# Chapter 5

# Preconditioning

When I first realized that practical imaging methods in widespread industrial use amounted merely to the adjoint of forward modeling, I (and others) thought an easy way to achieve fame and fortune would be to introduce the first steps towards inversion along the lines of Chapter 2. Although inversion generally requires a prohibitive number of steps, I felt that moving in the gradient direction, the direction of steepest descent, would move us rapidly in the direction of practical improvements. This turned out to be optimistic. It was too slow. But then I learned about the conjugate gradient method that spectacularly overcomes a well-known speed problem with the method of steepest descents. I came to realize that it was still too slow. I learned this by watching the convergence in Figure 1.6. This led me to the helix method in Chapter 4. Here we'll see how it speeds many applications.

We'll also come to understand why the gradient is such a poor direction both for steepest descent and for conjugate gradients. An indication of our path is found in the contrast between and exact solution $\mathbf{m} = (\mathbf{A}'\mathbf{A})^{-1}\mathbf{A}'\mathbf{d}$ and the gradient $\Delta\mathbf{m} = \mathbf{A}'\mathbf{d}$ (which is the first step starting from $\mathbf{m} = \mathbf{0}$). Notice that $\Delta\mathbf{m}$ differs from $\mathbf{m}$ by the factor $(\mathbf{A}'\mathbf{A})^{-1}$. This factor is sometimes called a spectrum and in some situations it literally is a frequency spectrum. In these cases, $\Delta\mathbf{m}$ simply gets a different spectrum from $\mathbf{m}$ and many iterations are required to fix it. Here we'll find that for many problems, "preconditioning" with the helix is a better way.

## 5.1 PRECONDITIONED DATA FITTING

Iterative methods (like conjugate-directions) can sometimes be accelerated by a change of variables. The simplest change of variable is called a "trial solution". Formally, we write the solution as

$$\mathbf{m} \quad = \quad \mathbf{S}\mathbf{p} \tag{5.1}$$

where $\mathbf{m}$ is the map we seek, columns of the matrix $\mathbf{S}$ are "shapes" that we like, and coefficients in $\mathbf{p}$ are unknown coefficients to select amounts of the favored shapes. The variables $\mathbf{p}$ are often called the "preconditioned variables". It is not necessary that $\mathbf{S}$ be an invertible matrix,

but we'll see later that invertibility is helpful. Take this trial solution and insert it into a typical fitting goal

$$\mathbf{0} \quad \approx \quad \mathbf{Fm} - \mathbf{d} \tag{5.2}$$

and get

$$\mathbf{0} \quad \approx \quad \mathbf{FSp} - \mathbf{d} \tag{5.3}$$

We pass the operator **FS** to our iterative solver. After finding the best fitting **p**, we merely evaluate $\mathbf{m} = \mathbf{Sp}$ to get the solution to the original problem.

We hope this change of variables has saved effort. For each iteration, there is a little more work: Instead of the iterative application of **F** and $\mathbf{F}'$ we have iterative application of **FS** and $\mathbf{S}'\mathbf{F}'$. Our hope is that the number of iterations decreases because we are clever, or because we have been lucky in our choice of **S**. Hopefully, the extra work of the preconditioner operator **S** is not large compared to **F**. If we should be so lucky that $\mathbf{S} = \mathbf{F}^{-1}$, then we get the solution immediately. Obviously we would try any guess with $\mathbf{S} \approx \mathbf{F}^{-1}$. Where I have known such **S** matrices, I have often found that convergence is accelerated, but not by much. Sometimes it is worth using **FS** for a while in the beginning, but later it is cheaper and faster to use only **F**. A practitioner might regard the guess of **S** as prior information, like the guess of the initial model $\mathbf{m}_0$.

For a square matrix **S**, the use of a preconditioner should not change the ultimate solution. Taking **S** to be a wide rectangular matrix, reduces the number of adjustable parameters, changes the solution, gets it quicker, but lower resolution.

### 5.1.1 Preconditioner with a starting guess

In many applications, for many reasons, we have a starting guess $\mathbf{m}_0$ of the solution. You might worry that you could not find the starting preconditioned variable $\mathbf{p}_0 = \mathbf{S}^{-1}\mathbf{m}_0$ because you did not know the inverse of **S**. The way to avoid this problem is to reformulate the problem in terms of a new variable $\tilde{\mathbf{m}}$ where $\mathbf{m} = \tilde{\mathbf{m}} + \mathbf{m}_0$. Then $\mathbf{0} \approx \mathbf{Fm} - \mathbf{d}$ becomes $\mathbf{0} \approx \mathbf{F}\tilde{\mathbf{m}} - (\mathbf{d} - \mathbf{Fm}_0)$ or $\mathbf{0} \approx \mathbf{F}\tilde{\mathbf{m}} - \tilde{\mathbf{d}}$. Thus we have accomplished the goal of taking a problem with a nonzero starting model and converting it a problem of the same type with a zero starting model. Thus we do not need the inverse of **S** because the iteration starts from $\tilde{\mathbf{m}} = \mathbf{0}$ so $\mathbf{p}_0 = \mathbf{0}$.

## 5.2 PRECONDITIONING THE REGULARIZATION

The basic formulation of a geophysical estimation problem consists of setting up *two* goals, one for data fitting, and the other for model shaping. With two goals, preconditioning is somewhat different. The two goals may be written as:

$$\mathbf{0} \quad \approx \quad \mathbf{Fm} - \mathbf{d} \tag{5.4}$$

$$\mathbf{0} \quad \approx \quad \mathbf{Am} \tag{5.5}$$

which defines two residuals, a so-called "data residual" and a "model residual" that are usually minimized by conjugate-gradient, least-squares methods.

To fix ideas, let us examine a toy example. The data and the first three rows of the matrix below are random numbers truncated to integers. The model roughening operator **A** is a first differencing operator times 100.

```
d(m)     F(m,n)                                                 iter  Norm
---     ------------------------------------------------------  ---- -----------
 41.    -55. -90. -24. -13. -73.  61. -27. -19.  23. -55.     1 20.00396538
 33.      8. -86.  72.  87. -41.  -3. -29.  29. -66.  50.     2 12.14780140
-58.     84. -49.  80.  44. -52. -51.   8.  86.  77.  50.     3  8.94393635
  0.    100.   0.   0.   0.   0.   0.   0.   0.   0.   0.     4  6.04517126
  0.   -100. 100.   0.   0.   0.   0.   0.   0.   0.   0.     5  2.64737511
  0.      0.-100. 100.   0.   0.   0.   0.   0.   0.   0.     6  0.79238468
  0.      0.   0.-100. 100.   0.   0.   0.   0.   0.   0.     7  0.46083349
  0.      0.   0.   0.-100. 100.   0.   0.   0.   0.   0.     8  0.08301232
  0.      0.   0.   0.   0.-100. 100.   0.   0.   0.   0.     9  0.00542009
  0.      0.   0.   0.   0.   0.-100. 100.   0.   0.   0.    10  0.00000565
  0.      0.   0.   0.   0.   0.   0.-100. 100.   0.   0.    11  0.00000026
  0.      0.   0.   0.   0.   0.   0.   0.-100. 100.   0.    12  0.00000012
  0.      0.   0.   0.   0.   0.   0.   0.   0.-100. 100.    13  0.00000000
```

Notice at the tenth iteration, the residual suddenly plunges 4 significant digits. Since there are ten unknowns and the matrix is obviously full-rank, conjugate-gradient theory tells us to expect the exact solution at the tenth iteration. This is the first miracle of conjugate gradients. (The residual actually does not drop to zero. What is printed in the `Norm` column is the square root of the sum of the squares of the residual components at the `iter`-th iteration minus that at the last interation.)

### 5.2.1 The second miracle of conjugate gradients

The second miracle of conjugate gradients is exhibited below. The data and data fitting matrix are the same, but the model damping is simplified.

```
d(m)    F(m,n)                                                 iter  Norm
---    ------------------------------------------------------  ----  ----------
 41.   -55. -90. -24. -13. -73.  61. -27. -19.  23. -55.     1  3.64410686
 33.     8. -86.  72.  87. -41.  -3. -29.  29. -66.  50.     2  0.31269890
-58.    84. -49.  80.  44. -52. -51.   8.  86.  77.  50.     3 -0.00000021
  0.   100.   0.   0.   0.   0.   0.   0.   0.   0.   0.     4 -0.00000066
  0.     0. 100.   0.   0.   0.   0.   0.   0.   0.   0.     5 -0.00000080
  0.     0.   0. 100.   0.   0.   0.   0.   0.   0.   0.     6 -0.00000065
  0.     0.   0.   0. 100.   0.   0.   0.   0.   0.   0.     7 -0.00000088
  0.     0.   0.   0.   0. 100.   0.   0.   0.   0.   0.     8 -0.00000074
  0.     0.   0.   0.   0.   0. 100.   0.   0.   0.   0.     9 -0.00000035
  0.     0.   0.   0.   0.   0.   0. 100.   0.   0.   0.    10 -0.00000037
  0.     0.   0.   0.   0.   0.   0.   0. 100.   0.   0.    11 -0.00000018
  0.     0.   0.   0.   0.   0.   0.   0.   0. 100.   0.    12  0.00000000
  0.     0.   0.   0.   0.   0.   0.   0.   0.   0. 100.    13  0.00000000
```

Even though the matrix is full-rank, we see the residual drop about 6 decimal places after the third iteration! This convergence behavior is well known in the computational mathematics literature. Despite its practical importance, it doesn't seem to have a name or identified discoverer. So I call it the "second miracle."

