The helical coordinate

Jon Claerbout

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 toward overcoming this difficulty.

Many geophysical map estimation applications appear to be multidimensional, but in reality they are one-dimensional. To see the tip of the iceberg, consider this example: On a 2-dimensional Cartesian mesh, the function $\begin{array}{|c|c|c|c|}\hline 0&0&0&0\\\hline 0&1&1&0\\\hline 0&1&1&0\\\hline 0&0&0&0\\\hline\end{array}$

has the autocorrelation $\begin{array}{|c|c|c|}\hline 1&2&1\\\hline 2&4&2\\\hline 1&2&1\\\hline\end{array}$.

Likewise, on a 1-dimensional cartesian mesh,

the function $\begin{array}{|c|c|c|c|c|c|c|c|}\hline 1&1&0&0&\cdots&0&1&1\\\hline\end{array}$

has the autocorrelation $\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|}\hline 1&2&1&0&\cdots&0&2&4&2&0&\cdots&1&2&1\\\hline\end{array}$.

Observe the numbers in the 1-dimensional world are identical with the numbers in the 2-dimensional world. This correspondence is no accident.

FILTERING ON A HELIX

Figure 1 shows some 2-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.

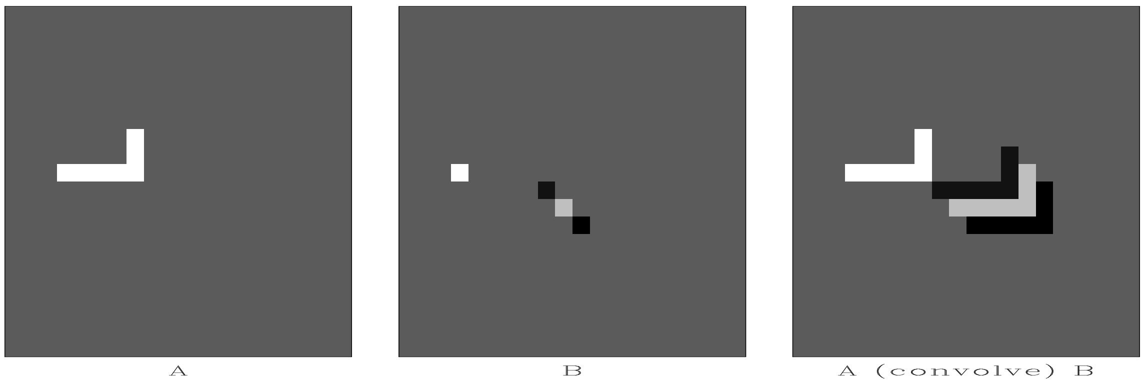

Figure 1: Two-dimensional convolution as performed in one dimension by module `helicon`

A surprising, indeed amazing, fact is that Figure 1 was not computed with a 2-dimensional convolution program. It was computed with a 1-dimensional computer program. It could have been done with anybody's 1-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; and at each location you find an output by crossmultiplying the filter coefficients times the underlying data and summing the products.

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. The concatenated columns make a helix. This arrangement is Fortran's way of storing 2-D arrays in 1-dimensional memory, and it 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 2 shows a helical mesh for 2-D data on a cylinder. Darkened squares depict a 2-D filter shaped like the Laplacian operator $\partial_{xx} + \partial_{yy}$. 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.

