to be a logical vector. The call argument `known=(xx/=0.)` sets the mask vector components to `.true.` where components of **x** are nonzero and sets it to `.false.` where the components are zero.

I sought reference material on conjugate gradients with constraints and didn't find anything, leaving me to fear that this chapter was in error and that I had lost the magic property of convergence in a finite number of iterations. I tested the code and it did converge in a finite number of iterations. The explanation is that these constraints are almost trivial. We pretended we had extra variables, and computed a $\Delta\mathbf{m}$ for each of them. Then we set the $\Delta\mathbf{m}$ to zero, hence making no changes to anything, like as if we had never calculated the extra $\Delta\mathbf{m}$'s.

EXERCISES:

1 Figures 1.3–1.6 seem to extrapolate to vanishing signals at the side boundaries. Why is that so, and what could be done to leave the sides unconstrained in that way?

2 Show that the interpolation curve in Figure 1.4 is not parabolic as it appears, but cubic. (HINT: First show that $(\nabla^2)'\nabla^2 u = \mathbf{0}$.)

3 Verify by a program example that the number of iterations required with simple constraints is the number of free parameters.

4 A signal on a uniform mesh has missing values. How should we estimate the mean?

1.2 WELLS NOT MATCHING THE SEISMIC MAP

Accurate knowledge comes from a **well**, but wells are expensive and far apart. Less accurate knowledge comes from surface seismology, but this knowledge is available densely in space and can indicate significant **trend**s between the wells. For example, a prospective area may contain 15 wells but 600 or more seismic stations. To choose future well locations, it is helpful to match the known well data with the seismic data. Although the seismic data is delightfully dense in space, it often mismatches the wells because there are systematic differences in the nature of the measurements. These discrepancies are sometimes attributed to velocity **anisotropy**. To work with such measurements, we do not need to track down the physical model, we need only to merge the information somehow so we can appropriately **map** the trends between wells and make a proposal for the next drill site. Here we consider only a scalar value at each location. Take **w** to be a vector of 15 components, each component being the seismic travel time to some fixed depth in a well. Likewise let **s** be a 600-component vector each with the seismic travel time to that fixed depth as estimated wholly from surface seismology. Such empirical corrections are often called "**fudge factor**s". An example is the Chevron oil field in Figure 1.8. The binning of the seismic data in Figure 1.8 is not really satisfactory when we have available the techniques of missing data estimation to fill the empty bins. Using the ideas of subroutine `mis1()` on the facing page, we can extend the seismic data into the empty part of the plane. We use the same principle that we minimize the energy

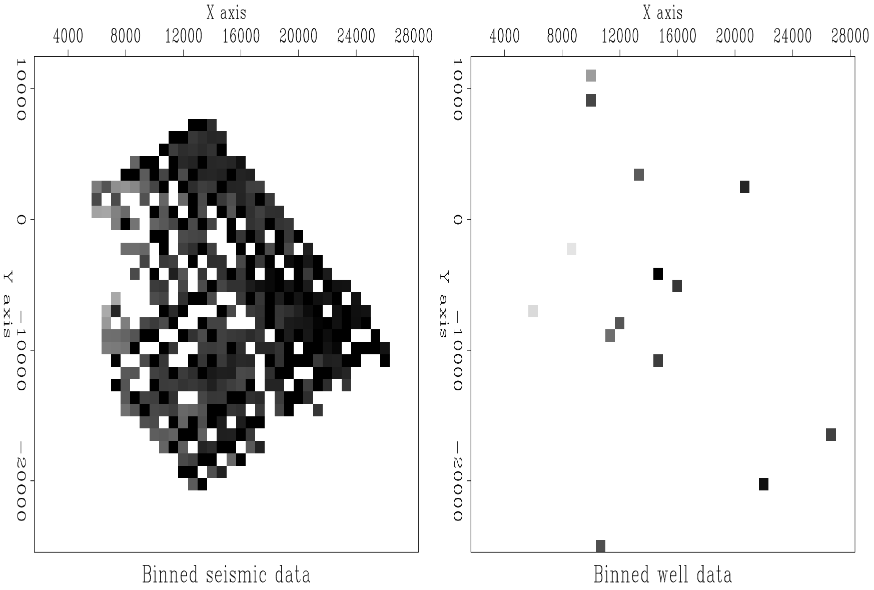

Figure 1.8: Binning by data push. Left is seismic data. Right is well locations. Values in bins are divided by numbers in bins. (Toldi) iin-wellseis90 [ER]

in the filtered map where the map must match the data where it is known. I chose the filter $\mathbf{A} = \nabla'\nabla = -\nabla^2$ to be the Laplacian operator (actually, its negative) to obtain the result in Figure 1.9.

Figure 1.9 also involves a **boundary condition calculation**. Many differential equations have a solution that becomes infinite at infinite distance, and in practice this means that the largest solutions may often be found on the boundaries of the plot, exactly where there is the least information. To obtain a more pleasing result, I placed artificial “average” data along the outer boundary. Each boundary point was given the value of an average of the interior data values. The average was weighted, each weight being an inverse power of the separation distance of the boundary point from the interior point.