Practitioners usually don't like the identity operator for model-shaping. Generally they prefer to penalize wiggliness. For practitioners, the lesson of the second miracle of conjugate gradients is that we have a choice of many iterations, or learning to transform independent variables so that the regularization operator becomes an identity matrix. Basically, such a transformation reduces the iteration count from something about the size of the model space to something about the size of the data space. Such a transformation is called preconditioning. In practice, data is often accumulated in bins. Then the iteration count is reduced (in principle) to the count of full bins and should be independent of the count of the empty bins. This allows refining the bins, enhancing the resolution.

More generally, the model goal $\mathbf{0} \approx \mathbf{Am}$ introduces a roughening operator like a gradient, Laplacian (and in chapter 6 a Prediction-Error Filter (PEF)). Thus the model goal is usually a filter, unlike the data-fitting goal which involves all manner of geometry and physics. When the model goal is a filter its inverse is also a filter. Of course this includes multidimensional filters with a helix.

The preconditioning transformation $\mathbf{m} = \mathbf{Sp}$ gives us

$$\begin{aligned} \mathbf{0} &\approx \mathbf{FSp} - \mathbf{d} \\ \mathbf{0} &\approx \mathbf{ASp} \end{aligned} \tag{5.6}$$

The operator $\mathbf{A}$ is a roughener while $\mathbf{S}$ is a smoother. The choices of both $\mathbf{A}$ and $\mathbf{S}$ are somewhat subjective. This suggests that we eliminate $\mathbf{A}$ altogether by *defining* it to be proportional to the inverse of $\mathbf{S}$, thus $\mathbf{AS} = \mathbf{I}$. The fitting goals become

$$\begin{aligned} \mathbf{0} &\approx \mathbf{FSp} - \mathbf{d} \\ \mathbf{0} &\approx \epsilon\, \mathbf{p} \end{aligned} \tag{5.7}$$

which enables us to benefit from the "second miracle". After finding $\mathbf{p}$, we obtain the final model with $\mathbf{m} = \mathbf{Sp}$.

### 5.2.2 Importance of scaling

Another simple toy example shows us the importance of scaling. We use the same example as above except that the $i$-th column is multiplied by $i/10$ which means the $i$-th model variable has been divided by $i/10$.

```
d(m)    F(m,n)                                              iter  Norm
---    ------------------------------------------------    ---- -----------
 41.    -6. -18.  -7.  -5. -36.  37. -19. -15.  21. -55.     1 11.59544849
 33.     1. -17.  22.  35. -20.  -2. -20.  23. -59.  50.     2  6.97337770
-58.     8. -10.  24.  18. -26. -31.   6.  69.  69.  50.     3  5.64414406
```

```
 0.    10.   0.   0.   0.   0.   0.   0.   0.   0.   0.     4  4.32118177
 0.     0.  20.   0.   0.   0.   0.   0.   0.   0.   0.     5  2.64755201
 0.     0.   0.  30.   0.   0.   0.   0.   0.   0.   0.     6  2.01631355
 0.     0.   0.   0.  40.   0.   0.   0.   0.   0.   0.     7  1.23219979
 0.     0.   0.   0.   0.  50.   0.   0.   0.   0.   0.     8  0.36649203
 0.     0.   0.   0.   0.   0.  60.   0.   0.   0.   0.     9  0.28528941
 0.     0.   0.   0.   0.   0.   0.  70.   0.   0.   0.    10  0.06712411
 0.     0.   0.   0.   0.   0.   0.   0.  80.   0.   0.    11  0.00374284
 0.     0.   0.   0.   0.   0.   0.   0.   0.  90.   0.    12 -0.00000040
 0.     0.   0.   0.   0.   0.   0.   0.   0.   0. 100.    13  0.00000000
```

We observe that solving the same problem for the scaled variables has required a severe increase in the number of iterations required to get the solution. We lost the benefit of the second CG miracle. Even the rapid convergence predicted for the 10-th iteration is delayed until the 12-th.

### 5.2.3 Statistical interpretation

This book is not a statistics book. Never-the-less, many of you have some statistical knowledge that allows you a statistical interpretation of these views of preconditioning.

A statistical concept is that we can combine many streams of random numbers into a composite model. Each stream of random numbers is generally taken to be uncorrelated with the others, to have zero mean, and to have the same variance as all the others. This is often abbreviated as IID, denoting Independent, Identically Distributed. Linear combinations like filtering and weighting operations of these IID random streams can build correlated random functions much like those observed in geophysics. A geophysical practitioner seeks to do the inverse, to operate on the correlated unequal random variables and create the statistical ideal random streams. The identity matrix required for the "second miracle", and our search for a good preconditioning transformation are related ideas. The relationship will become more clear in chapter 6 when we learn how to estimate the best roughening operator $\mathbf{A}$ as a prediction-error filter.

Two philosophies to find a preconditioner:

1. Dream up a smoothing operator $\mathbf{S}$.
2. Estimate a prediction-error filter $\mathbf{A}$, and then use its inverse $\mathbf{S} = \mathbf{A}^{-1}$.

Deconvolution on a helix is an all-purpose preconditioning strategy for multidimensional model regularization.

The outstanding acceleration of convergence by preconditioning suggests that the philosophy of image creation by optimization has a dual orthonormality: First, Gauss (and common sense) tells us that the data residuals should be roughly equal in size. Likewise in Fourier

space they should be roughly equal in size, which means they should be roughly white, i.e. orthonormal. (I use the word "orthonormal" because white means the autocorrelation is an impulse, which means the signal is statistically orthogonal to shifted versions of itself.) Second, to speed convergence of iterative methods, we need a whiteness, another orthonormality, in the solution. The map image, the physical function that we seek, might not be itself white, so we should solve first for another variable, the whitened map image, and as a final step, transform it to the "natural colored" map.

### 5.2.4 The preconditioned solver

Summing up the ideas above, we start from fitting goals

$$\begin{aligned} \mathbf{0} &\approx \mathbf{Fm} - \mathbf{d} \\ \mathbf{0} &\approx \mathbf{Am} \end{aligned} \tag{5.8}$$

and we change variables from $\mathbf{m}$ to $\mathbf{p}$ using $\mathbf{m} = \mathbf{A}^{-1}\mathbf{p}$

$$\begin{aligned} \mathbf{0} &\approx \mathbf{Fm} - \mathbf{d} &&= \mathbf{FA}^{-1}\ \mathbf{p} - \mathbf{d} \\ \mathbf{0} &\approx \mathbf{Am} &&= \mathbf{I}\ \ \mathbf{p} \end{aligned} \tag{5.9}$$

Preconditioning means iteratively fitting by adjusting the $\mathbf{p}$ variables and then finding the model by using $\mathbf{m} = \mathbf{A}^{-1}\mathbf{p}$. A new reusable preconditioned solver is the module `solver_prc` on page 6. Likewise the modeling operator $\mathbf{F}$ is called `Fop` and the smoothing operator $\mathbf{A}^{-1}$ is called `Sop`. Details of the code are only slightly different from the regularized solver `solver_reg` on page 87.

```
module solver_prc_mod{                          # 0 = W (F S J p - d)
  use chain0_mod + solver_report_mod            # 0 =      I   p
  logical, parameter, private  :: T = .true., F = .false.
contains
  subroutine solver_prc( m,d, Fop, Sop, stepper, nSop, niter,eps &
  ,             Wop,Jop,p0,rm0,err,resd,resm,mmov,rmov,verb) {
    optional :: Wop,Jop,p0,rm0,err,resd,resm,mmov,rmov,verb
    interface { #------------------------- begin definitions -----------
       integer function Fop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Sop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Wop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function Jop(adj,add,m,d){real::m(:),d(:);logical,intent(in)::adj,add}
       integer function stepper(forget,m,g,rr,gg) {
         real, dimension(:)  ::        m,g,rr,gg
         logical             :: forget            }
    }
    real, dimension(:),    intent(in)          :: d, p0,rm0
    integer,               intent(in)          :: niter, nSop
    logical,               intent(in)          :: verb
    real,                  intent(in)          :: eps
    real, dimension(:),    intent(out)         :: m,err, resd,resm
    real, dimension(:,:),  intent(out)         ::        rmov,mmov
    real, dimension(size( m))                  :: p ,  g
    real, dimension(size( d) + nSop), target   :: rr, gg, tt
```

```
    real, dimension(:), pointer                :: rd, gd, td
    real, dimension(:), pointer                :: rm, gm, tm
    integer                                    :: iter, stat
    logical                                    :: forget
    rd => rr(1:size(d)); rm => rr(1+size(d):)
    gd => gg(1:size(d)); gm => gg(1+size(d):)
    td => tt(1:size(d)); tm => tt(1+size(d):)
    if(present( Wop)) stat=Wop(F,F,-d,rd) # begin initialization --------
    else rd = -d                                                  #Rd  = -W    d
    rm = 0.; if(present(rm0)) rm=rm0                              #Rm  =      Rm0
    if(present( p0)){ p=p0                                        # p  =       p0
       if(present( Wop)) call chain0(Wop,Fop,Sop,F,T,p,rd,tm,td)
       else              call chain0(    Fop,Sop,F,T,p,rd,tm   )#Rd +=  WFS  p0
       rm = rm + eps*p                                            #Rm +=   e I p0
    } else p=0
    forget = T; #-------------------------- begin iterations ------------
    do iter = 1,niter {
       if(present(Wop)) call chain0(Wop,Fop,Sop,T,F,g,rd,tm,td)
       else             call chain0(    Fop,Sop,T,F,g,rd,tm   ) #g  = (WFS)'Rd
       g = g + eps*rm                                             #g +=   e I'Rm
       if(present(Jop)){ tm=g;         stat=Jop(F,F,tm,g      )}#g  =     J g
       if(present(Wop)) call chain0(Wop,Fop,Sop,F,F,g,gd,tm,td)
       else             call chain0(    Fop,Sop,F,F,g,gd,tm   ) #Gd = (WFS) g
       gm   =  eps*g                                              #Gm =   e I g
       stat = stepper(forget, p,g, rr,gg)                         #m+=dm; R+=dR
       if(stat ==1) exit # got stuck descending
       stat = Sop(F,F,p,m)                                        #m  = S p
       if(present( mmov)) mmov(:,iter) =  m(:size(mmov,1)) # report -----
       if(present( rmov)) rmov(:,iter) = rr(:size(rmov,1))
       if(present( err ))  err(  iter) = dot_product(rd,rd)
       if(present( verb)){ if(verb) call solver_report(iter,m,g,rd,rm)}
       forget=F
    }
    if(present( resd)) resd = rd
    if(present( resm)) resm = rm(:size(resm))
  }
}
```

