Chapter 4

The helical coordinate

For many years it has been true that our most powerful signal-analysis techniques are in *one*-dimensional space, while our most important applications are in *multi*-dimensional space. The helical coordinate system makes a giant step towards overcoming this difficulty.

4.1 FILTERING ON A HELIX

Figure 1.1 shows some two-dimensional shapes that are convolved together. The left panel shows an impulse response function, the center shows some impulses, and the right shows the superposition of responses.

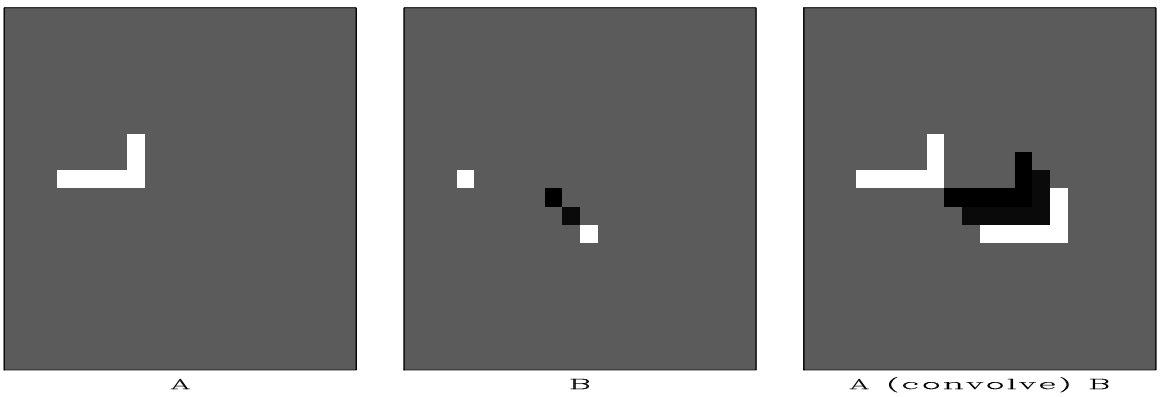


Figure 4.1: Two-dimensional convolution (as performed in one dimension). hlx-diamond90 [ER]

A surprising, indeed amazing, fact is that Figure 1.1 was not computed with a two-dimensional convolution program. It was computed with a one-dimensional computer program. It could have been done with anybody's one-dimensional convolution

program, either in the time domain or in the fourier domain. This magical trick is done with the helical coordinate system.

A basic idea of filtering, be it in one dimension, two dimensions, or more, is that you have some filter coefficients and some sampled data; you pass the filter over the data; at each location you find an output by crossmultiplying the filter coefficients times the underlying data and summing the terms.

The helical coordinate system is much simpler than you might imagine. Ordinarily, a plane of data is thought of as a collection of columns, side by side. Instead, imagine the columns stored end-to-end, and then coiled around a cylinder. This is the helix. Fortran programmers will recognize that Fortran's way of storing 2-D arrays in one-dimensional memory is exactly what we need for this helical mapping. Seismologists sometimes use the word "supertrace" to describe a collection of seismograms stored "end-to-end". Figure 1.2 shows a helical mesh for 2-D data on a cylinder. Darkened squares depict a 2-D filter shaped like the Laplacian operator. The input data, the filter, and the output data are all on helical meshes all of which could be unrolled into linear strips. A compact 2-D filter like a Laplacian, on a helix is a sparse 1-D filter with long empty gaps.

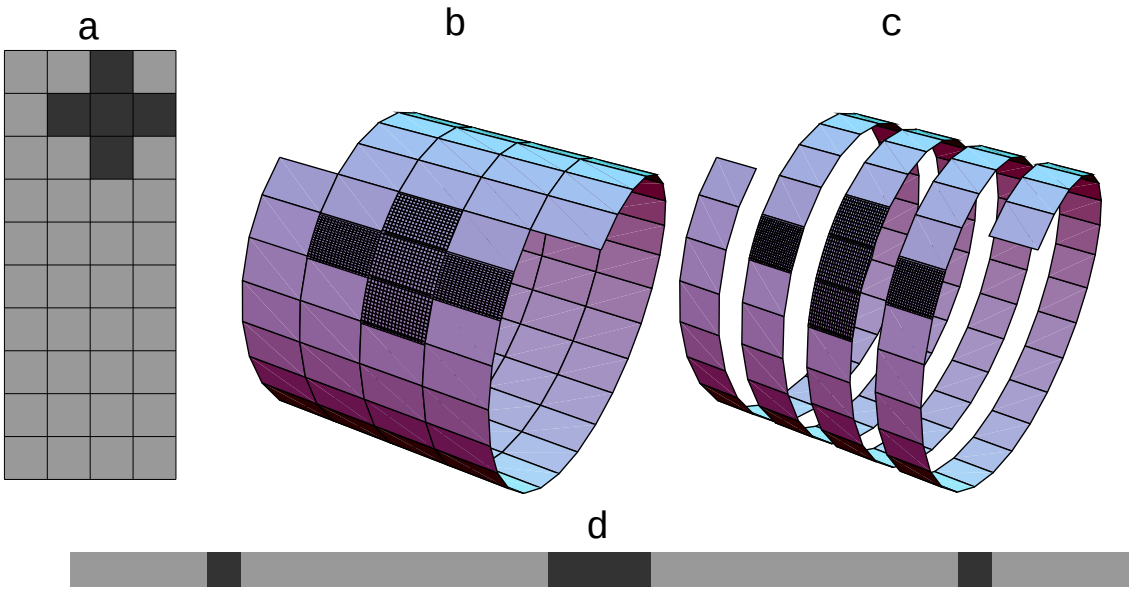


Figure 4.2: Filtering on a helix. The same filter coefficients overlay the same data values if the 2-D coils are unwound into 1-D strips. (drawing by Sergey Fomel) hlx-sergey-helix [NR]

Since the values output from filtering can be computed in any order, we can slide the filter coil over the data coil in any direction. The order that you produce the outputs is irrelevant. You could compute the results in parallel. We could, however, slide the filter over the data in the screwing order that a nut passes over a bolt. The

screw order is the same order that would be used if we were to unwind the coils into one-dimensional strips and convolve them across one another. The same filter coefficients overlay the same data values if the 2-D coils are unwound into 1-D strips. The helix idea allows us to obtain the same convolution output in either of two ways, a one-dimensional way, or a two-dimensional way. I used the one-dimensional way to compute the obviously two-dimensional result in Figure 1.1.