Parenthetically, we notice that all the unknown interior points could be guessed by the same method we used on the outer boundary. After some experience guessing what inverse power would be best for the weighting functions, I do not recommend this method. Like gravity, the forces of interpolation from the weighted sums are not blocked by intervening objects. But the temperature in a house is not a function of temperature in its neighbor’s house. To further isolate the more remote points, I chose weights to be the inverse fourth power of distance.

After the gaps are filled in the seismic information **s**, we are prepared to find another map **m** that matches the wells **w** exactly and tries to match the seismic information elsewhere. Now we must beware that the seismic information is likely to be inconsistent with the wells. We presume that the two should fit at high spatial frequencies but mismatch where there are wells. Think of a map of a model space **m** of infinitely many hypothetical wells that must match the real wells, where we have real wells. Otherwise, we seek the minimum residual **r** which is the

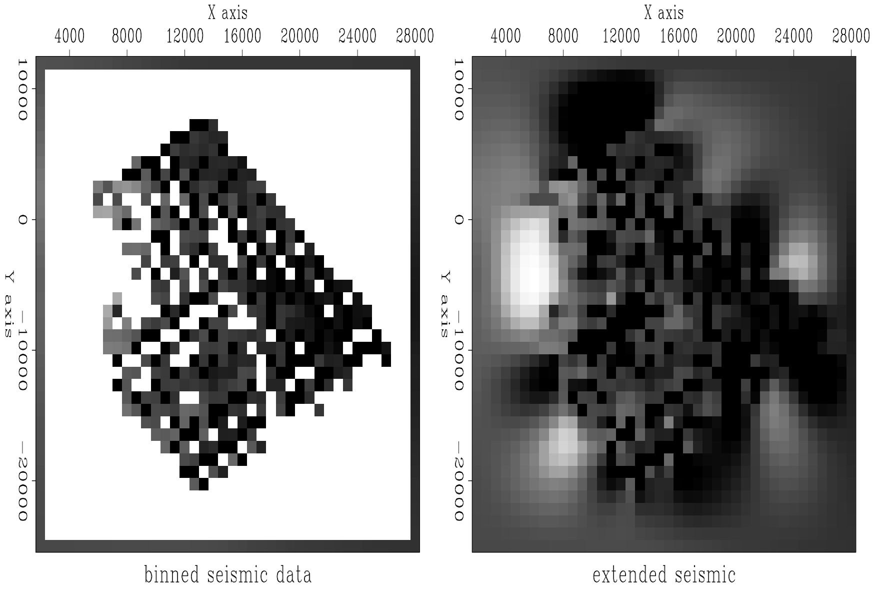

Figure 1.9: Seismic binned (left) and extended (right) by minimizing energy in $\nabla^2 \mathbf{s}$. iin-misseis90 [ER]

filtered difference between the seismic data and the map $\mathbf{m}$ of hypothetical omnipresent wells. Thus we seek to fit

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{A}(\mathbf{m}-\mathbf{s}) \tag{1.12}$$

along with the constraint

$$\mathbf{0} \quad = \quad \mathbf{J}_w(\mathbf{m}-\mathbf{w}) \tag{1.13}$$

which means the model matches the wells where we have wells. To solve this problem, we convert it to one that we have already solved. We define a new variable $\mathbf{y}$

$$\mathbf{y} \quad = \quad \mathbf{m}-\mathbf{s} \tag{1.14}$$

and we seek to minimize

$$\mathbf{0} \quad \approx \quad \mathbf{r} \quad = \quad \mathbf{A}\mathbf{y} \tag{1.15}$$

Initialize $\mathbf{y}_0$ so that `where(known) y0=(w-s)`, in other words,

$$\mathbf{y}_0 \quad = \quad \mathbf{J}_w(\mathbf{w}-\mathbf{s}) \tag{1.16}$$

After we optimize the fitting (1.15) for $\mathbf{y}$, we have our solution $\mathbf{m}=\mathbf{y}+\mathbf{s}$.

We have already coded a 2-D gradient operator `igrad2` on page ??. Let us combine it with its adjoint to get the 2-D laplacian operator. (You might notice that the laplacian operator is "self-adjoint" meaning that the operator does the same calculation that its adjoint does. Any operator of the form $\mathbf{A}'\mathbf{A}$ is self-adjoint because $(\mathbf{A}'\mathbf{A})' = \mathbf{A}'\mathbf{A}'' = \mathbf{A}'\mathbf{A}$.)

```
module laplac2 {                                     # Laplacian operator in 2-D
use igrad2
logical, parameter, private  :: T = .true., F = .false.
real, dimension (m1*m2*2), allocatable :: tmp
#%_init    (m1, m2)
    integer m1, m2
    call igrad2_init (m1, m2)
#%_lop (x, y)
    integer stat1, stat2
    if( adj) {
        stat1 = igrad2_lop (  F,  F, y, tmp)   # tmp = grad y
        stat2 = igrad2_lop (  T,  T, x, tmp)   # x = x + grad' tmp
    } else   {
        stat1 = igrad2_lop (  F,  F, x, tmp)   # tmp = grad x
        stat2 = igrad2_lop (  T,  T, y, tmp)   # y = y + grad' tmp
    }
}
```