## 5.3 OPPORTUNITIES FOR SMART DIRECTIONS

Recall the fitting goals (1.10)

$$\begin{array}{lllllll} \mathbf{0} & \approx & \mathbf{r}_d & = & \mathbf{Fm} - \mathbf{d} & = & \mathbf{FA}^{-1} \quad \mathbf{p} - \mathbf{d} \\ \mathbf{0} & \approx & \mathbf{r}_m & = & \mathbf{Am} & = & \mathbf{I} \quad\quad \mathbf{p} \end{array} \tag{5.10}$$

Without preconditioning we have the search direction

$$\Delta \mathbf{m}_{\text{bad}} \quad = \quad \left[ \begin{array}{cc} \mathbf{F}' & \mathbf{A}' \end{array} \right] \left[ \begin{array}{c} \mathbf{r}_d \\ \mathbf{r}_m \end{array} \right] \tag{5.11}$$

and with preconditioning we have the search direction

$$\Delta \mathbf{p}_{\text{good}} \quad = \quad \left[ \begin{array}{cc} (\mathbf{FA}^{-1})' & \mathbf{I} \end{array} \right] \left[ \begin{array}{c} \mathbf{r}_d \\ \mathbf{r}_m \end{array} \right] \tag{5.12}$$

The essential feature of preconditioning is not that we perform the iterative optimization in terms of the variable $\mathbf{p}$. The essential feature is that we use a search direction that is a gradient with respect to $\mathbf{p}'$ not $\mathbf{m}'$. Using $\mathbf{Am} = \mathbf{p}$ we have $\mathbf{A}\Delta\mathbf{m} = \Delta\mathbf{p}$. This enables us to define a good search direction in model space.

$$\Delta\mathbf{m}_{\text{good}} \quad = \quad \mathbf{A}^{-1}\Delta\mathbf{p}_{\text{good}} \quad = \quad \mathbf{A}^{-1}(\mathbf{A}^{-1})'\mathbf{F}'\mathbf{r}_d + \mathbf{A}^{-1}\mathbf{r}_m \tag{5.13}$$

Define the gradient by $\mathbf{g} = \mathbf{F}'\mathbf{r}_d$ and notice that $\mathbf{r}_m = \mathbf{p}$.

$$\Delta\mathbf{m}_{\text{good}} \quad = \quad \mathbf{A}^{-1}(\mathbf{A}^{-1})'\,\mathbf{g} + \mathbf{m} \tag{5.14}$$

The search direction (1.14) shows a positive-definite operator scaling the gradient. Each component of any gradient vector is independent of each other. All independently point a direction for descent. Obviously, each can be scaled by any positive number. Now we have found that we can also scale a gradient vector by a positive definite matrix and we can still expect the conjugate-direction algorithm to descend, as always, to the "exact" answer in a finite number of steps. This is because modifying the search direction with $\mathbf{A}^{-1}(\mathbf{A}^{-1})'$ is equivalent to solving a conjugate-gradient problem in $\mathbf{p}$.

## 5.4 NULL SPACE AND INTERVAL VELOCITY

A bread-and-butter problem in seismology is building the velocity as a function of depth (or vertical travel time) starting from certain measurements. The measurements are described elsewhere (BEI for example). They amount to measuring the integral of the velocity squared from the surface down to the reflector. It is known as the RMS (root-mean-square) velocity. Although good quality echos may arrive often, they rarely arrive continuously for all depths. Good information is interspersed unpredictably with poor information. Luckily we can also estimate the data quality by the "coherency" or the "stack energy". In summary, what we get from observations and preprocessing are two functions of travel-time depth, (1) the integrated (from the surface) squared velocity, and (2) a measure of the quality of the integrated velocity measurement. Some definitions:

**d** is a data vector whose components range over the vertical traveltime depth $\tau$, and whose component values contain the scaled RMS velocity squared $\tau v^2_{\text{RMS}}/\Delta\tau$ where $\tau/\Delta\tau$ is the index on the time axis.

**W** is a diagonal matrix along which we lay the given measure of data quality. We will use it as a weighting function.

**C** is the matrix of causal integration, a lower triangular matrix of ones.

**D** is the matrix of causal differentiation, namely, $\mathbf{D} = \mathbf{C}^{-1}$.

**u** is a vector whose components range over the vertical traveltime depth $\tau$, and whose component values contain the interval velocity squared $v^2_{\text{interval}}$.

From these definitions, under the assumption of a stratified earth with horizontal reflectors (and no multiple reflections) the theoretical (squared) interval velocities enable us to define the theoretical (squared) RMS velocities by

$$\mathbf{Cu} \quad = \quad \mathbf{d} \tag{5.15}$$

With imperfect data, our data fitting goal is to minimize the residual

$$\mathbf{0} \quad \approx \quad \mathbf{W}[\mathbf{Cu}-\mathbf{d}] \tag{5.16}$$

To find the interval velocity where there is no data (where the stack power theoretically vanishes) we have the "model damping" goal to minimize the wiggliness $\mathbf{p}$ of the squared interval velocity $\mathbf{u}$.

$$\mathbf{0} \quad \approx \quad \mathbf{Du} \quad = \quad \mathbf{p} \tag{5.17}$$

We precondition these two goals by changing the optimization variable from interval velocity squared $\mathbf{u}$ to its wiggliness $\mathbf{p}$. Substituting $\mathbf{u} = \mathbf{Cp}$ gives the two goals expressed as a function of wiggliness $\mathbf{p}$.

$$\mathbf{0} \quad \approx \quad \mathbf{W}\left[\mathbf{C}^2\mathbf{p}-\mathbf{d}\right] \tag{5.18}$$

$$\mathbf{0} \quad \approx \quad \epsilon\,\mathbf{p} \tag{5.19}$$

### 5.4.1 Balancing good data with bad

Choosing the size of $\epsilon$ chooses the stiffness of the curve that connects regions of good data. Our first test cases gave solutions that we interpreted to be too stiff at early times and too flexible at later times. This leads to two possible ways to deal with the problem. One way modifies the model shaping and the other modifies the data fitting. The program below weakens the data fitting weight with time. This has the same effect as stiffening the model shaping with time.

```
module vrms2int_mod {               # Transform from RMS to interval velocity
  use causint
  use weight
  use mask1
  use cgstep_mod
  use solver_prc_mod
contains
  subroutine vrms2int( niter, eps, weight, vrms, vint) {
    integer,            intent( in)     :: niter     # iterations
    real,               intent( in)     :: eps       # scaling parameter
    real, dimension(:), intent( in out) :: vrms      # RMS velocity
    real, dimension(:), intent( out)    :: vint      # interval velocity
    real, dimension(:), pointer         :: weight    # data weighting
    integer                             :: st,it,nt
    logical, dimension( size( vint))    :: mask
```