Deconvolution (polynomial division) can undo convolution (polynomial multiplication). A magical property of the helix is that we can consider 1-D convolution to be the same as 2-D convolution. Hence is a second magical property: We can use 1-D *de*convolution to undo convolution, whether that convolution was 1-D or 2-D. Thus, we have discovered how to undo 2-D convolution. We have discovered that 2-D deconvolution on a helix is equivalent to 1-D deconvolution. The helix enables us to do multidimensional deconvolution.

Deconvolution is recursive filtering. Recursive filter outputs cannot be computed in parallel, but must be computed sequentially as in one dimension, namely, in the order that the nut screws on the bolt.

Recursive filtering sometimes solves big problems with astonishing speed. It can propagate energy rapidly for long distances. Unfortunately, recursive filtering can also be unstable. The most interesting case, near resonance, is also near instability. There is a large literature and extensive technology about recursive filtering in one dimension. The helix allows us to apply that technology to two (and more) dimensions. It is a huge technological breakthrough.

In 3-D we simply append one plane after another (like a 3-D Fortran array). It is easier to code than to explain or visualize a spool or torus wrapped with string, etc.

4.1.1 Review of 1-D recursive filters

Convolution is the operation we do on polynomial coefficients when we multiply polynomials. Deconvolution is likewise for polynomial division. Often these ideas are described as polynomials in the variable Z. Take $X(Z)$ to denote the polynomial whose coefficients are samples of input data, and let $A(Z)$ likewise denote the filter. The convention I adopt here is that the first coefficient of the filter has the value $+1$, so the filter's polynomial is $A(Z) = 1 + a_1 Z + a_2 Z^2 + \cdots$. To see how to convolve, we now identify the coefficient of Z^k in the product $Y(Z) = A(Z)X(Z)$. The usual case (k larger than the number N_a of filter coefficients) is

$$y_k \quad = \quad x_k + \sum_{i=1}^{N_a} a_i x_{k-i} \tag{4.1}$$

Convolution computes y_k from x_k whereas deconvolution (also called back substitution) does the reverse. Rearranging (1.1) we get

$$x_k \quad = \quad y_k - \sum_{i=1}^{N_a} a_i x_{k-i} \tag{4.2}$$

where now we are finding the output x_k from its past outputs x_{k-i} and from the present input y_k. We see that the deconvolution process is essentially the same as the convolution process, except that the filter coefficients are used with opposite polarity; and they are applied to the past *outputs* instead of the past *inputs*. That is why deconvolution must be done sequentially while convolution can be done in parallel.

4.1.2 Examples of simple 2-D recursive filters

Figures 1.3 and 1.4 contain this 2-D filter

$$\begin{bmatrix} 0 & -1/4 \\ 1 & -1/4 \\ -1/4 & -1/4 \end{bmatrix} \tag{4.3}$$

Let us experiment using this 2-D filter as a recursive filter. In Figure 1.3 the input is shown on the left. This input contains two copies of the filter (1.3) near the top of the frame and some impulses near the bottom boundary. The second frame in Figure 1.3 is the result of deconvolution by the filter (1.3). Notice that deconvolution turns the filter itself into an impulse, while it turns the impulses into comet-like images. The use of a helix is evident by the comet images wrapping around the vertical axis.

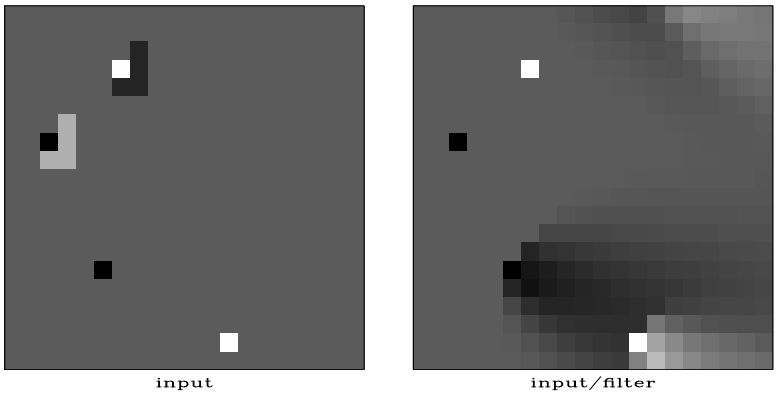


Figure 4.3: Illustration of 2-D deconvolution. Left is the input. Right is after deconvolution with the filter (1.3) hlx-wrap90 [ER]

In Figure 1.4, many inputs are tested. Starting from the left are a low-pass blob, a Ricker wavelet, the filter (1.3) itself, and a couple impulses, one near the bottom boundary. The second frame shows deconvolution by the filter (1.3). The third frame

compounds the second frame with an adjoint (reverse time) deconvolution. (Instead of blowing plumes to the right, it blows them to the left.) The fourth frame convolves the result with the original filter and its adjoint; and we see we are back where we started. No errors, no evidence remains of any of the boundaries where we have wrapped and truncated!