Subroutine `lapfill2()` on this page is the same idea as `mis1()` on page 8 except that the filter **A** has been specialized to the laplacian implemented by module `laplac2` on the preceding page. Subroutine `lapfill2()` solves two problems: (1) extending the seismic data to fill space, and (2) fitting the map approximately to the seismic data while exactly to the wells.

```
module lapfill {  # fill empty 2-D bins by minimum output of Laplacian operator
  use laplac2
  use cgstep_mod
  use solver_mod
contains
  subroutine lapfill2( niter, m1, m2, yy, mfixed) {
    integer,                     intent (in)     :: niter, m1, m2
    logical, dimension (m1*m2), intent (in)     :: mfixed # mask for known
    real,    dimension (m1*m2), intent (in out) :: yy     # model
    real,    dimension (m1*m2)                  :: zero   # laplacian output
    call laplac2_init ( m1,m2);            zero = 0.      # initialize
    call solver ( laplac2_lop, cgstep, yy,  zero, niter,
                                 x0 = yy,  known = mfixed)
    call laplac2_close ()                                 # garbage collection
    call cgstep_close ()                                  # garbage collection
  }
}
```

The final map is shown in Figure 1.10.

Results can be computed with various filters. I tried both ∇^2 and ∇. There are disadvantages of each, ∇ being too cautious and ∇^2 perhaps being too aggressive. Figure 1.11 shows the difference $\mathbf{y}$ between the extended seismic data and the extended wells. Notice that for ∇ the difference shows a localized "tent pole" disturbance about each well. For ∇^2 there could be large overshoot between wells, especially if two nearby wells have significantly different values. I don't see that problem here.

To understand the behavior theoretically, recall that in one dimension the filter ∇ interpolates with straight lines and ∇^2 interpolates with cubics. This is because the fitting goal

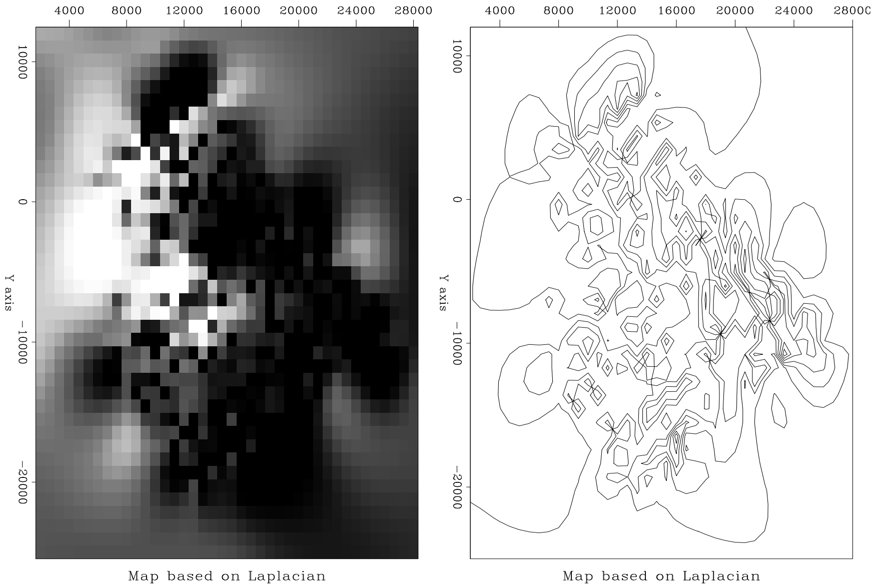

Figure 1.10: Final map based on Laplacian roughening. iin-finalmap90 [ER,M]

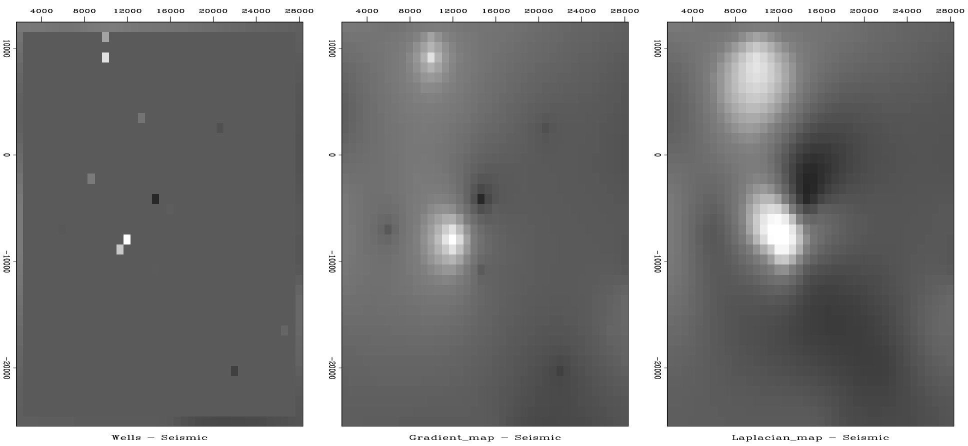

Figure 1.11: Difference between wells (the final map) and the extended seismic data. Left is plotted at the wells (with gray background for zero). Center is based on gradient roughening and shows tent-pole-like residuals at wells. Right is based on Laplacian roughening. iin-diffdiff90 [ER]

$\mathbf{0} \approx \nabla \mathbf{m}$, leads to $\frac{\partial}{\partial \mathbf{m}'} \mathbf{m}' \nabla' \nabla \mathbf{m} = \mathbf{0}$ or $\nabla' \nabla \mathbf{m} = \mathbf{0}$, whereas the fitting goal $\mathbf{0} \approx \nabla^2 \mathbf{m}$ leads to $\nabla^4 \mathbf{m} = \mathbf{0}$ which is satisfied by cubics. In two dimensions, minimizing the output of ∇ gives us solutions of Laplace's equation with sources at the known data. It is as if ∇ stretches a rubber sheet over poles at each well, whereas ∇^2 bends a stiff plate.