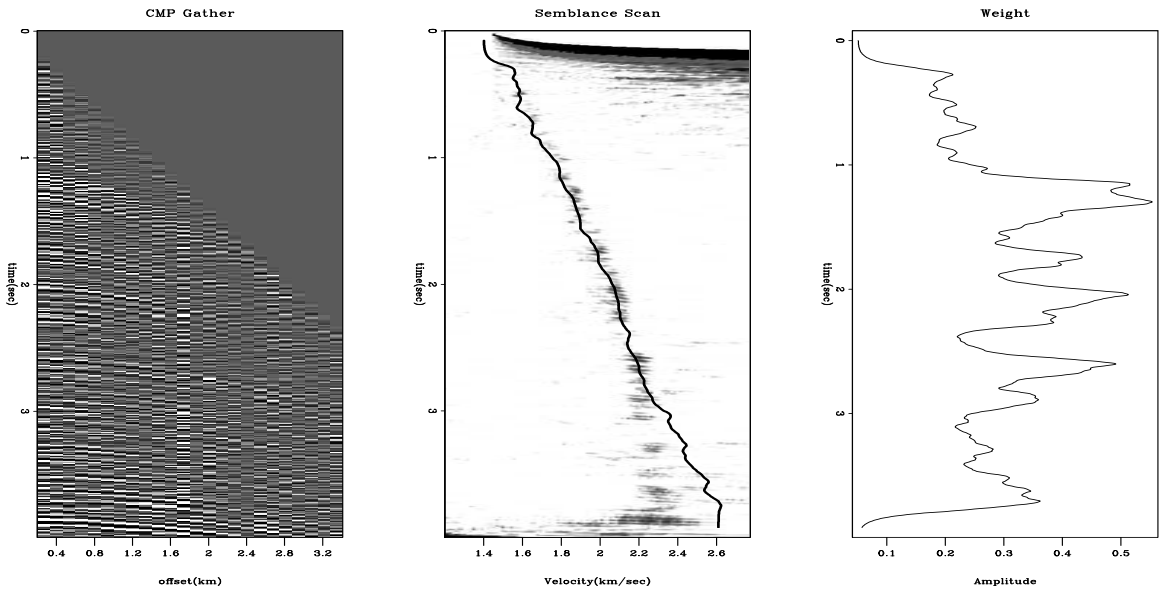


Figure 5.1: Raw CMP gather (left), Semblance scan (middle), and semblance value used for weighting function (right). (Clapp) prc-clapp [ER,M]

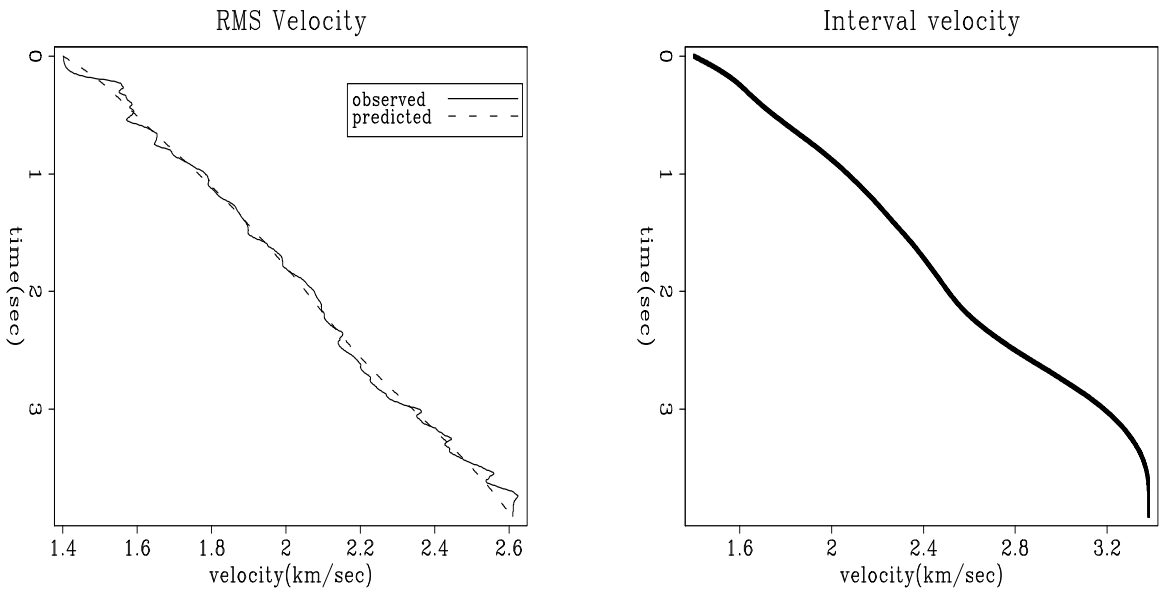


Figure 5.2: Observed RMS velocity and that predicted by a stiff model with $\epsilon = 4$. (Clapp) prc-stiff [ER]

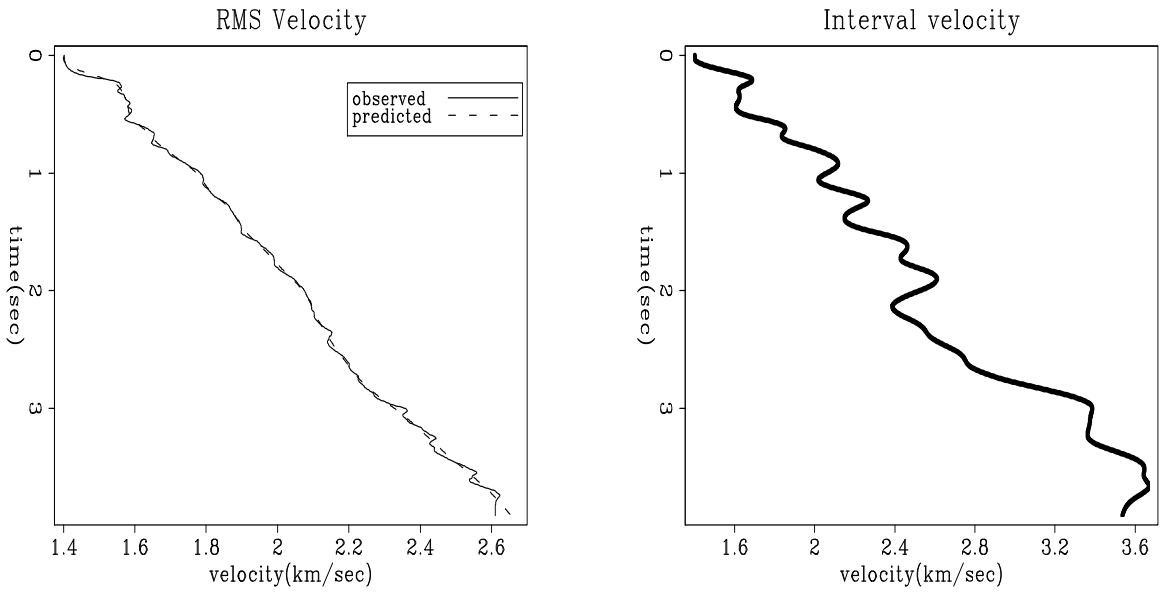


Figure 5.3: Observed RMS velocity and that predicted by a flexible model with $\epsilon = .25$ (Clapp) prc-flex [ER]

```
    logical, dimension(:), pointer      :: msk
    real,    dimension( size( vrms))    :: dat ,wt
    real, dimension(:), pointer         :: wght
    nt = size( vrms)
    do it= 1, nt {
       dat( it) = vrms( it) * vrms( it) * it
       wt( it) = weight( it)*(1./it)           # decrease weight with time
       }
    mask = .false.;   mask( 1) = .true.            # constrain first point
    vint = 0.     ;   vint( 1) = dat( 1)
    allocate(wght(size(wt)))
    wght=wt
    call weight_init(wght)
    allocate(msk(size(mask)))
    msk=.not.mask
    call mask1_init(msk)
    call solver_prc( m=vint, d=dat,Fop=causint_lop, stepper=cgstep, niter=niter, &
        Sop= causint_lop, nSop=nt, eps = eps,verb=.true.,Jop=mask1_lop, &
        p0=vint, Wop=weight_lop)
    call cgstep_close()
    st = causint_lop( .false., .false., vint, dat)
    do it= 1, nt
       vrms( it) = sqrt( dat( it)/it)
    vint = sqrt( vint)
    }
}
```

### 5.4.2 Lateral variations

The analysis above appears one dimensional in depth. Conventional interval velocity estimation builds a velocity-depth model independently at each lateral location. Here we have a logical path for combining measurements from various lateral locations. We can change the regularization to something like $\mathbf{0} \approx \nabla \mathbf{u}$. Instead of merely minimizing the vertical gradient of velocity we minimize its spatial gradient. Luckily we have preconditioning and the helix to speed the solution.

Likewise the construction of the data quality screen $\mathbf{G}$ would naturally involve the full three-dimensional setting.

### 5.4.3 Blocky models

Sometimes we seek a velocity model that increases smoothly with depth through our scattered measurements of good-quality RMS velocities. Other times, we seek a blocky model. (Where seismic data is poor, a well log could tell us whether to choose smooth or blocky.) Here we see an estimation method that can choose the blocky alternative, or some combination of smooth and blocky.

Consider the five layer model in Figure 1.4. Each layer has unit traveltime thickness (so integration is simply summation). Let the squared interval velocities be $(a,b,c,d,e)$ with strong reliable reflections at the base of layer $c$ and layer $e$, and weak, incoherent, "bad" reflections at bases of $(a,b,d)$. Thus we measure $V_c^2$ the RMS velocity squared of the top three layers and $V_e^2$ that for all five layers. Since we have no reflection from at the base of the fourth layer, the velocity in the fourth layer is not measured but a matter for choice. In a smooth linear fit we would want $d = (c+e)/2$. In a blocky fit we would want $d = e$.