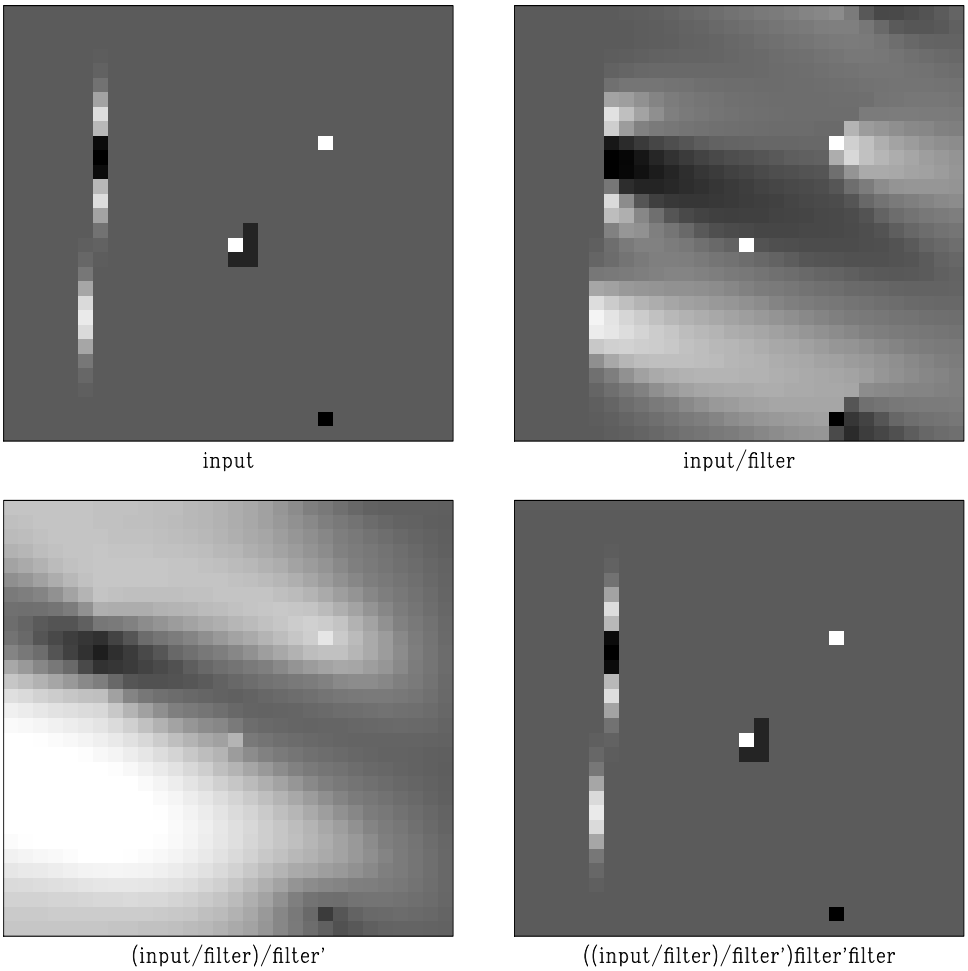


Figure 4.4: Recursive filtering backwards (leftward on the space axis) is done by the *adjoint* of 2-D deconvolution. Here we see that 2-D deconvolution compounded with its adjoint is exactly inverted by 2-D convolution and its adjoint. hlx-pdadj90 [ER]

In seismology we often have occasion to steer summation along beams. Such an impulse response is shown in Figure 1.5. Finally, I have long had an interest in filters that would destroy plane waves. The inverse of such a filter creates plane waves. A filter that creates two plane waves is illustrated in figure 1.6.

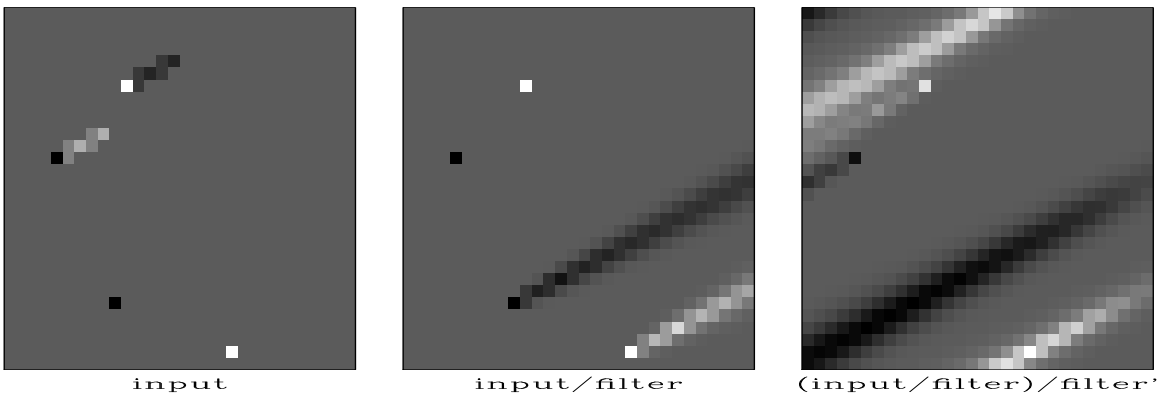
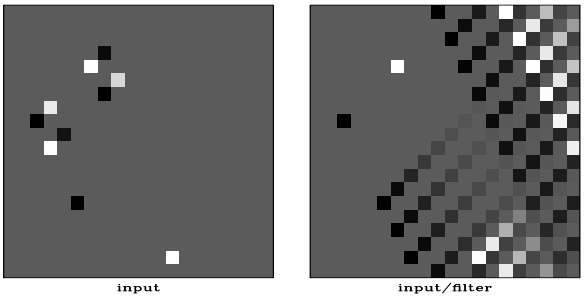


Figure 4.5: A simple low-order 2-D filter whose inverse times its inverse adjoint, is approximately a dipping seismic arrival. hlx-dip90 [ER]

Figure 4.6: A simple low-order 2-D filter whose inverse contains plane waves of two different dips. One of them is spatially aliased. hlx-waves90 [ER]



4.1.3 Coding multidimensional de/convolution

Let us unroll the filter helix seen in Figure 1.2 and see what we have. Start from the idea that a 2-D filter is generally made from a cluster of values near one another in two dimensions similar to the Laplacian operator in the figure. We see that in the helical approach, a 2-D filter is a 1-D filter containing some long intervals of zeros. The intervals are about the length of a 1-D seismogram.

Our program for 2-D convolution with a 1-D convolution program, could convolve with the somewhat long 1-D strip, but it is much more cost effective to ignore the many zeros, which is what we do. We do not multiply by the backside zeros, nor do we even store them in memory. Whereas an ordinary convolution program would do time shifting by a code line like `iy=ix+lag`, Module `helicon` on page 7 ignores the many zero filter values on backside of the tube by using the code `iy=ix+lag(ia)` where a counter `ia` ranges over the nonzero filter coefficients. Before operator `helicon` is invoked, we need to prepare two lists, one list containing nonzero filter coefficients `aa(ia)`, and the other list containing the corresponding lags `lag(ia)` measured to include multiple wraps around the helix. For example, the 2-D Laplace operator

$$\begin{array}{ccc} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{array} \tag{4.4}$$

can be thought of as the 1-D filter

$$1, 0, \cdots, 0, 1, -4, 1, 0, \cdots, 0, 1, 0, \cdots \tag{4.5}$$

The first filter coefficient in equation (1.5) is +1 as implicit to module `helicon`. To apply the Laplacian on a 1000×1000 mesh requires the filter inputs:

```
 i    lag(i)   aa(i)
---   ------   -----
 1      999      1
 2     1000     -4
 3     1001      1
 4     2000      1
```