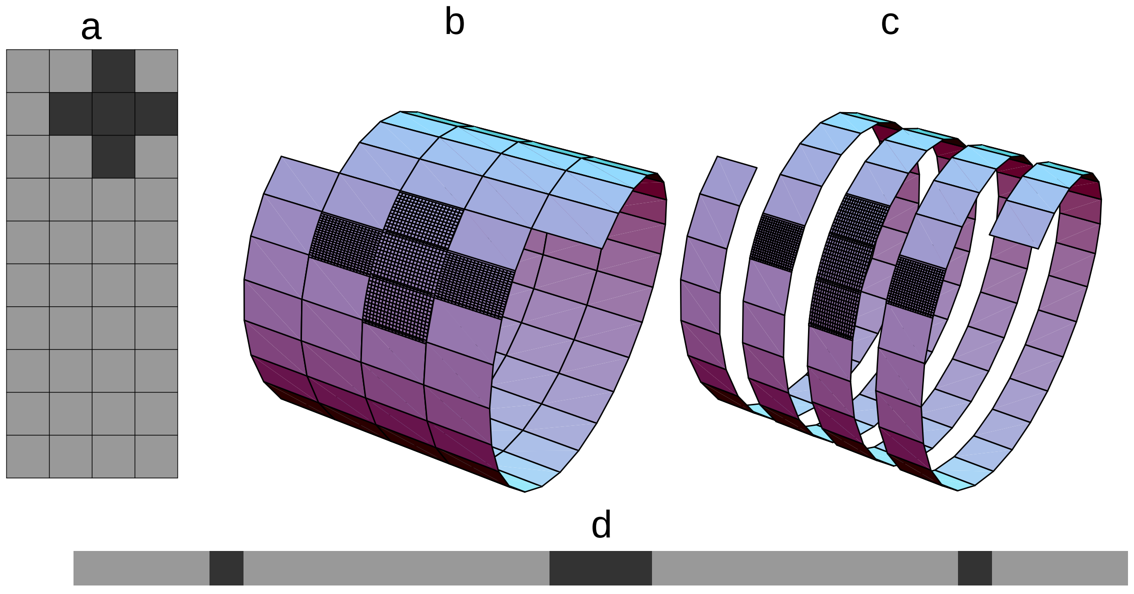

Figure 2: Filtering on a helix. The same filter coefficients overlay the same data values if the 2-D coils are unwound into 1-D strips. (*Mathematica* drawing by Sergey Fomel)

Because 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 1-dimensional strips and convolve the strips 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 1-dimensional way or a 2-dimensional way. I used the 1-dimensional way to compute the obviously 2-dimensional result in Figure 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 with coefficients being 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:

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

Convolution computes y_k from x_k, whereas, deconvolution (also called back substitution) does the reverse. Rearranging (1); we get:

$$x_k \quad = \quad y_k - \sum_{i=1}^{N_a} a_i x_{k-i} \tag{2}$$

where now, we are finding the output x_k from its past outputs x_{k-i} and 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 the coefficients are applied to the past *outputs* instead of the past *inputs*. Needing past outputs is why deconvolution must be done sequentially while convolution can be done in parallel.

Multidimensional deconvolution breakthrough

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. Consequently, 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.

Examples of simple 2-D recursive filters

Let us associate x- and y-derivatives with a finite-difference stencil or template. (For simplicity, take $\Delta x = \Delta y = 1$.)

$$\frac{\partial}{\partial x} \quad = \quad \begin{array}{|r|r|}\hline 1 & -1 \\ \hline \end{array} \tag{3}$$

$$\frac{\partial}{\partial y} \quad = \quad \begin{array}{|r|}\hline 1 \\ \hline -1 \\ \hline \end{array} \tag{4}$$

Convolving a data plane with the stencil (3) forms the x-derivative of the plane. Convolving a data plane with the stencil (4) forms the y-derivative of the plane. On the other hand, *deconvolving* with (3) integrates data along the x-axis for each y. Likewise, deconvolving with (4) integrates data along the y-axis for each x. Next, we look at a fully 2-dimensional operator (like the cross derivative ∂_{xy}).

A nontrivial 2-dimensional convolution stencil is:

$$\begin{array}{|r|r|}\hline 0 & -1/4 \\ \hline 1 & -1/4 \\ \hline -1/4 & -1/4 \\ \hline \end{array} \tag{5}$$

We convolve and deconvolve a data plane with this operator. Although everything is shown on a plane, the actual computations are done in one dimension with equations (1) and (2). Let us manufacture the simple data plane shown on the left in Figure 3. Beginning with a zero-valued plane, we add in a copy of the filter (5) near the top of the frame. Nearby, add another copy with opposite polarity. Finally, add some impulses near the bottom boundary. The second frame in Figure 3 is the result of deconvolution by the filter (5) using the 1-dimensional equation (2). Notice that deconvolution turns the filter 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.

The filtering in Figure 3 ran along a helix from left to right. Figure 4 shows a second filtering running from right to left. Filtering in the reverse direction is the adjoint. After deconvolving both ways, we have accomplished a symmetrical smoothing. The final frame undoes the smoothing to bring us exactly back to where we started. The smoothing was done with two passes of *deconvolution*, and it is undone by two passes of *convolution*. No errors, and no evidence remains at any of the boundaries where we have wrapped and truncated.

Chapter **??** explains the important practical role to be played by a multidimensional operator for which we know the exact inverse. Other than multidimensional Fourier transformation, transforms based on polynomial multiplication and division on a helix are the only known easily invertible linear operators.

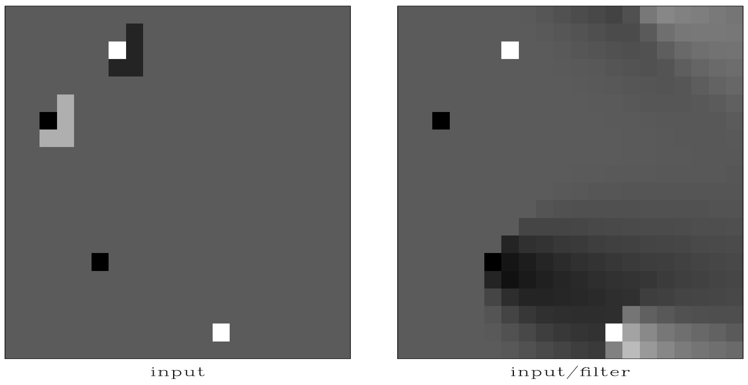

Figure 3: Illustration of 2-D deconvolution. Left is the input. Right is after deconvolution with the filter (5) as preformed by by module `polydiv`