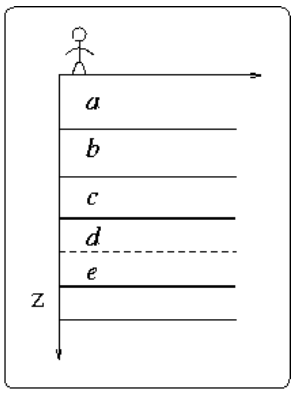


Figure 5.4: A layered earth model. The layer interfaces cause reflections. Each layer has a constant velocity in its interior. prc-rosales [NR]

Our screen for good reflections looks like $(0,0,1,0,1)$ and our screen for bad ones looks like the complement $(1,1,0,1,0)$. We put these screens on the diagonals of diagonal matrices

**G** and **B**. Our fitting goals are:

$$
\begin{aligned}
3V_c^2 &\approx a+b+c && (5.20)\\
5V_e^2 &\approx a+b+c+d+e && (5.21)\\
u_0 &\approx a && (5.22)\\
0 &\approx -a+b && (5.23)\\
0 &\approx -b+c && (5.24)\\
0 &\approx -c+d && (5.25)\\
0 &\approx -d+e && (5.26)
\end{aligned}
$$

For the blocky solution, we do not want the fitting goal (1.25). Further explanations await completion of examples.

## 5.5 INVERSE LINEAR INTERPOLATION

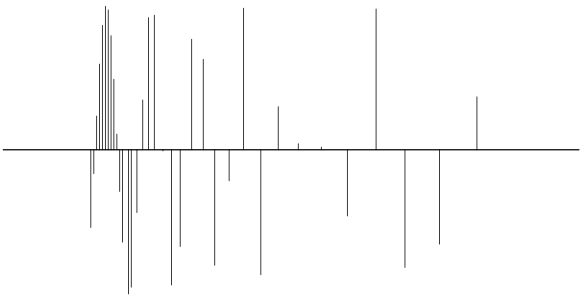

Figure 5.5: The input data are irregularly sampled. prc-data [ER]

The first example is a simple synthetic test for 1-D inverse interpolation. The input data were randomly subsampled (with decreasing density) from a sinusoid (Figure 1.5). The forward operator **L** in this case is linear interpolation. We seek a regularly sampled model that could predict the data with a forward linear interpolation. Sparse irregular distribution of the input data makes the regularization enforcement a necessity. I applied convolution with the simple $(1,-1)$ difference filter as the operator **D** that forces model continuity (the first-order spline). An appropriate preconditioner **S** in this case is recursive causal integration. As expected, preconditioning provides a much faster rate of convergence. Since iteration to the exact solution is never achieved in large-scale problems, the results of iterative optimization may turn out quite differently. Bill Harlan points out that the two goals in (1.8) conflict with each other: the first one enforces "details" in the model, while the second one tries to smooth them out. Typically, regularized optimization creates a complicated model at early iterations. At first, the data fitting goal (1.8) plays a more important role. Later, the regularization goal (1.8) comes into play and simplifies (smooths) the model as much as needed. Preconditioning acts differently. The very first iterations create a simplified (smooth) model. Later, the data fitting goal adds more details into the model. If we stop the iterative process early, we end up with an insufficiently complex model, not in an insufficiently simplified one. Figure 1.6 provides a clear illustration of Harlan's observation.

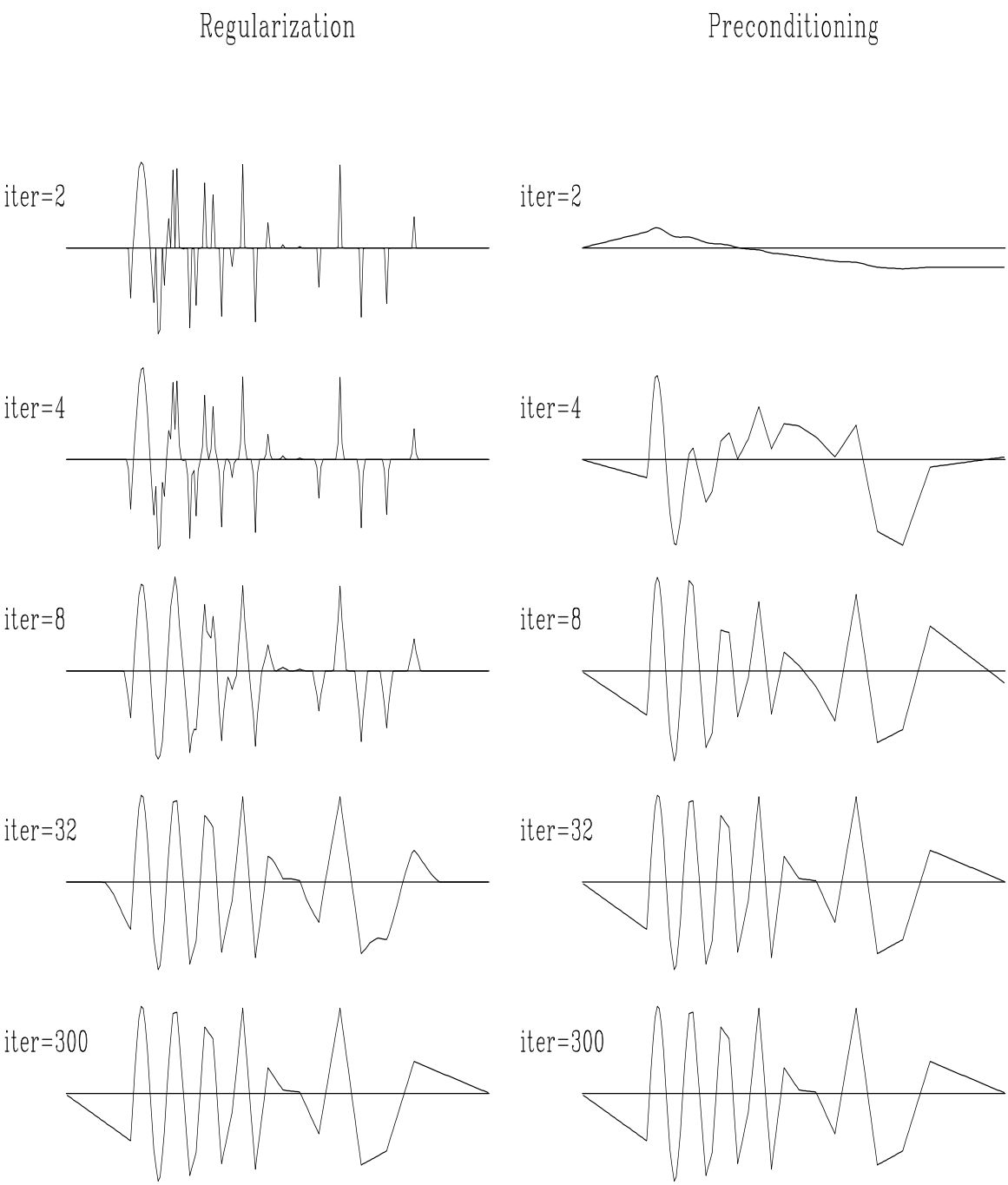


Figure 5.6: Convergence history of inverse linear interpolation. Left: regularization, right: preconditioning. The regularization operator **A** is the derivative operator (convolution with $(1, -1)$. The preconditioning operator **S** is causal integration. prc-conv1 [ER,M]

Figure 1.7 measures the rate of convergence by the model residual, which is a distance from the current model to the final solution. It shows that preconditioning saves many iterations. Since the cost of each iteration for each method is roughly equal, the efficiency of preconditioning is evident.

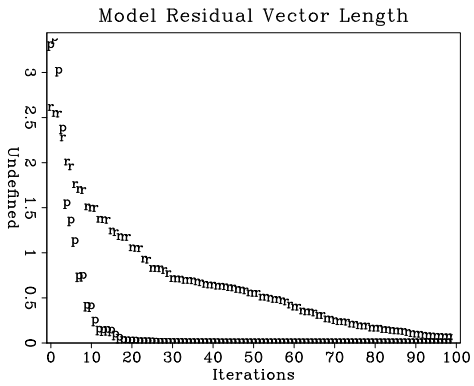


Figure 5.7: Convergence of the iterative optimization, measured in terms of the model residual. The "p" points stand for preconditioning; the "r" points, regularization. prc-schwab1 [ER]

The module `invint2` on page 15 invokes the solvers to make Figures 1.6 and 1.7. We use convolution with `helicon` on page 100 for the regularization and we use deconvolution with `polydiv` on page 101 for the preconditioning. The code looks fairly straightforward except for the oxymoron `known=aa%mis`.

```
module invint2 {   # Inverse linear interpolation
  use lint1
  use helicon                           # regularized    by helix   filtering
  use polydiv                           # preconditioned by inverse filtering
  use cgstep_mod
  use solver_reg_mod
  use solver_prc_mod
contains
  subroutine invint( niter, coord,ord, o1,d1, mm,mmov, eps, aa, method) {
    logical,                intent( in)  :: method
    integer,                intent( in)  :: niter
    real,                   intent( in)  :: o1, d1, eps
    real,    dimension( :), intent( in)  :: ord
    type( filter),          intent( in)  :: aa
    real,    dimension( :), intent( out) :: mm
    real,    dimension( :,:), intent( out) :: mmov         # model movie
    real,    dimension( :), pointer      :: coord        # coordinate
    call lint1_init( o1, d1, coord)
    if( method) {                                        # preconditioning
       call polydiv_init( size(mm), aa)
       call solver_prc( Fop=lint1_lop, stepper=cgstep, niter=niter, m=mm, d=ord,
                        Sop=polydiv_lop, nSop=size(mm), eps=eps, mmov=mmov, verb=.true.)
       call polydiv_close()
    } else {                                             # regularization
       call helicon_init( aa)
       call solver_reg( Fop=lint1_lop, stepper=cgstep, niter=niter, m=mm, d=ord,
                        Aop=helicon_lop, nAop=size(mm), eps=eps, mmov=mmov, verb=.true.)
    }
    call cgstep_close()
  }
}
```

## 5.6 EMPTY BINS AND PRECONDITIONING

There are at least three ways to fill empty bins. Two require a roughening operator $\mathbf{A}$ while the third requires a smoothing operator which (for comparison purposes) we denote $\mathbf{A}^{-1}$. The three methods are generally equivalent though they differ in important details.