Operator `helicon` did the convolution job for Figure 1.1. The first half of operator `helicon` does adjoint filtering. The adjoint of filtering is filtering backwards. The ends of the 1-D axis are a little trickier than you might imagine, but there is no need to examine them now. (One end is transient, the other truncated.)

```
module helicon {                        # Convolution, inverse to deconvolution.
#                            Designed for gapped filters (helical 2-D, 3-D, etc).
#            Requires the filter be causal with an implicit "1." at the onset.
real,    dimension (:), pointer :: aa
integer, dimension (:), pointer :: lag
#%  _init ( aa, lag)
if( any( lag <= 0)) call erexit ("Filter is not causal")
#%  _lop ( xx,  yy)
integer iy, ix, ia
if( adj)              # zero lag
        xx += yy
else
        yy += xx
do ia= 1, size(aa) {
do ix= 1, size(xx)-lag(ia) {    iy = ix + lag(ia)
        if( adj)
                        xx(ix) += yy(iy) * aa(ia)
        else
                        yy(iy) += xx(ix) * aa(ia)
        }}
}
```

The module `polydiv` on page 20 was coded in such a way that it works on a helix.

4.1.4 Causality in two-dimensions

Time-series analysis is rich with concepts that the helix now allows us to apply to many dimensions. First is the notion of an impulse function. Observe that an impulse function on the 2-D surface of the helical cylinder maps to an impulse function on

the 1-D line of the unwound coil. An autocorrelation function that is an impulse corresponds both to a white (constant) spectrum in 1-D and to a white (constant) spectrum in 2-D. An autocorrelation of a typical two-dimensional data field will drop off with two-dimensional distance from the zero lag. On the one-dimensional helix, the autocorrelation gets re-energized when the lag is an integer multiple of the circumference of the helix. A causal filter in one dimension has a curious shape on the two-dimensional helix. I adopt the convention that the zero-lag response of the 1-D filter has the value "1". In one dimension, the causal filter has zeros before the "1" and various values after it. Supposing that nonzero filter coefficients lie within a short distance (two lags) from the "1", we can extract from the helix the 1-D causal filter and view it as a two-dimensional array

$$\begin{array}{ccccccc} h & c & 0 & & h & c & \cdot & & \cdot & \cdot & 0 \\ p & d & 0 & & p & d & \cdot & & \cdot & \cdot & 0 \\ q & e & \mathbf{1} & = & q & e & \cdot & + & \cdot & \cdot & \mathbf{1} \\ s & f & a & & s & f & a & & \cdot & \cdot & \cdot \\ u & g & b & & u & g & b & & \cdot & \cdot & \cdot \end{array} \tag{4.6}$$

$$2-\text{D filter} \quad = \quad \text{variable} \quad + \quad \text{constrained}$$

where $a, b, c, ..., u$ are adjustable coefficients. Thus we conclude that the 2-D analog of a 1-D causal filter has its abrupt beginning along the side of the 2-D filter.

4.2 FACTORING THE LAPLACIAN ON A HELIX

Discretize the (x, y)-plane to an $N \times M$ array and pack the array into a vector of $N \times M$ components. Likewise pack the Laplacian operator $\partial_{xx} + \partial_{yy}$ into a matrix.

For a 4×3 plane, that matrix is shown in equation (1.7).

$$
-\nabla^2 \;=\;
\left[\begin{array}{cccc|cccc|cccc}
-4 & 1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
1 & -4 & 1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & 1 & -4 & 1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & 1 & -4 & h & \cdot & \cdot & 1 & \cdot & \cdot & \cdot & \cdot \\
\hline
1 & \cdot & \cdot & h & -4 & 1 & \cdot & \cdot & 1 & \cdot & \cdot & \cdot \\
\cdot & 1 & \cdot & \cdot & 1 & -4 & 1 & \cdot & \cdot & 1 & \cdot & \cdot \\
\cdot & \cdot & 1 & \cdot & \cdot & 1 & -4 & 1 & \cdot & \cdot & 1 & \cdot \\
\cdot & \cdot & \cdot & 1 & \cdot & \cdot & 1 & -4 & h & \cdot & \cdot & 1 \\
\hline
\cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & h & -4 & 1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & 1 & -4 & 1 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & 1 & -4 & 1 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & 1 & \cdot & \cdot & 1 & -4
\end{array}\right]
\tag{4.7}
$$

The two-dimensional matrix of coefficients for the Laplacian operator is shown in (1.7), where, on a cartesian space, $h = 0$, and in the helix geometry, $h = 1$. (A similar partitioned matrix arises from packing a cylindrical surface into a 4×4 array.) Notice that the partitioning becomes transparent for the helix, $h = 1$.

With the partitioning thus invisible, the matrix simply represents one-dimensional convolution and we have an alternative analytical approach, one-dimensional Fourier Transform. We often need to solve sets of simultaneous equations with a matrix similar to (1.7). A costly method is to factor the matrix into upper and lower triangular form that can be "backsolved" which in this case amounts to recursive filtering.

The Fourier approach of the next section is roughly equivalent to factoring $-\nabla^2$ into upper and lower triangular matrices, but it is much faster owing to the convolutional nature of the matrix. The (negative of the) Laplacian operator is regarded as an autocorrelation

$$\mathbf{r} = (\cdots, -1, \cdots, -1, 4, -1, \cdots, -1, \cdots) \tag{4.8}$$

Using the Kolmogoroff spectral-factorization method, a wavelet is found which has this autocorrelation. We'll see how to compute this wavelet

$$\mathbf{a} \quad = \quad (1.791, -.651, -.044, -.024, \cdots, -.044, -.087, -.200, -.558) \tag{4.9}$$

whose autocorrelation is (1.8). Displaying these signals on a helix, the 2-D autocorrelation is:

$$\mathbf{r} \quad = \quad \begin{bmatrix} & -1 & \\ -1 & 4 & -1 \\ & -1 & \end{bmatrix} \tag{4.10}$$

and the 2-D wavelet with this autocorrelation is

$$\mathbf{a} \quad = \quad \begin{bmatrix} & & & & 1.791 & -.651 & -.044 & -.024 & \cdots \\ \cdots & -.044 & -.087 & -.200 & -.558 & & & & \end{bmatrix} \tag{4.11}$$

where now I am wrapping the top row around to a second row. In the representation (1.11) we see the coefficients diminishing rapidly away from maximum value 1.791. In a more abstract notation, we might write

$$-\nabla^2 \quad = \quad \mathbf{A}'\mathbf{A} \tag{4.12}$$

Where the triangular matrix $\mathbf{A}$ is constructed from the row (1.9) as follows: Each row of the matrix $\mathbf{A}'$ contains the row (1.9) shifted along itself so that the 1.791 lies along the main diagonal of the matrix.