Just because ∇^2 gives smoother maps than ∇ does not mean those maps are closer to reality. This is a deeper topic, addressed in Chapter 6. It is the same issue we noticed when comparing figures 1.3-1.7.

1.2.1 Abandoned theory for matching wells and seismograms

Let us consider theory to construct a map $\mathbf{m}$ that fits dense seismic data $\mathbf{s}$ and the well data $\mathbf{w}$. The first goal $\mathbf{0} \approx \mathbf{L}\mathbf{m} - \mathbf{w}$ says that when we linearly interpolate from the map, we should get the well data. The second goal $\mathbf{0} \approx \mathbf{A}(\mathbf{m} - \mathbf{s})$ (where $\mathbf{A}$ is a roughening operator like ∇ or ∇^2) says that the map $\mathbf{m}$ should match the seismic data $\mathbf{s}$ at high frequencies but need not do so at low frequencies.

$$\begin{array}{lcl} \mathbf{0} & \approx & \mathbf{L}\mathbf{m} - \mathbf{w} \\ \mathbf{0} & \approx & \mathbf{A}(\mathbf{m} - \mathbf{s}) \end{array} \tag{1.17}$$

Although (1.17) is the way I originally formulated the well-fitting problem, I abandoned it for several reasons: First, the map had ample pixel resolution compared to other sources of error, so I switched from linear interpolation to binning. Once I was using binning, I had available the simpler empty-bin approaches. These have the further advantage that it is not necessary to experiment with the relative weighting between the two goals in (1.17). A formulation like (1.17) is more likely to be helpful where we need to handle rapidly changing functions; where binning is inferior to linear interpolation, perhaps in reflection seismology where high resolution is meaningful.

EXERCISES:

1 It is desired to find a compromise between the Laplacian roughener and the gradient roughener. What is the size of the residual space?

2 Like the seismic prospecting industry, you have solved a huge problem using binning. You have computer power left over to do a few iterations with linear interpolation. How much does the cost per iteration increase? Should you refine your model mesh, or can you use the same model mesh that you used when binning?

1.3 SEARCHING THE SEA OF GALILEE

Figure 1.12 shows a bottom-sounding survey of the Sea of Galilee[1] at various stages of processing. The ultimate goal is not only a good map of the depth to bottom, but images useful for the purpose of identifying **archaeological**, geological, or geophysical details of the sea bottom. The Sea of Galilee is unique because it is a *fresh*-water lake *below* sea-level. It seems to be connected to the great rift (pull-apart) valley crossing east Africa. We might delineate the Jordan River delta. We might find springs on the water bottom. We might find archaeological objects.

The raw data is 132,044 triples, (x_i, y_i, z_i), where x_i ranges over about 12 km and where y_i ranges over about 20 km. The lines you see in Figure 1.12 are sequences of data points, i.e., the track of the survey vessel. The depths z_i are recorded to an accuracy of about 10 cm.

The first frame in Figure 1.12 shows simple binning. A coarser mesh would avoid the empty bins but lose resolution. As we refine the mesh for more detail, the number of empty bins grows as does the care needed in devising a technique for filling them. This first frame uses the simple idea from Chapter **??** of spraying all the data values to the nearest bin with `bin2()` on page ?? and dividing by the number in the bin. Bins with no data obviously need to be filled in some other way. I used a missing data program like that in the recent section on "wells not matching the seismic map." Instead of roughening with a Laplacian, however, I used the gradient operator `igrad2` on page ?? The solver is `grad2fill()`.

```
module grad2fill {   # min r(m) = L J m + L known  where L is a lowcut filter.
  use igrad2
  use cgstep_mod
  use solver_mod
contains
  subroutine grad2fill2( niter, m1, m2, mm, mfixed) {
    integer,                    intent (in)     :: niter, m1,m2
    logical, dimension (m1*m2), intent (in)     :: mfixed      # mask for known
    real,    dimension (m1*m2), intent (in out) :: mm          # model
    real,    dimension (m1*m2*2)                :: yy          # lowcut output
    call igrad2_init(m1,m2);                       yy = 0.     # initialize
    call solver ( igrad2_lop, cgstep, mm, yy, niter,
                                    x0 = mm, known = mfixed )
    call cgstep_close ()
  }
}
```

The output of the roughening operator is an image, a filtered version of the depth, a filtered version of something real. Such filtering can enhance the appearance of interesting features. For example, scanning the shoreline of the roughened image (after missing data was filled), we see several ancient shorelines, now submerged.

[1] Data collected by Zvi **ben Avraham**, TelAviv University. Please communicate with him `zvi@jupiter1.tau.ac.il` for more details or if you make something publishable with his data.