The original way in Chapter 3 is to restore missing data by ensuring that the restored data, after specified filtering, has minimum energy, say $\mathbf{Am} \approx \mathbf{0}$. Introduce the selection mask operator $\mathbf{K}$, a diagonal matrix with ones on the known data and zeros elsewhere (on the missing data). Thus $\mathbf{0} \approx \mathbf{A}(\mathbf{I}-\mathbf{K}+\mathbf{K})\mathbf{m}$ or

$$\mathbf{0} \quad \approx \quad \mathbf{A}(\mathbf{I}-\mathbf{K})\mathbf{m} \;+\; \mathbf{A}\mathbf{m}_k \;, \tag{5.27}$$

where we define $\mathbf{m}_k$ to be the data with missing values set to zero by $\mathbf{m}_k = \mathbf{Km}$.

A second way to find missing data is with the set of goals

$$\begin{array}{rcrcl} \mathbf{0} & \approx & \mathbf{Km} & - & \mathbf{m}_k \\ \mathbf{0} & \approx & \epsilon\mathbf{Am} & & \end{array} \tag{5.28}$$

and take the limit as the scalar $\epsilon \to 0$. At that limit, we should have the same result as equation (1.27).

There is an important philosophical difference between the first method and the second. The first method strictly honors the known data. The second method acknowledges that when data misfits the regularization theory, it might be the fault of the data so the data need not be strictly honored. Just what balance is proper falls to the numerical choice of $\epsilon$, a nontrivial topic.

A third way to find missing data is to precondition equation (1.28), namely, try the substitution $\mathbf{m} = \mathbf{A}^{-1}\mathbf{p}$.

$$\begin{array}{rcrcl} \mathbf{0} & \approx & \mathbf{K}\mathbf{A}^{-1}\mathbf{p} & - & \mathbf{m}_k \\ \mathbf{0} & \approx & \epsilon\mathbf{p} & & \end{array} \tag{5.29}$$

There is no simple way of knowing beforehand what is the best value of $\epsilon$. Practitioners like to see solutions for various values of $\epsilon$. Of course that can cost a lot of computational effort. Practical exploratory data analysis is more pragmatic. Without a simple clear theoretical basis, analysts generally begin from $\mathbf{p} = 0$ and abandon the fitting goal $\epsilon\mathbf{Ip} \approx 0$. Implicitly, they take $\epsilon = 0$. Then they examine the solution as a function of iteration, imagining that the solution at larger iterations corresponds to smaller $\epsilon$. There is an eigenvector analysis indicating some kind of basis for this approach, but I believe there is no firm guidance.

Before we look at coding details for the three methods of filling the empty bins, we'll compare results of trying all three methods. For the roughening operator $\mathbf{A}$, we'll take the helix derivative $\mathbf{H}$. This is logically equivalent to roughening with the gradient $\nabla$ because the (negative) laplacian operator is $\nabla'\nabla = \mathbf{H}'\mathbf{H}$.

### 5.6.1 SEABEAM: Filling the empty bins with a laplacian

Figure 1.8 shows a day's worth of data[1] collected at sea by SeaBeam, an apparatus for measuring water depth both directly under a ship, and somewhat off to the sides of the ship's track. The data is measurements of depth $h(x, y)$ at miscellaneous locations in the $(x, y)$-plane. The

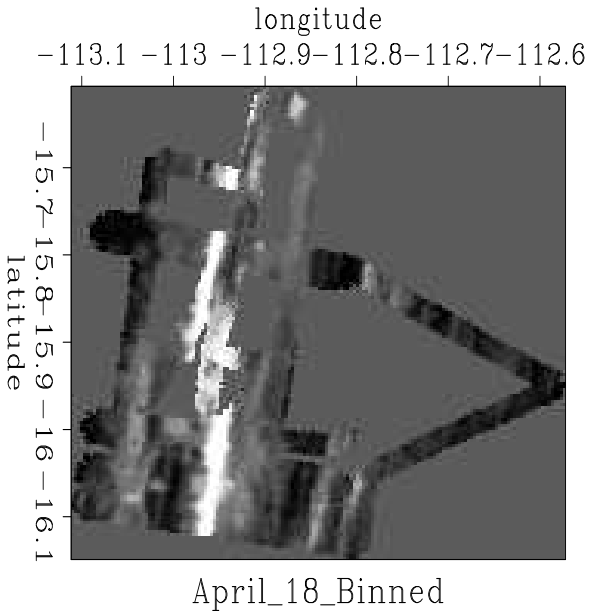


Figure 5.8: Depth of the ocean under ship tracks. Empty bins are displayed with an average depth $\bar{h}$. prc-seabin90 [ER]

locations are scattered about, according to various aspects of the ship's navigation and the geometry of the SeaBeam sonic antenna. Figure 1.8 was made by binning with `bin2()` on page 13 and equation (1.15). The spatial spectra of the noise in the data could be estimated where tracks cross over themselves. This might be worth while, but we do not pursue it now.

Here we focus on the empty mesh locations where no data is recorded (displayed with the value of the mean depth $\bar{h}$). These empty bins were filled with module `mis2` on page 19. Results are in Figure 1.9. In Figure 1.9 the left column results from 20 iterations while the right column results from 100 iterations.

The top row in Figure 1.9 shows that more iterations spreads information further into the region of missing data.

It turned out that the original method strictly honoring known data gave results so similar to the second method (regularizing) that the plots could not be visually distinguished. The middle row in Figure 1.9 therefore shows the difference in the result of the two methods. We see an outline of the transition between known and unknown regions. Obviously, the missing data is pulling known data towards zero.

The bottom row in Figure 1.9 shows that preconditioning spreads information to great distances much quicker but early iterations make little effort to honor the data. (Even though these results are for $\epsilon = 0$.) Later iterations make little change at long distance but begin to restore sharp details on the small features of the known topography.

[1] I'd like to thank Alistair Harding for this interesting data set named April 18.

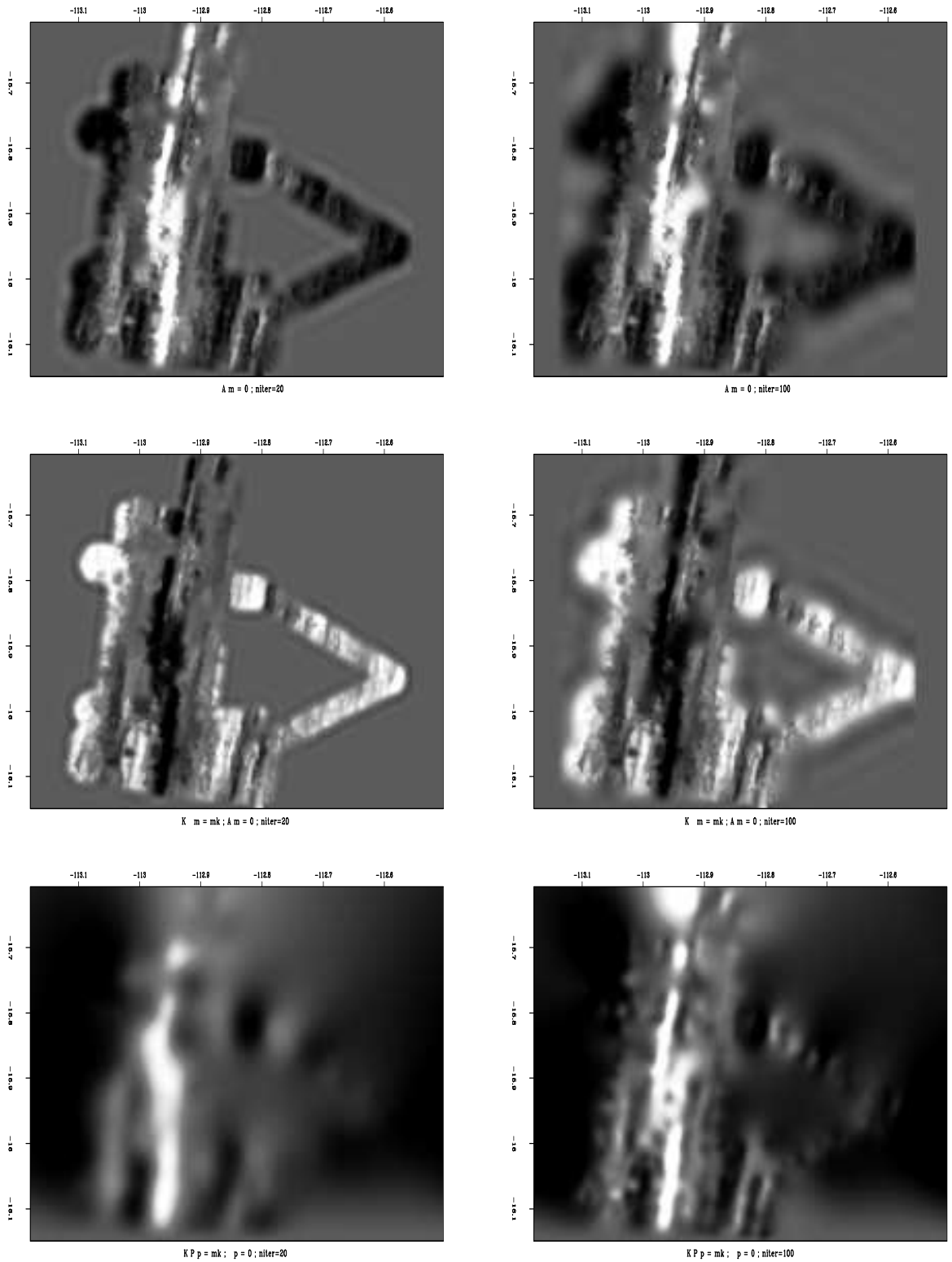


Figure 5.9: The ocean bottom restoring missing data with a helix derivative. prc-prcfill [ER,M]