Unfortunately, we see that the factored operator $\mathbf{A}$ has a great number of nonzero terms, but fortunately they seem to be converging rapidly so truncation seems reasonable. We can use this feedback operator to solve Poisson's equation very rapidly. Polynomial division and its adjoint gives us $\mathbf{p} = (\mathbf{q}/\mathbf{A})/\mathbf{A}'$ which means that we have solved the PDE $\nabla^2\mathbf{p} = -\mathbf{q}$ by using polynomial division on a helix. Using the seven coefficients shown, the cost is fourteen multiplications (because we need to run both ways) per mesh point. An example is shown in Figure 1.7.

We are all aware of the factorization of ∇^2 into a divergence dotted into a gradient, where the divergence is the adjoint of the gradient. In two-dimensional physical space, the gradient maps one field to *two* fields. The factorization of $-\nabla^2$ with the helix gives us an operator that maps one field to *one* field.

Any fact this basic should be well known in some arcane field of mathematics or theoretical physics. Meanwhile, being ignorant of any pre-existant name, I have chosen the name "helix derivative" or "helical derivative" for the operator $\mathbf{A}$. (Admittedly, the concept exists on an infinite cartesian plane without a helix, but all my codes in a finite space involve the helix, and the helix concept led me to it.)

Our construction makes $\mathbf{A}$ have the energy spectrum $k_x^2 + k_y^2$, so the magnitude of the Fourier transform is $\sqrt{|k_x^2 + k_y^2|}$. This rotationally invariant function in the Fourier domain contrasts sharply with the nonrotationally invariant function shape in (x, y)-space. The difference must arise from the phase spectrum. The factorization (1.11) is nonunique in that causality associated with the helix mapping can be defined along either x or y axes; thus the operator (1.11) can be rotated or reflected.

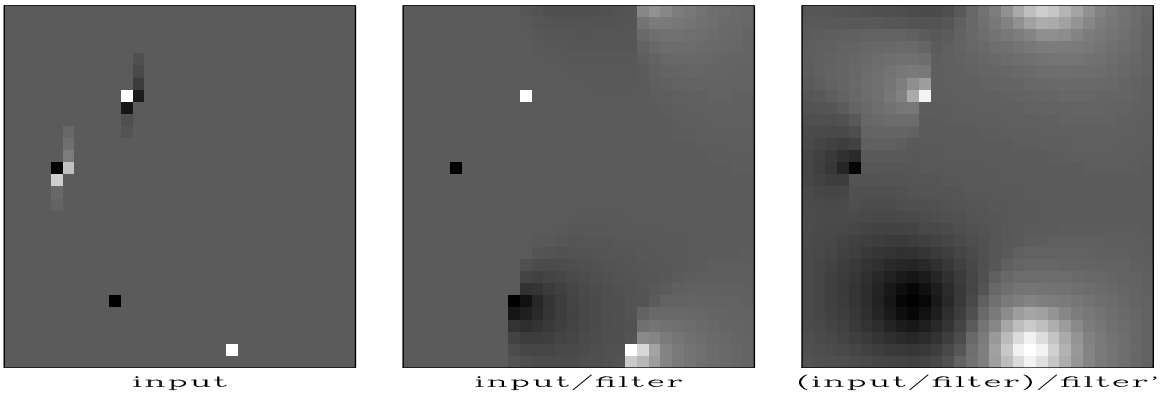


Figure 4.7: Deconvolution by a filter whose autocorrelation is the two-dimensional Laplacian operator. Amounts to solving the Poisson equation. Left is $\mathbf{q}$; Middle is $\mathbf{q}/\mathbf{A}$; Right is $(\mathbf{q}/\mathbf{A})/\mathbf{A}'$. hlx-lapfac90 [ER]

4.3 SPECTRAL FACTORIZATION CODE AND THEORY

My book PVI gives a full account of spectral factorization. Here is subroutine `lapfac()` which was used in the previous section to factor the Laplacian operator, and I give a sketch of the underlying Kolmogoroff theory. To invoke this subroutine, you need to supply one side of an autocorrelation function. For example, let us specify the negative of the 2-D Laplacian in a vector `n =` 256×256 points long.

```
rr(1)     =  4.00004
rr(2)     = -1.
rr(1+256) = -1.
```

The reason for the 4.00004 instead of 4.0 arises in the theory below where we are required to take the logarithm of the spectrum. The spectrum of the Laplacian operator is $k_x^2 + k_y^2$, which vanishes at the point $(k_x, k_y) = (0, 0)$ where the logarithm is minus infinity. The .00004 prevents that.

Subroutine `lapfac()` finds the helical derivative (factored negative Laplacian) and then prepares the required filter coefficient tables for the helix convolution and deconvolution subroutines.

```
module lapfac_mod { # Factor 2-D Laplacian.  Prepare input tables for helicon()
use autofac_mod
contains
  subroutine lapfac(e, n1, aa, lag) {
    real,                    intent (in)  :: e
    integer,                 intent (in)  :: n1
    integer, dimension (:),  intent (out) :: lag
    real,    dimension (:),  intent (out) :: aa
    complex, dimension ( 1024*64)         :: rr
    real                                  :: scale
```

```
    integer                                 :: i, na
    rr = 0.
    rr(1)      = 4.00004 + e
    rr(2)      = -1.
    rr(1+1024) = -1
    call autofac( rr);   scale= rr(1);   rr = rr/scale
    na = size (lag)
    do i=1,na/2 {
        lag(i) = i;          lag(na/2+i) =      n1    -(i-1)
         aa(i) = rr(i+1);     aa(na/2+i) = rr( 1024+1 -(i-1))
    }
  }
}
```

Subroutine `lapfac()` has its main job done by subroutine `autofac()` which will be shown after we examine the basic theory.