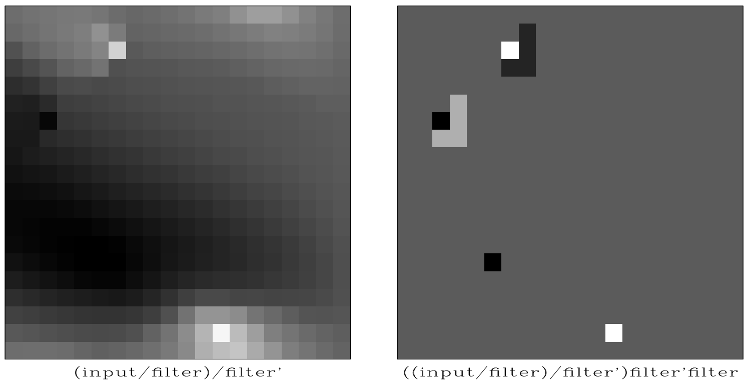

Figure 4: Recursive filtering backward (leftward on the space axis) is done by the *adjoint* of 2-D deconvolution. Here we see that 2-D *deconvolution* compounded with its adjoint is exactly inverted by 2-D *convolution* and its adjoint.

In seismology we often have occasion to steer summation along beams. Such an impulse response is shown in Figure 5.

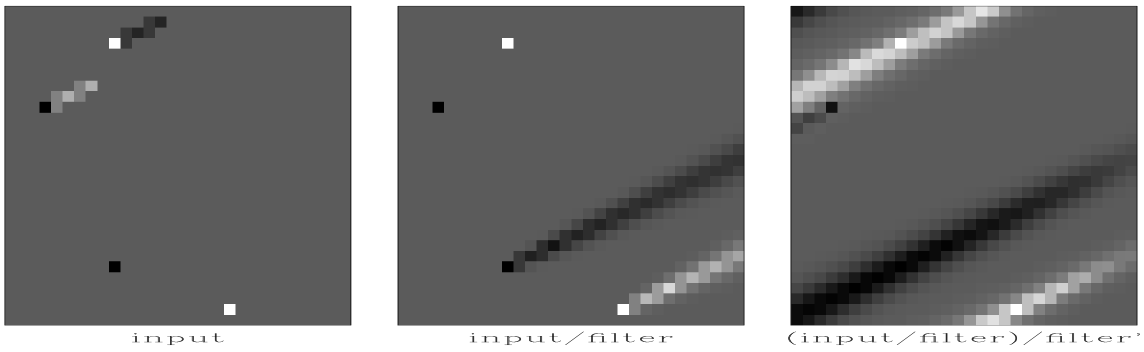

Figure 5: Useful for directional smoothing is a simulated dipping seismic arrival, made by combining a simple low-order 2-D filter with its adjoint.

Of special interest are filters that destroy plane waves. The inverse of such a filter creates plane waves. Such filters are like wave equations. A filter that creates two plane waves is illustrated in figure 6.

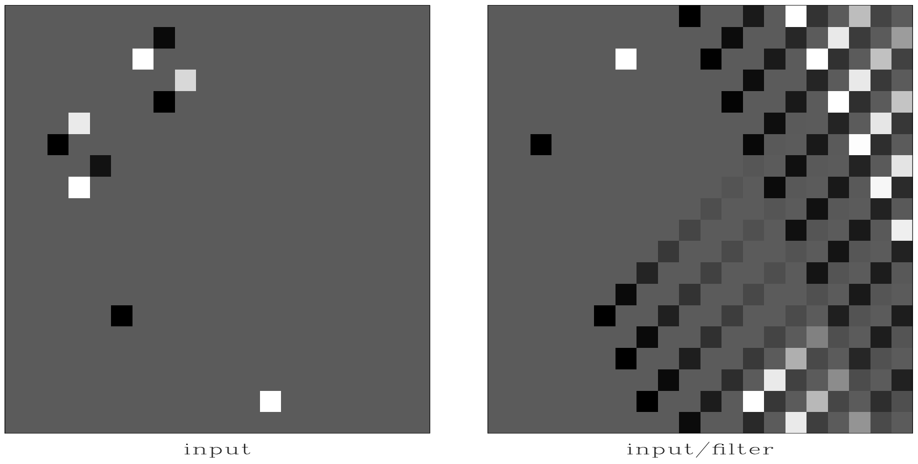

Figure 6: A simple low-order 2-D filter with inverse containing plane waves of two different dips. One is spatially aliased.