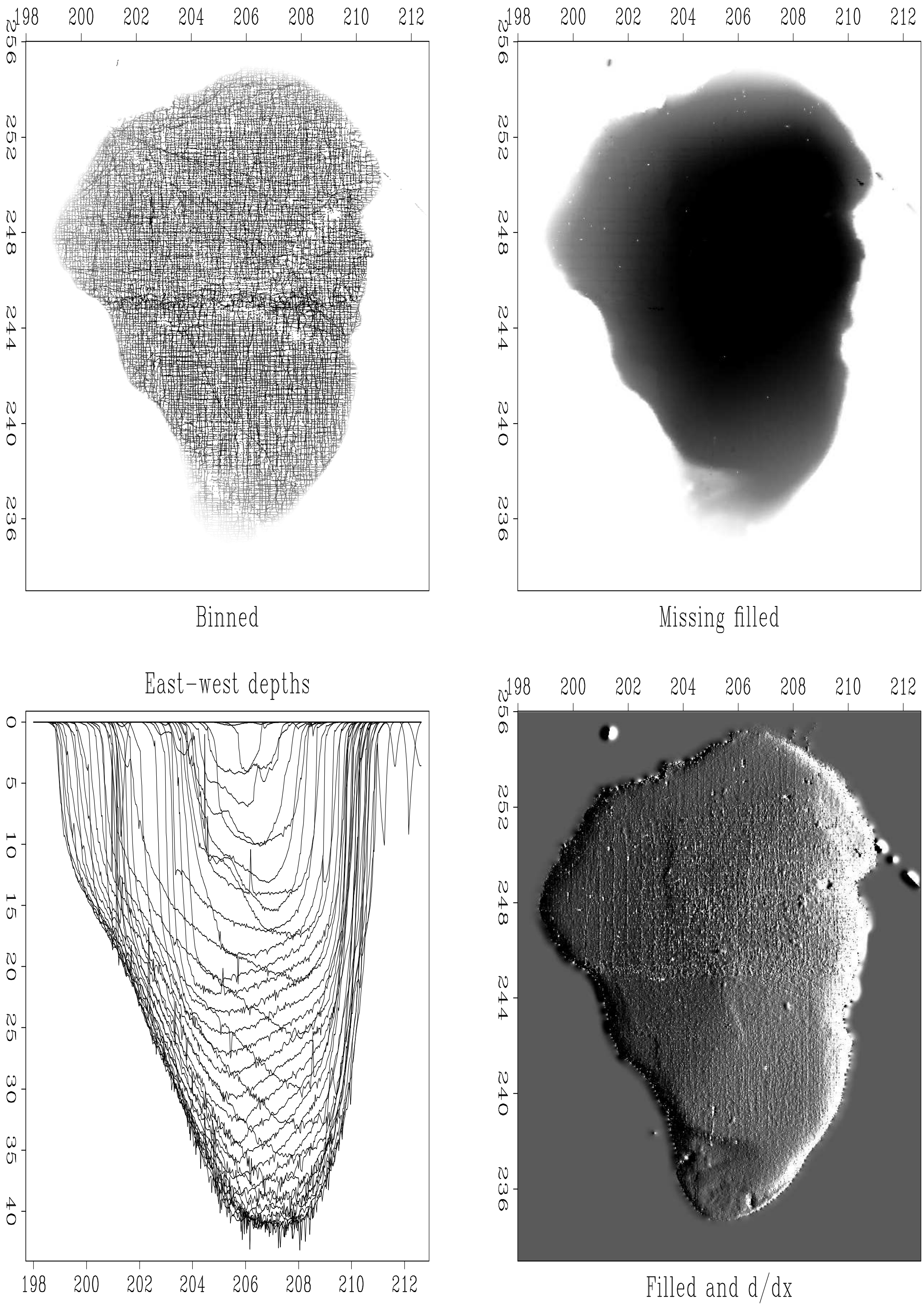

Figure 1.12: Views of the bottom of the Sea of Galilee. iin-locfil90 [ER,M]

> The adjoint is the easiest image to build. The roughened map is a better image, often more informative than the map itself.

The views expose several defects of the data acquisition and of our data processing. The impulsive glitches (St. Peter's fish?) need to be removed but we must be careful not to throw out the sunken ships along with the bad data points. Even our best image shows clear evidence of the recording vessel's tracks. Strangely, some tracks are deeper than others. Perhaps the survey is assembled from work done in different seasons and the water level varied by season. Perhaps some days the vessel was more heavily loaded and the depth sounder was on a deeper keel. As for the navigation equipment, we can see that some data values are reported outside the lake!

> A good image of the earth hides our data **acquisition footprint**.

We want the sharpest possible view of this classical site. A treasure hunt is never easy and no one guarantees we will find anything of great value but at least the exercise is a good warm-up for submarine petroleum exploration.

1.4 INVERSE LINEAR INTERPOLATION

In Chapter **??** we defined **linear interpolation** as the extraction of values from between mesh points. In a typical setup (occasionally the role of data and model are swapped), a model is given on a uniform mesh and we solve the easy problem of extracting values between the mesh points with subroutine `lint1()` on page ??. The genuine problem is the inverse problem, which we attack here. Data values are sprinkled all around, and we wish to find a function on a uniform mesh from which we can extract that data by **linear interpolation**. The adjoint operator for subroutine `lint1()` simply piles data back into its proper location in model space without regard to how many data values land in each region. Thus some model values may have many data points added to them while other model values get none. We could interpolate by minimizing the energy in the model gradient, or that in the second derivative of the model, or that in the output of any other roughening filter applied to the model.