4.3.1 The exponential of a causal is causal.

Begin with a **causal** filter response c_t and its associated $C(Z)$. The Z-transform $C(Z)$ could be evaluated, giving a complex value for each real ω. This complex value could be exponentiated to get another value, say

$$B(Z(\omega)) \quad = \quad e^{C(Z(\omega))} \tag{4.13}$$

Next, we could inverse transform $B(Z(\omega))$ back to b_t. We will prove the amazing fact that b_t must be causal too.

First notice that if $C(Z)$ has no negative powers of Z, then $C(Z)^2$ does not either. Likewise for the third power or any positive integer power, or sum of positive integer powers. Now recall the basic power-series definition of the exponential function:

$$e^x \quad = \quad 1 + x + \frac{x^2}{2} + \frac{x^3}{2 \cdot 3} + \frac{x^4}{2 \cdot 3 \cdot 4} + \frac{x^5}{2 \cdot 3 \cdot 4 \cdot 5} + \cdots \tag{4.14}$$

Including equation (1.13) gives the **exponential of a causal**:

$$B(Z) \quad = \quad e^{C(Z)} \quad = \quad 1 + C(Z) + \frac{C(Z)^2}{2} + \frac{C(Z)^3}{2 \cdot 3} + \frac{C(Z)^4}{2 \cdot 3 \cdot 4} + \cdots \tag{4.15}$$

Each term in the infinite series corresponds to a causal response, so the sum, b_t, is causal. (If you have forgotten the series for the exponential function, then recall that the solution to $dy/dx = y$ is the definition of the exponential function $y(x) = e^x$, and that the power series satisfies the differential equation term by term, so it must be the exponential too. The factorials in the denominators assure us that the power series always converges, i.e., it is finite for any finite x.)

Putting one polynomial into another or one infinite series into another is an onerous task, even if it does lead to a wavelet that is exactly causal. In practice we do

operations that are conceptually the same, but for speed we do them with discrete Fourier transforms. The disadvantage is periodicity, i.e., negative times are represented computationally like negative frequencies. Negative times are the last half of the elements of a vector, so there can be some blurring of late times into negative ones. The subroutine we will use for Fourier transformation is `fts()`

```
module fft { #   complex fourier transform.
contains
  subroutine fts( signi, rr ) {
  #   if (signi>0) scale = 1; else scale=1/nx
  #
  #                        nx          signi*2*pi*i*(j-1)*(k-1)/nx
  #   rr(k)  =  scale * sum rr(j) * e
  #                       j=1             for k=1,2,...,nx=2**integer
  #
  integer signi, nx, i, j, k, m, istep
  real    arg
  complex rr(:), cw, cdel, ct
  nx = size (rr); if( nx != pad2(nx) ) call erexit('fts: nx not a power of 2')
  if( signi < 0) rr = rr / nx
  j = 1;  k = 1
  do i= 1, nx {
        if (i<=j) { ct = rr(j); rr(j) = rr(i); rr(i) = ct }
        m = nx/2
        do while (j>m .and. m>1) { j = j-m; m = m/2 }
        j = j+m
        }
  do {
        istep = 2*k;   cw = 1.;   arg = signi*3.14159265/k
        cdel = cmplx( cos(arg), sin(arg))
        do m= 1, k {
                do i= m, nx, istep
                        { ct=cw*rr(i+k);  rr(i+k)=rr(i)-ct;  rr(i)=rr(i)+ct }
                cw = cw * cdel
                }
        k = istep
        if(k>=nx) exit
        }
   }
   integer function pad2( n ) {
   integer n
   pad2 = 1
   do while( pad2 < n )
        pad2 = pad2 * 2
   }
}
```

4.3.2 Finding a causal wavelet from a prescribed spectrum

Now for the task of going from a spectrum to a causal wavelet. Take as given the spectrum of the causal wavelet $B(Z)$. This means that we are not given $B(Z)$ itself, but we are given $S(Z) = \overline{B}(1/Z)B(Z)$. Assuming no zeros in the spectrum $S(\omega)$, it is easy to find the log of the spectrum $U = \ln S$. The spectrum may be specified as **autocorrelation** coefficients or values on the unit circle $Z = e^{i\omega}$. Thus,

$$\overline{B}(1/Z)\, B(Z) = S(Z) = e^{\ln S(Z)} = e^{U(Z)} = e^{\overline{C}(1/Z)+C(Z)} = e^{\overline{C}(1/Z)}\, e^{C(Z)} \tag{4.16}$$

Given the spectrum $S(Z)$ for each value on the unit circle, we could deduce the **log spectrum** $U(Z) = \ln S(Z)$ at each point on the unit circle:

$$U(Z) \quad = \quad \ln[S(Z)] \quad = \quad \overline{C}(1/Z) + C(Z) \tag{4.17}$$

This is the answer we have been looking for. Given $U(Z)$ for all real values of ω, we could inverse transform to the time domain, obtaining the two-sided function $u_t = \bar{c}_{-t} + c_t$. Setting to zero the coefficients at negative times eliminates $\bar{c}_{-t}$, leaving just c_t; hence $C(Z)$. And we already know that the exponential of $C(Z)$ gives $B(Z)$ with a causal b_t. This method is known as "**Kolmogoroff spectral factorization**," after the mathematician who discovered it.

The autocorrelation factorization subroutine is `autofac()`

```
module autofac_mod { # Autocorrelation factorization.
use fft
contains
  subroutine autofac( rr) {
    complex, dimension (:) :: rr # input:  rr = autocorrelation
    integer n                    # output: rr = min phase wavelet
    n = size (rr)
    rr(n:n/2:-1) = conjg( rr(2:n/2+2));    call fts( +1, rr) # make spectrum
    rr = clog( rr);                        call fts( -1, rr)
    rr(1    ) = rr(1    ) / 2.
    rr(1+n/2) = rr(1+n/2) / 2.
    rr(1+n/2+1: n) = 0.;                   call fts( +1, rr)
    rr = cexp( rr);                        call fts( -1, rr)
  }
}
```

where the inputs and outputs were explained earlier with subroutine `lapfac()` on page 11. Subroutine `autofac()` is based on the classic 1-D discrete FT subroutine `fts()` on page 13.