Coding multidimensional convolution and deconvolution

Let us unroll the filter helix seen previously in Figure 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. These intervals complete the length of a single 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 those zeros in memory. Whereas, an ordinary convolution program would do time shifting by a code line like `iy=ix+lag`, Module `helicon` 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 `flt[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 can be thought of as the 1-D filter:

$$\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|}\hline 1 & 0 & \cdots & 0 & 1 & -4 & 1 & 0 & \cdots & 0 & 1 \\ \hline \end{array} \quad \rightarrow \text{helical boundaries} \quad \begin{array}{|c|c|c|}\hline & 1 & \\ \hline 1 & -4 & 1 \\ \hline & 1 & \\ \hline \end{array} \tag{6}$$

The first filter coefficient in equation (6) is +1 as implicit to module `helicon`. To apply the Laplacian on a 1,000 × 1,000 mesh requires the filter inputs:

i	lag[i]	flt[i]
0	999	1
1	1000	-4
2	1001	1
3	2000	1

Here, we choose to use "declaration of a type", a modern computer language feature that is absent from Fortran 77. Fortran 77 has the built in complex arithmetic type. In module `helix`, we define a type `filter`, actually, a helix filter. After making this definition, it is used by many programs. The helix filter consists of three vectors, a real valued vector of filter coefficients, an integer valued vector of filter lags, and an optional vector that has logical values "`true`" for output locations that are not computed (either because of boundary conditions or because of missing inputs). The filter vectors are the size of the nonzero filter coefficients (excluding the leading 1.), while the logical vector is long and relates to the data size. The `helix` module allocates and frees memory for a helix filter.

api/c/helix.c

```c
typedef struct sf_helixfilter {
    int     nh;
    float*  flt;
    int*    lag;
    bool*   mis;
    float   h0;
} *sf_filter;
```

For those of you with no C experience, the "`->`" appearing in the helix module denotes a pointer. Fortran 77 has no pointers (or everything is a pointer). The behavior of pointers is somewhat different in each language. In C, pointer behavior is straightforward. In module `helicon` you see the expression `aa->flt[ia]`. It refers to the filter named `aa`. Any filter defined by the `helix` module contains three vectors, one of which is named `flt`. The second component of the `flt` vector in the `aa` filter is referred to as `aa->flt[1]` which in the foregoing example refers to the value 4.0 in the center of the Laplacian operator. For data sets like above with 1,000 points on the 1-axis, this value 4.0 occurs after 1,000 lags, thus `aa->lag[1]=1000`.

Our first convolution operator `tcai1` was limited to one dimension and a particular choice of end conditions. With the helix and C pointers, the operator `helicon` is a *multi-dimensional* filter with considerable flexibility (because of the `mis` vector) to work around boundaries and missing data. The code fragment `aa->lag[ia]` corresponds to `b-1` in `tcai1`.

api/c/helicon.c

```c
void sf_helicon_lop( bool adj, bool add,
                     int nx, int ny, float* xx, float*yy)
/*< linear operator >*/
{
    int ia, iy, ix;

    sf_copy_lop(adj, add, nx, nx, xx, yy);

    for (ia = 0; ia < aa->nh; ia++) {
        for (iy = aa->lag[ia]; iy < nx; iy++) {
            if( aa->mis != NULL && aa->mis[iy]) continue;
            ix = iy - aa->lag[ia];
            if(adj) {
                xx[ix] += yy[iy] * aa->flt[ia];
            } else {
                yy[iy] += xx[ix] * aa->flt[ia];
            }
        }
    }
}
```

Operator `helicon` did the convolution job for Figure 1. As with `tcai1`, the adjoint of helix filtering is reversing the screw—filtering backwards.

The companion to convolution is deconvolution. The module `polydiv` is essentially the same as `polydiv1` but here it was coded using our new `filter` type in module `helix` which simplifies our many future uses of convolution and deconvolution. Although convolution allows us to work around missing input values, deconvolution does not (any input affects all subsequent outputs), so `polydiv` never references `aa->mis[ia]`.

KOLMOGOROFF SPECTRAL FACTORIZATION

Spectral factorization addresses a deep mathematical problem not solved by mathematicians until 1939. Given any spectrum $|F(\omega)|$, find a causal time function $f(t)$ with this spectrum. A causal time function is one that vanishes at negative time $t < 0$. We mix spectral factorization with the helix idea to find many applications in geophysical image estimation.