What if we can only afford 100 iterations? Perhaps we should first do 50 iterations with preconditioning to develop the remote part of the solution and then do 50 iterations by one of the other two methods to be sure we attended to the details near the known data. A more unified approach (not yet tried, as far as I know) would be to unify the techniques. The conjugate direction method searches two directions, the gradient and the previous step. We could add a third direction, the smart direction of equation (1.14). Instead of having a $2 \times 2$ matrix solution like equation (2.67) for two distances, we would need to solve a $3 \times 3$ matrix for three.

Figure 1.9 has a few artifacts connected with the use of the helix derivative. Examine equation (4.11) to notice the shape of the helix derivative. In principle, it is infinitely long in the horizontal axis in both equation (4.11) and Figure 1.9. In practice, it is truncated. The truncation is visible as bands along the sides of Figure 1.9.

As a practical matter, no one would use the first two bin filling methods with helix derivative for the roughener because it is theoretically equivalent to the gradient operator $\nabla$ which has many fewer coefficients. Later, in Chapter 6 we'll find a much smarter roughening operator **A** called the Prediction Error Filter (PEF) which gives better results.

### 5.6.2 Three codes for inverse masking

The selection (or masking) operator **K** is implemented in `mask1()` on page 19.

```
module mask1 { # masking operator
  logical, dimension( :), pointer :: m
#% _init( m)
#% _lop( x, y)
  if( adj)
           where( m) x += y
  else  #
           where( m) y += x
}
```

All the results shown in Figure 1.9 were created with the module `mis2` on page 19. Code locations with `style=0,1,2` correspond to the fitting goals (1.27), (1.28), (1.29).

```
module mis2_mod {
  use mask1 + helicon + polydiv + cgstep_mod
  use solver_smp_mod + solver_reg_mod + solver_prc_mod
contains
  subroutine mis2( niter, xx, aa, known, style) {
    integer,                intent( in)     :: style
    integer,                intent( in)     :: niter
    type( filter),          intent( in)     :: aa
    logical, dimension( :), intent( in)     :: known
    real,    dimension( :), intent( in out) :: xx       # fitting variables
    real,    dimension( :), allocatable     :: dd
    logical, dimension( :), pointer         :: msk
    integer                                 :: nx
```

```
    nx = size( xx)
    allocate( dd(nx))
    allocate(msk(nx))
    if(style==0) {
            dd = 0.
            msk = .not.known
            call mask1_init( msk)
            call helicon_init( aa)
            call solver_smp( m=xx, d=dd, Fop=helicon_lop, stepper=cgstep,
                niter=niter, Jop=mask1_lop, m0=xx)
            call helicon_close()
    } else if(style==1) {
            dd=xx
            msk = known
            call mask1_init( msk)
            call helicon_init( aa)
            call solver_reg( m=xx, d=dd, Fop=mask1_lop, stepper=cgstep,
                niter=niter, Aop=helicon_lop, nAop=nx, eps=0.1)
            call helicon_close()
    } else {
            dd=xx
            msk = known
            call mask1_init( msk)
            call polydiv_init( nx, aa)
            call solver_prc( m=xx, d=dd, Fop=mask1_lop, stepper=cgstep,
                niter=niter, Sop=polydiv_lop, nSop=nx, eps=0.0)
            call polydiv_close()
    }
    call cgstep_close( )
    deallocate( msk,dd)
  }
}
```

## 5.7 THEORY OF UNDERDETERMINED LEAST-SQUARES

Construct theoretical data with

$$\mathbf{d} \quad = \quad \mathbf{F}\mathbf{m} \tag{5.30}$$

Assume there are fewer data points than model points and that the matrix $\mathbf{F}\mathbf{F}'$ is invertible. From the theoretical data we estimate a model $\mathbf{m}_0$ with

$$\mathbf{m}_0 \quad = \quad \mathbf{F}'(\mathbf{F}\mathbf{F}')^{-1}\mathbf{d} \tag{5.31}$$

To verify the validity of the estimate, insert the estimate (1.31) into the data modeling equation (1.30) and notice that the estimate $\mathbf{m}_0$ predicts the correct data. Notice that equation (1.31) is not the same as equation (2.27) which we derived much earlier. What's the difference? The central issue is which matrix of $\mathbf{F}\mathbf{F}'$ and $\mathbf{F}'\mathbf{F}$ actually has an inverse. If $\mathbf{F}$ is a rectangular matrix, then it is certain that one of the two is not invertible. (There are plenty of real cases where neither matrix is invertible. That's one reason we use iterative solvers.) Here we are dealing with the case with more model points than data points.

Now we will show that of all possible models $\mathbf{m}$ that predict the correct data, $\mathbf{m}_0$ has the least energy. (I'd like to thank Sergey Fomel for this clear and simple proof that does *not* use Lagrange multipliers.) First split (1.31) into an intermediate result $\mathbf{d}_0$ and final result:

$$\mathbf{d}_0 = (\mathbf{F}\mathbf{F}')^{-1}\mathbf{d} \tag{5.32}$$

$$\mathbf{m}_0 = \mathbf{F}'\mathbf{d}_0 \tag{5.33}$$

Consider another model ($\mathbf{x}$ not equal to zero)

$$\mathbf{m} = \mathbf{m}_0 + \mathbf{x} \tag{5.34}$$

which fits the theoretical data $\mathbf{d} = \mathbf{F}(\mathbf{m}_0 + \mathbf{x})$. Since $\mathbf{d} = \mathbf{F}\mathbf{m}_0$, we see that $\mathbf{x}$ is a null space vector.

$$\mathbf{F}\mathbf{x} = \mathbf{0} \tag{5.35}$$

First we see that $\mathbf{m}_0$ is orthogonal to $\mathbf{x}$ because

$$\mathbf{m}_0'\mathbf{x} = (\mathbf{F}'\mathbf{d}_0)'\mathbf{x} = \mathbf{d}_0'\mathbf{F}\mathbf{x} = \mathbf{d}_0'\mathbf{0} = 0 \tag{5.36}$$

Therefore,

$$\mathbf{m}'\mathbf{m} = \mathbf{m}_0'\mathbf{m}_0 + \mathbf{x}'\mathbf{x} + 2\mathbf{x}'\mathbf{m}_0 = \mathbf{m}_0'\mathbf{m}_0 + \mathbf{x}'\mathbf{x} \geq \mathbf{m}_0'\mathbf{m}_0 \tag{5.37}$$

so adding null space to $\mathbf{m}_0$ can only increase its energy. In summary, the solution $\mathbf{m}_0 = \mathbf{F}'(\mathbf{F}\mathbf{F}')^{-1}\mathbf{d}$ has less energy than any other model that satisfies the data.

Not only does the theoretical solution $\mathbf{m}_0 = \mathbf{F}'(\mathbf{F}\mathbf{F}')^{-1}\mathbf{d}$ have minimum energy, but the result of iterative descent will too, provided that we begin iterations from $\mathbf{m}_0 = 0$ or any $\mathbf{m}_0$ with no null-space component. In (1.36) we see that the orthogonality $\mathbf{m}_0'\mathbf{x} = 0$ does not arise because $\mathbf{d}_0$ has any particular value. It arises because $\mathbf{m}_0$ is of the form $\mathbf{F}'\mathbf{d}_0$. Gradient methods contribute $\Delta\mathbf{m} = \mathbf{F}'\mathbf{r}$ which is of the required form.