4.3.3 Why the causal wavelet can be a divisor

The danger of polynomial division is that the coefficients will diverge like this, $1/(1-4Z) = 1 + 4Z + 16Z^2 + 64Z^3 + 256Z^4 + 1024Z^5 + \cdots$. The polynomial $1 - 4Z$

is an example of one that cannot be used in polynomial division. Our factoring of operators like the Laplacian gives us one factor that works with polynomial division in the forward time direction and a reversed one for the backward time direction.

Next we see why the causal wavelet $B(Z)$, of the Kolmogoroff theory can be a divisor, namely, why the inverse does not diverge. We have our wavelet in the form $B(Z) = e^{C(Z)}$. Consider another wavelet $A(Z) = e^{-C(Z)}$, constructed analogously. By the same reasoning, a_t is also causal. Since $A(Z)B(Z) = 1$, we have found a causal, inverse wavelet. This means that polynomial division with $B(Z)$ should not give divergent coefficients. This property of the b_t wavelet is often called **minimum-phase**, for reasons found in my book PVI. Other notable properties of a minimum-phase wavelet are: (1) Poles and zeros are outside the unit circle in the $|Z|$-plane; (2) Equivalent to transmitted waves in a layered medium where reflection coefficients are weaker than unity; (3) Phase at minus Nyquist equals that at plus Nyquist. (Observe this is not true for the pure delay operator $Z = e^{i\omega}$.)

4.4 WILSON'S SPECTRAL FACTORIZATION

Newton's iteration for square roots

$$a_{t+1} \quad = \quad \frac{1}{2}\left(a_t \; + \; \frac{s}{a_t}\right) \tag{4.18}$$

converges quadratically starting from any real initial guess a_0 except zero. When a_0 is negative, Newton's iteration converges to the negative square root.

Quadratic convergence means that the square of the error $a_t - \sqrt{s}$ at one iteration is proportional to the error at the next iteration

$$a_{t+1} - \sqrt{s} \quad \sim \quad (a_t - \sqrt{s})^2 \quad = \quad a_t^2 - 2a_t\sqrt{s} + s \quad > \quad 0 \tag{4.19}$$

so, for example if the error is one significant digit at one iteration, at the next iteration it is two digits, then four, etc. We cannot use equation (1.19) in place of the Newton iteration itself, because it uses the answer $\sqrt{s}$ to get the answer a_{t+1}, and also we need the factor of proportionality. Notice, however, if we take the factor to be $1/(2a_t)$, then $\sqrt{s}$ cancels and equation (1.19) becomes itself the Newton iteration (1.18).

Another interesting feature of the Newton iteration is that all iterations (except possibly the initial guess) are above the ultimate square root. This is obvious from equation (1.19).

We can insert spectral functions in the Newton square-root iteration, for example $s(\omega)$ and $a(\omega)$. Where the first guess a_0 happens to match $\sqrt{s}$, it will match $\sqrt{s}$ at all iterations. Something inspires Wilson to write the Newton iteration as

$$2\,\frac{a_{t+1}}{a_t} \quad = \quad 1 \; + \; \frac{s}{a_t^2} \tag{4.20}$$

and then express the spectrum $S = \bar{A}A$ as a Z-transform (**?**).

$$\frac{\bar{A}_{t+1}(1/Z)}{\bar{A}_t(1/Z)} + \frac{A_{t+1}(Z)}{A_t(Z)} = 1 + \frac{S(Z)}{\bar{A}_t(1/Z)\, A_t(Z)} \tag{4.21}$$

Now we are ready for the algorithm: Compute the right side of (1.21) by polynomial division forwards and backwards and then add 1. Then abandon negative lags and take half of the zero lag. Now you have $A_{t+1}(Z)/A_t(Z)$. Multiply out (convolve) the denominator $A_t(Z)$, and you have the desired result $A_{t+1}(Z)$. Iterate as long as you wish.

We show that $A_{t+1}(Z)$ is minimum phase: Both sides of (1.21) are positive, as noted earlier. Both terms on the right are positive. Since the Newton iteration always overestimates, the 1 dominates the rightmost term. After masking off the negative powers of Z (and half the zero power), the right side of (1.21) adds two wavelets. The 1/2 is wholly real, and hence its real part always dominates the real part of the rightmost term. Thus (after masking negative powers) the wavelet on the right side of (1.21) has a positive real part, so the phase cannot loop about the origin. This wavelet multiplies $A_t(Z)$ to give the final wavelet $A_{t+1}(Z)$ and the product of two minimum-phase wavelets is minimum phase.

Claerbout is interested in factoring the Laplacian

$$S(Z) = -\frac{1}{Z^{100}} - \frac{1}{Z} + 4 - Z - Z^{100} \tag{4.22}$$

by the Wilson method. It seems less confusing than the Kolmogoroff method. Furthermore, it might be better because we can think of upper and lower triangular matrices, instead of recursive filters. The advantage of the triangular matrices is that they embody time-variable filters.

EXERCISES:

1 You hear from three different people that a more isotropic representation of the Laplacian is minus one fifth of

$$\begin{matrix} -1 & -4 & -1 \\ -4 & 20 & -4 \\ -1 & -4 & -1 \end{matrix}$$

What changes need to be made to subroutine `autofac()`?

4.5 HELIX LOW-CUT FILTER

When an image is spoiled by excessive low frequencies, we can cut them out by taking a gradient. Figure 1.8 compares the topography around Stanford University with its

helical derivative and with its southward derivative. Although the helical derivative is also taken in the southward direction, the two derivative images are very different. The helical derivative appears more appropriate for looking at drainage patterns and the ordinary derivative for mountain slopes.