api/c/polydiv.c

```
void sf_polydiv_lop( bool adj, bool add,
                     int nx, int ny, float* xx, float*yy)
/*< linear operator >*/
{
    int ia, iy, ix;

    sf_adjnull( adj, add, nx, ny, xx, yy);

    for (ix=0; ix < nx; ix++) tt[ix] = 0.;

    if (adj) {
        for (ix = nx-1; ix >= 0; ix--) {
            tt[ix] = yy[ix];
            for (ia = 0; ia < aa->nh; ia++) {
                iy = ix + aa->lag[ia];
                if( iy >= ny) continue;
                tt[ix] -= aa->flt[ia] * tt[iy];
            }
        }
        for (ix=0; ix < nx; ix++) xx[ix] += tt[ix];
    } else {
        for (iy = 0; iy < ny; iy++) {
            tt[iy] = xx[iy];
            for (ia = 0; ia < aa->nh; ia++) {
                ix = iy - aa->lag[ia];
                if( ix < 0) continue;
                tt[iy] -= aa->flt[ia] * tt[ix];
            }
        }
        for (iy=0; iy < ny; iy++) yy[iy] += tt[iy];
    }
}
```

The most abstract method of spectral factorization is of the Russian mathematician A.N.Kolmogoroff. I include it here, because it is by far the fastest, so much so that giant problems become practical, such as the solar physics example coming up.

Given that $C(\omega)$ Fourier transforms to a causal function of time, it is next proven that e^C Fourier transforms to a causal function of time. Its filter inverse is e^{-C}. Grab yourself a cup of coffee and hide yourself away in a quiet place while you focus on the proof in the next paragraph.

A causal function c_τ vanishes at negative τ. Its Z transform $C(Z) = c_0 + c_1 Z + c_2 Z^2 + c_3 Z^3 + \cdots$, with $Z = e^{i\omega\Delta t}$ is really a Fourier sum. Its square $C(Z)^2$ convolves a causal with itself so it is causal. Each power of $C(Z)$ is causal, therefore, $e^C = 1 + C + C^2/2 + \cdots$, a sum of causals, is causal. The time-domain coefficients for e^C could be computed putting polynomials into power series or computed faster with Fourier transforms (by understanding $C(Z = e^{i\omega\Delta t})$ as an FT.) By the same reasoning, the wavelet e^C has inverse e^{-C} which is also causal. A causal with a causal inverse is said to be "minimum phase." The filter $1 - Z/2$ with inverse $1 + Z/2 + Z^2/4 + \cdots$ is minimum phase because both are causal, and they multiply to make the impulse "1", so are mutually inverse. The delay filter Z^5 has the noncausal inverse Z^{-5} which is not causal (output before input).

The next paragraph defines "Kolmogoroff spectral factorization." It arises in applications where one begins with an energy spectrum $|r|^2$ and factors it into an $re^{i\phi}$ times its conjugate. The inverse Fourier transform of that $re^{i\phi}$ is causal.

Relate amplitude $r = r(\omega)$ and phase $\phi = \phi(\omega)$ to a causal time function c_τ.

$$|r|e^{i\phi} \;=\; e^{\ln|r|}e^{i\phi} \;=\; e^{\ln|r|+i\phi} \;=\; e^{c_0+c_1Z+c_2Z^2+c_3Z^3+\cdots} \;=\; e^{\sum_{\tau=0} c_\tau Z^\tau} \tag{7}$$

Given a spectrum $r(\omega)$, we find a filter with that spectrum. Because $r(\omega)$ is a real even function of ω, so is its logarithm. Let the inverse Fourier transform of $\ln|r(\omega)|$ be u_τ, where u_τ is a real even function of time. Imagine a real odd function of time v_τ.

$$|r|e^{i\phi} \;=\; e^{\ln|r|+i\phi} \;=\; e^{\sum_\tau (u_\tau+v_\tau)Z^\tau} \tag{8}$$

The phase $\phi(\omega)$ transforms to v_τ. We can assert causality by choosing v_τ so that $u_\tau + v_\tau = 0$ $(= c_\tau)$ for all negative τ. This choice defines v_τ at negative τ. Since v_τ is odd, it is also known at positive lags. More simply, v_τ is created when u_τ is multiplied by a step function of size 2. This causal exponent $(c_0, c_1, \cdots)$ creates a causal filter $|r|e^{i\phi}$ with the specified spectrum $r(\omega)$.

We easily manufacture an inverse filter by changing the polarity of the c_τ. This filter is also causal by the same reasoning. Thus, these filters are causal with a causal inverse. Such filters are commonly called "minimum phase."

Spectral factorization arises in a variety of contexts. Here is one: Rain drops showering on a tin roof create for you a signal with a spectrum you can compute, but what would be the sound of a single drop, the wavelet of a single drop? Spectral factorization gives the answer. Divide this wavelet out from the data to get a record of impulses, one for each rain drop (theoretically!). Similarly, the boiling surface of the sun is coming soon.

Constant Q medium

From the absorption law of a material, spectral factorization yields its impulse response. The most basic absorption law is the *constant* Q model. According to it, for a downgoing wave, the absorption is proportional to the frequency ω, proportional to time in the medium z/v, and inversely proportional to the "quality" Q of the medium. Altogether, the spectrum of a wave passing through a thickness z is changed by the factor $e^{-|\omega|\tau} = e^{-|\omega|(z/v)/Q}$. This frequency function is plotted in the top line of Figure 7.