Formalizing now our wish that data $\mathbf{d}$ be extractable by **linear interpolation** $\mathbf{L}$, from a model $\mathbf{m}$, and our wish that application of a roughening filter with an operator $\mathbf{A}$ have minimum energy, we write the fitting goals:

$$\begin{array}{lll} \mathbf{0} & \approx & \mathbf{Lm} - \mathbf{d} \\ \mathbf{0} & \approx & \mathbf{Am} \end{array} \tag{1.18}$$

Suppose we take the roughening filter to be the second difference operator $(1,-2,1)$ scaled by a constant ϵ, and suppose we have a data point near each end of the model and a third data point exactly in the middle. Then, for a model space 7 points long, the fitting goal could look like

$$
\left[\begin{array}{cccccc}
.8 & .2 & . & . & . & . \\
. & . & 1 & . & . & . \\
. & . & . & . & .5 & .5 \\
\hline
\epsilon & . & . & . & . & . \\
-2\epsilon & \epsilon & . & . & . & . \\
\epsilon & -2\epsilon & \epsilon & . & . & . \\
. & \epsilon & -2\epsilon & \epsilon & . & . \\
. & . & \epsilon & -2\epsilon & \epsilon & . \\
. & . & . & \epsilon & -2\epsilon & \epsilon \\
. & . & . & . & \epsilon & -2\epsilon \\
. & . & . & . & . & \epsilon \\
. & . & . & . & . & .
\end{array} \right]
\left[\begin{array}{c} m_0 \\ m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \end{array} \right]
-
\left[\begin{array}{c} d_0 \\ d_1 \\ d_2 \\ \hline 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{array} \right]
=
\left[\begin{array}{c} \mathbf{r}_d \\ \mathbf{r}_m \end{array} \right]
\approx \mathbf{0}
\tag{1.19}
$$

The residual vector has two parts, a data part $\mathbf{r}_d$ on top and a model part $\mathbf{r}_m$ on the bottom. The data residual should vanish except where contradictory data values happen to lie in the same place. The model residual is the roughened model.

Two fitting goals (1.18) are so common in practice that it is convenient to adopt our least-square fitting subroutine `solver` on page ?? accordingly. The modification is shown in module `reg_solver` on the current page. In addition to specifying the "data fitting" operator $\mathbf{L}$ (parameter `oper`), we need to pass the "model regularization" operator $\mathbf{A}$ (parameter `reg`) and the size of its output (parameter `nreg`) for proper memory allocation.

(When I first looked at module `reg_solver` I was appalled by the many lines of code, especially all the declarations. Then I realized how much much worse was Fortran 77 where I needed to write a new solver for every pair of operators. This one solver module works for all operator pairs and for many optimization descent strategies because these "objects" are arguments. These more powerful objects require declarations that are more complicated than the simple objects of Fortran 77. As an author I have a dilemma: To make algorithms compact (and seem simple) requires many careful definitions. When these definitions put in the code, they are careful, but the code becomes annoyingly verbose. Otherwise, the definitions must go in the surrounding natural language where they are not easily made precise.)

```
module reg_solver  {
  logical, parameter, private  :: T = .true., F = .false.
contains
  subroutine solver_reg( oper, solv, reg, nreg, x, dat, niter, eps, x0)  {
    optional                                                    :: x0
    interface {
       integer function oper(  adj, add, x, dat) {
            logical, intent( in) :: adj, add
            real, dimension( :)  :: x, dat
            }
       integer function reg(   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, x0          # data, initial
   real, dimension( :), intent( out) :: x                # solution
   real,                intent( in)  :: eps              # scaling
   integer,             intent( in)  :: niter, nreg      # iterations, size
   real, dimension( size( x))        :: g                # gradient
   real, dimension( size( dat) + nreg), target :: rr, gg # residual, conj grad
   real, dimension( :), pointer      :: rd,rm, gd,gm
   integer                           :: i, stat1, stat2, stat
   rm => rr( 1: nreg); rd => rr( 1 + nreg:)          # model and data resids
   gm => gg( 1: nreg); gd => gg( 1 + nreg:)          # model and data grads
                      rd = - dat                     # initialize r_d
   if( present( x0)) { stat1 = oper( F, T, x0, rd)   # r_d = L x0 - dat
                      stat2 =  reg( F, F, x0, rm)    # r_m = eps A x0
                      rm = rm * eps                  #
                      x = x0          }              # start with x0
   else {             x = 0.;  rm = 0.}              # start with zero
   do i = 1, niter  {
      stat1 = oper( T, F, g, rd)     # g   = L' r_d
                       gm = rm*eps   #
      stat2 =  reg( T, T, g, gm)     # g   = L' r_d + eps A' r_m
      stat1 = oper( F, F, g, gd)     # g_d = L g
      stat2 =  reg( F, F, g, gm)     # g_m = eps A g
                       gm = gm*eps   #
      stat  = solv( F, x, g, rr, gg) # step in x and rr
      }
  }
}
```