The Poisson equation solver by means of the helix derivative is a convenient isotropic smoothing filter. However, this smoothing operation has no adjustable coefficient to choose the cut-off frequency. A slightly more complicated form allows us to build a filter with a spatial high cut or low cut that has an adjustable cut off frequency. Recall the factorization (1.29) that we did of the low-cut filter in one dimension $\omega^2/(\omega^2+\epsilon^2)$. With this idea, we can use the helix to build a two-dimensional lowcut filter.

```
module helocut {
use lapfac2
use helicon
use polydiv
real,    dimension (:), pointer, private :: aa, bb
integer, dimension (:), pointer, private :: lag
real,    dimension (n1*n2), allocatable  :: tt
#%  _init( eps, na, n1, n2)
real       eps
integer         na, n1, n2
allocate( aa (na), bb (na), lag (na))
call lapfac( 0.0, n1, aa, lag)
call lapfac( eps, n1, bb, lag)
call helicon_init( aa, lag)
call polydiv_init( bb, lag)
write (0,*) "aa", aa
write (0,*) "bb", bb
#% _lop (pp, qq)
integer stat1, stat2
if( adj) {
        stat2 = polydiv_lop( adj, .false., tt, qq)
        stat1 = helicon_lop( adj, .true.,  pp, tt)
        }
else {
        stat1 = helicon_lop( adj, .false., pp, tt)
        stat2 = polydiv_lop( adj, .true.,  tt, qq)
        }
#% _close
deallocate( aa, bb, lag)
}
```

After some experimentation with this filter, I learned that I already had a powerful adjustable parameter in the number of coefficients that are used from the spectral factorization of $-\nabla^2$. I learned that taking not enough lags in the filter caused it not to drop to zero at zero frequency. A simple test of whether it drops to zero at zero frequency is to add all the coefficients in the definition 1.11 and see how close they come to zero. Doing this for the 8 coefficients given, I find the sum is .18 compared to

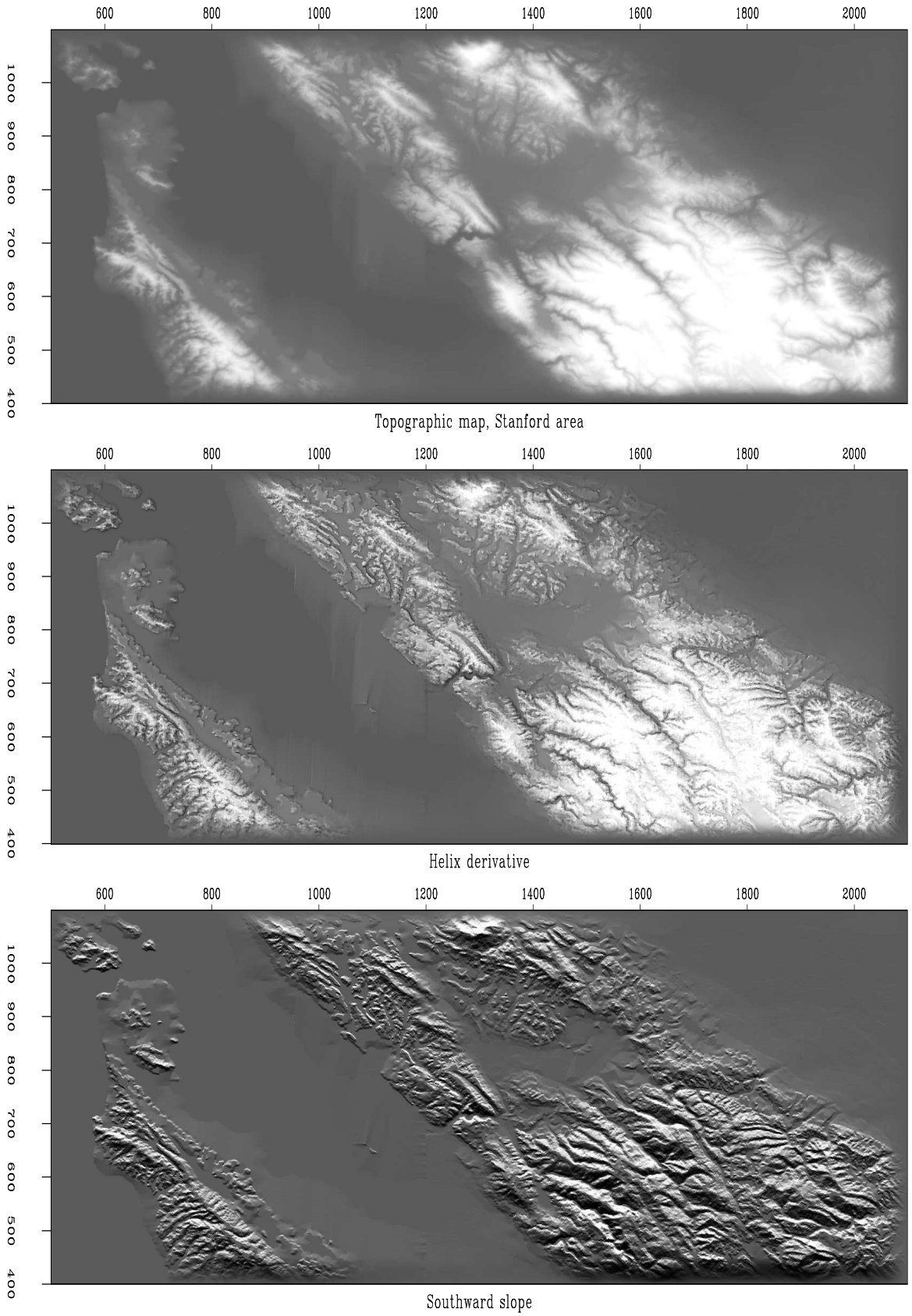


Figure 4.8: Topography, helical derivative, slope south. hlx-helocut90 [ER]

the lead coefficient 1.79, so it seems the "zero" at zero frequency actually only drops 90% of the way to zero. Although I coded a filter response to be the factorization of $-\nabla^2/(\nabla^2+\epsilon^2)$, I found it more desireable to fix ϵ at a large value (thus effectively neglecting the denominator) and make adjustments in the depth of the hole at zero frequency by adjusting the number of filter coefficients in the representation of the helix derivative. In preparing Figure 1.8 I found it most agreeable to use 16 filter coefficients.

Index

autocorrelation, 14
autofac module, 14

causal, 12

exponential of a causal, 12

fft module, 13

helicon operator module, 7
helocut operator module, 17

Kolmogoroff, 14

lapfac module, 11
log spectrum, 14

minimum-phase, 15
module
    `autofac`, spectral factorization, 14
    `fft`, Fourier transform, 13
    `lapfac`, factor 2-D Laplacian, 11

operator
    `helicon`, helical convolution, 7
    `helocut`, helix-lowcut filter, 17

spectral factorization, 14
spectral logarithm, 14