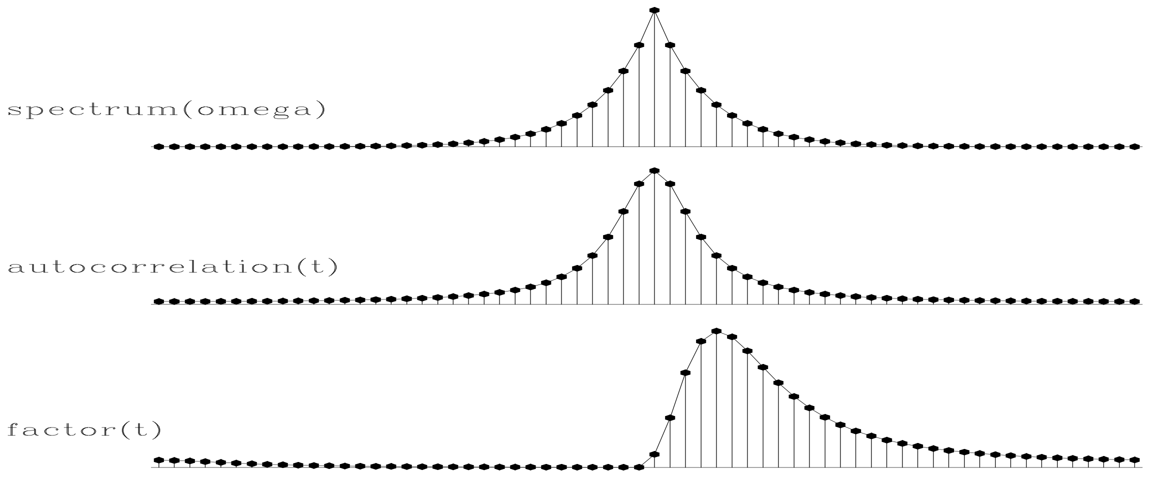

Figure 7: Autocorrelate the bottom signal to get the middle, then Fourier transform it to get the top. Spectral factorization works the other way, from top to bottom.

The middle function in Figure 7 is the autocorrelation giving on top the spectrum $e^{-|\omega|\tau}$. The third function is the factorization. An impulse entering the medium comes out with this shape. There is no physics in this analysis, only mathematics that assumes the broadened pulse is causal with an abrupt arrival. The short wavelengths are concentrated near the sharp corner, while the long wavelengths are spread throughout. A physical system could cause the pulse to spread further (effectively by an additional all-pass filter), but physics cannot make it more compact.

All distances from the source see the same shape, but stretched in proportion to distance. The apparent Q is the traveltime to the source divided by the width of the pulse.

Causality in two dimensions

Our foundations, the basic convolution-deconvolution pair (1) and (2) are applicable only to filters with all coefficients *after* zero lag. Filters of physical interest generally concentrate coefficients near zero lag. Requiring causality in 1-D and concentration in 2-D leads to shapes such as these:

$$\begin{array}{ccc} h & c & 0 \\ p & d & 0 \\ q & e & \mathbf{1} \\ s & f & a \\ u & g & b \end{array} \quad = \quad \begin{array}{ccc} h & c & \cdot \\ p & d & \cdot \\ q & e & \cdot \\ s & f & a \\ u & g & b \end{array} \quad + \quad \begin{array}{ccc} \cdot & \cdot & 0 \\ \cdot & \cdot & 0 \\ \cdot & \cdot & \mathbf{1} \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{array} \tag{9}$$

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

where $a, b, c, ..., u$ are coefficients we find by least squares.

The complete story is rich in mathematics and in concepts; but to sum up, filters fall into two categories according to the numerical values of their coefficients. There are filters for which equations (1) and (2) work as desired and expected. These filters are called "minimum phase." There are also filters for which equation (2) is a disaster numerically, the feedback process diverging to infinity.

Divergent cases correspond to physical processes that are not simply described by initial conditions but require also reflective boundary conditions, so information flows backward, i.e., anticausally. Equation (2) only allows for initial conditions.

I oversimplify by trying to collapse an entire book *FGDP* (Fundamentals of Geophysical Data Processing) into a few sentences by saying here that for any fixed 1-D spectrum there exist many filters. Of these, only one has stable polynomial division. That filter has its energy compacted as soon as possible after the "1.0" at zero lag.

Causality in three dimensions

The top plane in Figure 8 is the 2-D filter seen in equation (9). Geometrically, the 3-dimensional generalization of a helix, Figure 8 shows a causal filter in three dimensions. Think of the little cubes as packed with the string of the causal 1-D function. Under the "1" is packed with string, but none above it. Behind the "1" is packed with string, but none in front of it. The top plane can be visualized as the area around the end of the 1-D string. Above the top plane are zero-valued anticausal filter coefficients.