After all the definitions, we load the negative of the data into the residual. If a starting model $\mathbf{m}_0$ is present, then we update the data part of the residual $\mathbf{r}_d = \mathbf{L}\mathbf{m}_0 - \mathbf{d}$ and we load the model part of the residual $\mathbf{r}_m = \mathbf{A}\mathbf{m}_0$. Otherwise we begin from a zero model $\mathbf{m}_0 = \mathbf{0}$ and thus the model part of the residual $\mathbf{r}_m$ is also zero. After this initialization, subroutine `solver_reg()` begins an iteration loop by first computing the proposed model perturbation $\Delta\mathbf{m}$ with the adjoint operator:

$$\Delta\mathbf{m} \quad \longleftarrow \quad \begin{bmatrix} \mathbf{L}' & \mathbf{A}' \end{bmatrix} \begin{bmatrix} \mathbf{r}_d \\ \mathbf{r}_m \end{bmatrix} \tag{1.20}$$

Using this value of $\Delta\mathbf{m}$, we can find the implied change in residual $\Delta\mathbf{r}$ as

$$\Delta \begin{bmatrix} \mathbf{r}_d \\ \mathbf{r}_m \end{bmatrix} \quad \longleftarrow \quad \begin{bmatrix} \mathbf{L} \\ \mathbf{A} \end{bmatrix} \Delta\mathbf{m} \tag{1.21}$$

and the last thing in the loop is to use the optimization step function `solv()` to choose the length of the step size and to choose how much of the previous step to include.

An example of using the new solver is subroutine `invint1`. I chose to implement the model roughening operator $\mathbf{A}$ with the convolution subroutine `tcai1()` on page ??, which has transient end effects (and an output length equal to the input length plus the filter length). The adjoint of subroutine `tcai1()` suggests perturbations in the convolution input (not the filter).

```
module invint {                         # invint -- INVerse INTerpolation in 1-D.
  use lint1
  use tcai1
  use cgstep_mod
  use reg_solver
contains
  subroutine invint1( niter, coord, dd, o1, d1, aa, mm, eps) {
    integer,              intent (in)   :: niter              # iterations
    real,                 intent (in)   :: o1, d1, eps        # axis, scale
    real, dimension (:),  pointer       :: coord, aa          # aa is filter
    real, dimension (:),  intent (in)   :: dd                 # data
    real, dimension (:),  intent (out)  :: mm                 # model
    integer                             :: nreg               # size of A m
    nreg = size( aa) + size( mm)                              # transient
    call lint1_init( o1, d1, coord )                          # interpolation
    call tcai1_init( aa)                                      # filtering
    call solver_reg( lint1_lop, cgstep, tcai1_lop, nreg, mm, dd, niter, eps)
    call cgstep_close( )
    }
}
```

Figure 1.13 shows an example for a $(1, -2, 1)$ filter with $\epsilon = 1$. The continuous curve representing the model $\mathbf{m}$ passes through the data points. Because the models are computed with transient convolution end-effects, the models tend to damp linearly to zero outside the region where signal samples are given.

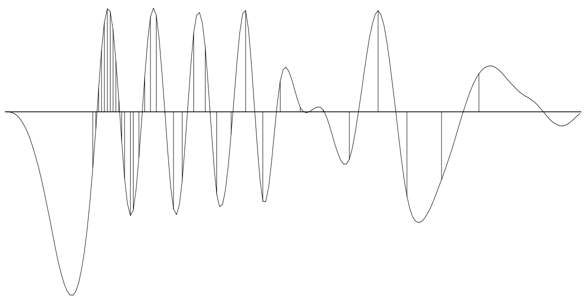

Figure 1.13: Sample points and estimation of a continuous function through them. iin-im1-2+190 [ER]

To show an example where the result is clearly a theoretical answer, I prepared another figure with the simpler filter $(1, -1)$. When we minimize energy in the first derivative of the waveform, the residual distributes itself uniformly between data points so the solution there is a straight line. Theoretically it should be a straight line because a straight line has a vanishing second derivative, and that condition arises by differentiating by $\mathbf{x}'$, the minimized quadratic form $\mathbf{x}'\mathbf{A}'\mathbf{A}\mathbf{x}$, and getting $\mathbf{A}'\mathbf{A}\mathbf{x} = \mathbf{0}$. (By this logic, the curves between data points in Figure 1.13 must be cubics.) The $(1, -1)$ result is shown in Figure 1.14.

The example of Figure 1.14 has been a useful test case for me. You'll see it again in later chapters. What I would like to show you here is a movie showing the convergence to

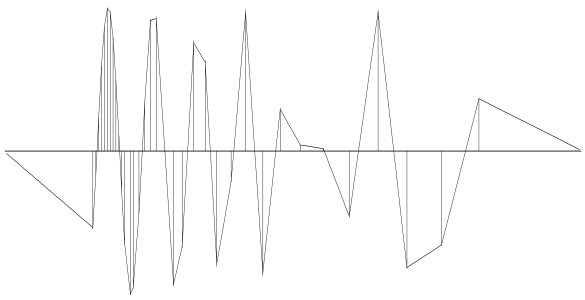

Figure 1.14: The same data samples and a function through them that minimizes the energy in the first derivative. iin-im1-1a90 [ER]

Figure 1.14. Convergence occurs rapidly where data points are close together. The large gaps, however, fill at a rate of one point per iteration.