## 5.8 SCALING THE ADJOINT

First I remind you of a rarely used little bit of mathematical notation. Given a vector $\mathbf{m}$ with components $(m_1, m_2, m_3)$, the notation **diag m** means

$$\mathbf{diag}\,\mathbf{m} = \begin{bmatrix} m_1 & 0 & 0 \\ 0 & m_2 & 0 \\ 0 & 0 & m_3 \end{bmatrix} \tag{5.38}$$

Given the usual linearized fitting goal between data space and model space, $\mathbf{d} \approx \mathbf{F}\mathbf{m}$, the simplest image of the model space results from application of the adjoint operator $\hat{\mathbf{m}} = \mathbf{F}'\mathbf{d}$. Unless $\mathbf{F}$ has no physical units, however, the physical units of $\hat{\mathbf{m}}$ do not match those of $\mathbf{m}$, so we need a scaling factor. The theoretical solution $\mathbf{m}_{\text{theor}} = (\mathbf{F}'\mathbf{F})^{-1}\mathbf{F}'\mathbf{d}$ suggests that the

scaling units should be those of $(\mathbf{F}'\mathbf{F})^{-1}$. We could probe the operator $\mathbf{F}$ or its adjoint with white noise or a zero-frequency input. Bill Symes suggests we probe with the data $\mathbf{d}$ because it has the spectrum of interest. He proposes we make our image with $\hat{\mathbf{m}} = \mathbf{W}^2\mathbf{F}'\mathbf{d}$ where we choose the weighting function to be

$$\mathbf{W}^2 \quad = \quad \frac{\mathbf{diag}\ \mathbf{F}'\mathbf{d}}{\mathbf{diag}\ \mathbf{F}'\mathbf{F}\mathbf{F}'\mathbf{d}} \tag{5.39}$$

which obviously has the correct physical units. (The mathematical function **diag** takes a vector and lies it along the diagonal of a square matrix.) The weight $\mathbf{W}^2$ can be thought of as a diagonal matrix containing the ratio of two images. A problem with the choice (1.39) is that the denominator might vanish or might even be negative. The way to stabilize any ratio is suggested at the beginning of Chapter 2; that is, we revise the ratio $a/b$ to

$$\mathbf{W}^2 \quad = \quad \frac{\mathbf{diag} < ab >}{\mathbf{diag} < b^2 + \epsilon^2 >} \tag{5.40}$$

where $\epsilon$ is a parameter to be chosen, and the angle braces indicate the possible need for local smoothing.

To go beyond the scaled adjoint we can use $\mathbf{W}$ as a preconditioner. To use $\mathbf{W}$ as a preconditioner we define implicitly a new set of variables $\mathbf{p}$ by the substitution $\mathbf{m} = \mathbf{W}\mathbf{p}$. Then $\mathbf{d} \approx \mathbf{F}\mathbf{m} = \mathbf{F}\mathbf{W}\mathbf{p}$. To find $\mathbf{p}$ instead of $\mathbf{m}$, we do CD iteration with the operator $\mathbf{F}\mathbf{W}$ instead of with $\mathbf{F}$. As usual, the first step of the iteration is to use the adjoint of $\mathbf{d} \approx \mathbf{F}\mathbf{W}\mathbf{p}$ to form the image $\hat{\mathbf{p}} = (\mathbf{F}\mathbf{W})'\mathbf{d}$. At the end of the iterations, we convert from $\mathbf{p}$ back to $\mathbf{m}$ with $\mathbf{m} = \mathbf{W}\mathbf{p}$. The result after the first iteration $\hat{\mathbf{m}} = \mathbf{W}\hat{\mathbf{p}} = \mathbf{W}(\mathbf{F}\mathbf{W})'\mathbf{d} = \mathbf{W}^2\mathbf{F}'\mathbf{d}$ turns out to be the same as Symes scaling.

By (1.39), $\mathbf{W}$ has physical units inverse to $\mathbf{F}$. Thus the transformation $\mathbf{F}\mathbf{W}$ has no units so the $\mathbf{p}$ variables have physical units of data space. Experimentalists might enjoy seeing the solution $\mathbf{p}$ with its data units more than viewing the solution $\mathbf{m}$ with its more theoretical model units.

The theoretical solution for underdetermined systems $\mathbf{m} = \mathbf{F}'(\mathbf{F}\mathbf{F}')^{-1}\mathbf{d}$ suggests an alternate approach using instead $\hat{\mathbf{m}} = \mathbf{F}'\mathbf{W}_d^2\mathbf{d}$. A possibility for $\mathbf{W}_d^2$ is

$$\mathbf{W}_d^2 \quad = \quad \frac{\mathbf{diag}\ \mathbf{d}}{\mathbf{diag}\ \mathbf{F}\mathbf{F}'\mathbf{d}} \tag{5.41}$$

Experience tells me that a broader methodology is needed. Appropriate scaling is required in both data space and model space. We need something that includes a weight for each space, $\mathbf{W}_m$ and $\mathbf{W}_d$ where $\hat{\mathbf{m}} = \mathbf{W}_m\mathbf{F}'\mathbf{W}_d\mathbf{d}$.

I have a useful practical example (stacking in $v(z)$ media) in another of my electronic books (BEI), where I found both $\mathbf{W}_m$ and $\mathbf{W}_d$ by iterative guessing. But I don't know how to give you a simple strategy that is not iterative. Either this is a major unsolved opportunity for a theorist, or I'll need to write down my iterative guess.

The PhD thesis of James Rickett experiments extensively with data space and model space weighting functions in the context of seismic velocity estimation.

## 5.9 A FORMAL DEFINITION FOR ADJOINTS

In mathematics, adjoints are defined a little differently than we have defined them here (as matrix transposes).[2] The mathematician begins by telling us that we cannot simply form any dot product we want. We are not allowed to take the dot product of any two vectors in model space $\mathbf{m}_1 \cdot \mathbf{m}_2$ or data space $\mathbf{d}_1 \cdot \mathbf{d}_2$. Instead, we must first transform them to a preferred coordinate system. Say $\tilde{\mathbf{m}}_1 = \mathbf{M}\mathbf{m}_1$ and $\tilde{\mathbf{d}}_1 = \mathbf{D}\mathbf{d}_1$, etc for other vectors. We complain we do not know $\mathbf{M}$ and $\mathbf{D}$. They reply that we do not really need to know them but we do need to have the inverses (aack!) of $\mathbf{M}'\mathbf{M}$ and $\mathbf{D}'\mathbf{D}$. A pre-existing common notation is $\sigma_m^{-2} = \mathbf{M}'\mathbf{M}$ and $\sigma_d^{-2} = \mathbf{D}'\mathbf{D}$. Now the mathematician buries the mysterious new positive-definite matrix inverses in the definition of dot product $< \mathbf{m}_1, \mathbf{m}_2 >= \mathbf{m}_1'\mathbf{M}'\mathbf{M}\mathbf{m}_2 = \mathbf{m}_1'\sigma_m^{-2}\mathbf{m}_2$ and likewise with $< \mathbf{d}_1, \mathbf{d}_2 >$. This suggests a total reorganization of our programs. Instead of computing $(\mathbf{m}_1'\mathbf{M}')(\mathbf{M}\mathbf{m}_2)$ we could compute $\mathbf{m}_1'(\sigma_m^{-2}\mathbf{m}_2)$. Indeed, this is the "conventional" approach. This definition of dot product would be buried in the solver code. The other thing that would change would be the search direction $\Delta\mathbf{m}$. Instead of being the gradient as we have defined it $\Delta\mathbf{m} = \mathbf{L}'\mathbf{r}$, it would be $\Delta\mathbf{m} = \sigma_m^{-2}\mathbf{L}'\sigma_d^{-2}\mathbf{r}$. A mathematician would *define* the adjoint of $\mathbf{L}$ to be $\sigma_m^{-2}\mathbf{L}'\sigma_d^{-2}$. (Here $\mathbf{L}'$ remains matrix transpose.) You might notice this approach nicely incorporates both residual weighting and preconditioning while yet evading the issue of where we get the matrices $\sigma_m^2$ and $\sigma_d^2$ or how we invert them. Fortunately, upcoming chapter **??** suggests how, in image estimation problems, to obtain sensible estimates of the elusive operators $\mathbf{M}$ and $\mathbf{D}$. Paranthetically, modeling calculations in physics and engineering often use similar mathematics in which the role of $\mathbf{M}'\mathbf{M}$ is not so mysterious. Kinetic energy is mass times velocity squared. Mass can play the role of $\mathbf{M}'\mathbf{M}$.

So, should we continue to use $(\mathbf{m}_1'\mathbf{M}')(\mathbf{M}\mathbf{m}_2)$ or should we take the conventional route and go with $\mathbf{m}_1'(\sigma_m^{-2}\mathbf{m}_2)$? One day while benchmarking a wide variety of computers I was shocked to see some widely differing numerical results. Now I know why. Consider adding $10^7$ identical positive floating point numbers, say 1.0's, in an arithmetic with precision of $10^{-6}$. After you have added in the first $10^6$ numbers, the rest will all truncate in the roundoff and your sum will be wrong by a factor of ten. If the numbers were added in pairs, and then the pairs added, etc, there would be no difficulty. Precision is scary stuff!

It is my understanding and belief that there is nothing wrong with the approach of this book, in fact, it seems to have some definite advantages. While the conventional approach requires one to compute the adjoint correctly, we do not. The method of this book (which I believe is properly called conjugate directions) has a robustness that, I'm told, has been superior in some important geophysical applications. The conventional approach seems to get in trouble when transpose operators are computed with insufficient precision.

[2] I would like to thank Albert Tarantola for suggesting this topic.

# Index

invint2 module, 15

mask1 operator module, 19
mis2 module, 19
module
    `invint2`, Inverse linear interpolation, 15
    `mis2`, Missing data interpolation with and without preconditioning, 19
    `solver_prc`, Preconditioned solver, 6
    `vrms2int`, Converting RMS to interval velocity, 9

operator
    `mask1`, copy under mask, 19

solver_prc module, 6

vrms2int module, 9