This 3-D cube is like the usual Fortran packing of a 3-D array with one confusing difference. The starting location where the "1" is located is not at the Fortran (1,1,1) location. Details of indexing are essential, but complicated, and found near the end of this chapter.

The "1" that defines the end of the 1-dimensional filter becomes in 3-D a point of central symmetry. Every point inside a 3-D filter has a mate opposite the "1" that is outside the filter. Altogether they fill the whole space leaving no holes. From this you may deduce that the "1" must lie on the side of a face as shown in Figure 8. It cannot lie on the corner of a cube. It cannot be at the Fortran of `f(1,1,1)`. If it were there, the filter points inside with their mirror points outside would not full the entire space. It could not represent all possible 3-D autocorrelation functions.

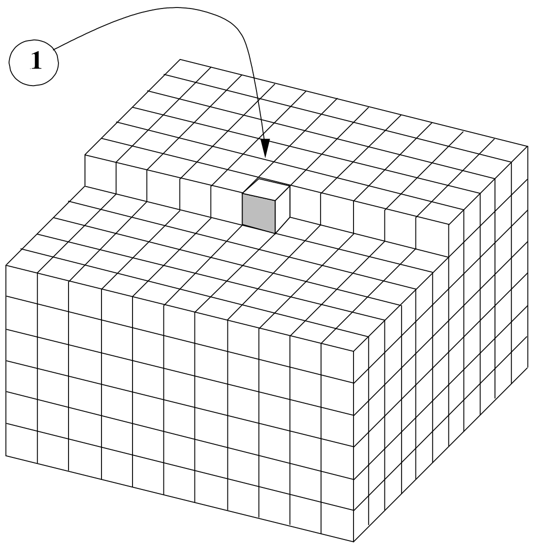

Figure 8: A 3-D causal filter at the starting end of a 3-D helix.

Blind deconvolution and the solar cube

An area of applications that leads directly to spectral factorization is “blind deconvolution.” Here, we begin with a signal. We form its spectrum and factor it. We could simply inspect the filter and interpret it, or we might deconvolve it out from the original data. This topic deserves a fuller exposition, say for example as defined in some of my earlier books. Here, we inspect a novel example that incorporates the helix.

Solar physicists have learned how to measure the seismic field of the sun surface. It is chaotic. If you created an impulsive explosion on the surface of the sun, what would the response be? James Rickett and I applied the helix idea along with Kolmogoroff spectral factorization to find the impulse response of the sun. Figure 9 shows a raw data cube and the derived impulse response. The sun is huge, so the distance scale is in megameters (Mm). The United States is 5-Mm wide. Vertical motion of the sun is measured with a videocamera-like device that measures vertical motion by an optical doppler shift. From an acoustic/seismic point of view, the surface of the sun is a very noisy place. The figure shows time in kiloseconds (Ks). We see roughly 15 cycles in 5 Ks which is 1 cycle in roughly 333 seconds. Thus, the sun seems to oscillate vertically with roughly a 5-minute period. The top plane of the raw data in Figure 9 (left panel) happens to have a sun spot in the center. The data analysis here is not affected by the sun spot, so please ignore it.

The first step of the data processing is to transform the raw data to its spectrum. With the helix assumption, computing the spectrum is virtually the same thing in 1-D space as in 3-D space. The resulting spectrum was passed to Kolmogoroff spectral factorization code, a 1-D code. The resulting impulse response is on the right side of Figure 9. The plane we see on the right top is not lag time $\tau = 0$; it is lag time $\tau = 3$ Ks. It shows circular rings, as ripples on a pond. Later lag times (not shown) would be the larger circles of expanding waves. The front and side planes show tent-like shapes.

The slope of the tent gives the (inverse) velocity of the wave (as seen on the surface of the sun). The horizontal velocity we see on the sun surface turns out (by Snell’s law) to be the same as that at the bottom of the ray. On the front face at early times we see the low-