1.5 PREJUDICE, THE CHOICE OF EPSILON, AND CROSS VALIDATION

First we first look at data $\mathbf{d}$. Then we think about a model $\mathbf{m}$, and an operator $\mathbf{L}$ to link the model and the data. Sometimes the operator is merely the first term in a series expansion about $(\mathbf{m}_0, \mathbf{d}_0)$. Then we fit $\mathbf{d}-\mathbf{d}_0 \approx \mathbf{L}(\mathbf{m}-\mathbf{m}_0)$. To fit the model, we must reduce the fitting residuals. Realizing that the importance of a data residual is not always simply the size of the residual but is generally a function of it, we conjure up (topic for later chapters) a weighting function (which could be a filter) operator $\mathbf{W}$. This defines our data residual:

$$\mathbf{r}_d \quad = \quad \mathbf{W}[\mathbf{L}(\mathbf{m}-\mathbf{m}_0) \ - \ (\mathbf{d}-\mathbf{d}_0)] \tag{1.22}$$

Next we realize that the data might not be adequate to determine the model, perhaps because our comfortable dense sampling of the model ill fits our economical sparse sampling of data. Thus we adopt a fitting goal that mathematicians call "regularization" and we might call a "model style" goal or more simply, a quantification of our prejudice about models. We express this by choosing an operator $\mathbf{A}$, often simply a roughener like a gradient (the choice again a topic in this and later chapters). It defines our model residual by $\mathbf{Am}$ or $\mathbf{A}(\mathbf{m}-\mathbf{m}_0)$, say we choose

$$\mathbf{r}_m \quad = \quad \mathbf{Am} \tag{1.23}$$

In an ideal world, our model prejudice would not conflict with measured data, however, life is not so simple. Since conflicts between data and preconceived notions invariably arise (and they are why we go to the expense of acquiring data) we need an adjustable parameter that measures our "bullheadedness", how much we intend to stick to our preconceived notions in spite of contradicting data. This parameter is generally called epsilon ϵ because we like to imagine that our bullheadedness is small. (In mathematics, ϵ is often taken to be an infinitesimally small quantity.) Although any bullheadedness seems like a bad thing, it must be admitted that measurements are imperfect too. Thus as a practical matter we often find ourselves minimizing

$$\min \quad := \quad \mathbf{r}_d \cdot \mathbf{r}_d \; + \; \epsilon^2 \, \mathbf{r}_m \cdot \mathbf{r}_m \tag{1.24}$$

and wondering what to choose for ϵ. I have two suggestions: My simplest suggestion is to choose ϵ so that the residual of data fitting matches that of model styling. Thus

$$\epsilon \quad = \quad \sqrt{\frac{\mathbf{r}_d \cdot \mathbf{r}_d}{\mathbf{r}_m \cdot \mathbf{r}_m}} \tag{1.25}$$

My second suggestion is to think of the force on our final solution. In physics, force is associated with a gradient. We have a gradient for the data fitting and another for the model styling:

$$\mathbf{g}_d \quad = \quad \mathbf{L}'\mathbf{W}'\mathbf{r}_d \tag{1.26}$$

$$\mathbf{g}_m \quad = \quad \mathbf{A}'\mathbf{r}_m \tag{1.27}$$

We could balance these forces by the choice

$$\epsilon \quad = \quad \sqrt{\frac{\mathbf{g}_d \cdot \mathbf{g}_d}{\mathbf{g}_m \cdot \mathbf{g}_m}} \tag{1.28}$$

Although we often ignore ϵ in discussing the formulation of a problem, when time comes to solve the problem, reality intercedes. Generally, $\mathbf{r}_d$ has different physical units than $\mathbf{r}_m$ (likewise $\mathbf{g}_d$ and $\mathbf{g}_m$) and we cannot allow our solution to depend on the accidental choice of units in which we express the problem. I have had much experience choosing ϵ, but it is only recently that I boiled it down to the above two suggestions. Normally I also try other values, like double or half those of the above choices, and I examine the solutions for subjective appearance. If you find any insightful examples, please tell me about them.

Computationally, we could choose a new ϵ with each iteration, but it is more expeditious to freeze ϵ, solve the problem, recompute ϵ, and solve the problem again. I have never seen a case where more than one iteration was necessary.

People who work with small problems (less than about 10^3 vector components) have access to an attractive theoretical approach called cross-validation. Simply speaking, we could solve the problem many times, each time omitting a different data value. Each solution would provide a model that could be used to predict the omitted data value. The quality of these predictions is a function of ϵ and this provides a guide to finding it. My objections to cross validation are two-fold: First, I don't know how to apply it in the large problems like we solve in this book (I should think more about it); and second, people who worry much about ϵ, perhaps first should think more carefully about their choice of the filters $\mathbf{W}$ and $\mathbf{A}$, which is the focus of this book. Notice that both $\mathbf{W}$ and $\mathbf{A}$ can be defined with a scaling factor which like scaling ϵ. On the other hand, in $\mathbf{W}$ and $\mathbf{A}$ we have a scaling factor that need not be constant but can be a function of space or spatial frequency within the data space and/or model space.

Index