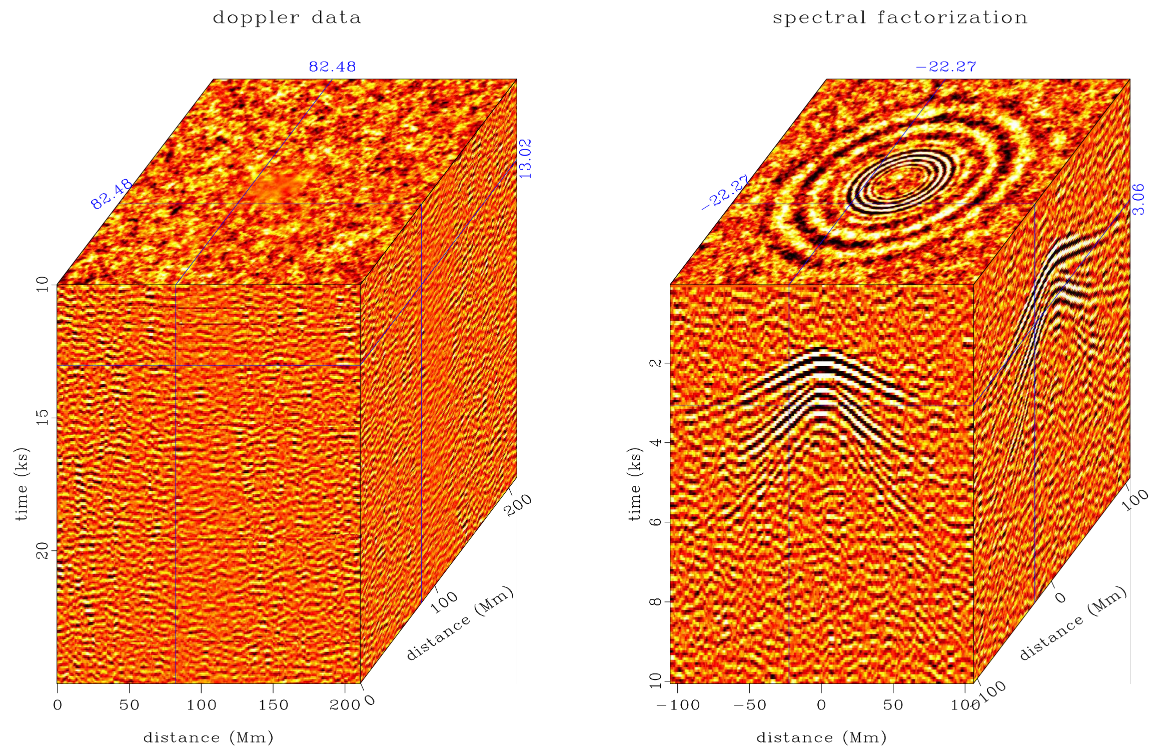

Figure 9: Raw seismic data on the sun (left). Impulse response of the sun (right) derived by Helix-Kolmogoroff spectral factorization.

velocity (steep) wavefronts and at later times we see the faster waves. Later arrivals reach more deeply into the sun except when they are late because they are “multiple reflections,” diving waves that bend back upward reaching the surface, then bouncing down again.

Multiple reflections from the sun surface are seen on the front face of the cube with the same slope, but double the time and distance. On the top face, the first multiple reflection is the inner ring with the shorter wavelengths.

Very close to $t = 0$ see horizontal waveforms extending only a short distance from the origin. These are electromagnetic waves of essentially infinite velocity.

FACTORED LAPLACIAN == HELIX DERIVATIVE

I had learned spectral factorization as a method for single seismograms. After I learned it, every time I saw a positive function I would wonder if it made sense to factor it. When total field magnetometers were invented, I found it as a way to deduce vertical and horizontal **magnetic**components. A few pages back, you saw how to use factorization to deduce the waveform passing through an absorptive medium. Then, we saw how the notion of “impulse response” applies not only to signals, but allows use of random noise on the sun to deduce the 3-D impulse response there. But the most useful application of spectral factorization so far is what comes next, factoring the Laplace operator, $-\nabla^2$. Its Fourier transform $-((ik_x)^2 + (ik_y)^2) \geq 0$ is positive, so it is a spectrum. The useful tool we uncover I dub the "helix derivative."

The signal:

$$\mathbf{r} \quad = \quad -\nabla^2 \quad = \quad \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|}\hline -1 & 0 & \cdots & 0 & -1 & 4 & -1 & 0 & \cdots & 0 & -1 \\ \hline \end{array} \tag{10}$$

is an autocorrelation function, because it is symmetrical about the "4," and the Fourier transform of $-\nabla^2$ is $-((ik_x)^2 + (ik_y)^2) \geq 0$, which is positive for all frequencies (k_x, k_y). Kolmogoroff spectral-factorization gives this wavelet $\mathbf{h}$:

$$\mathbf{h} \quad = \quad \begin{array}{|c|c|c|c|c|c|c|c|c|c|}\hline 1.791 & -.651 & -.044 & -.024 & \cdots & \cdots & -.044 & -.087 & -.200 & -.558 \\ \hline \end{array} \tag{11}$$

In other words, the autocorrelation of (11) is (10). This fact is not obvious from the numbers, because the computation requires a little work, but dropping all the small numbers allows you a rough check.

In this book section only, I use abnormal notation for bold letters. Here $\mathbf{h}$, $\mathbf{r}$ are signals, while $\mathbf{H}$ and $\mathbf{R}$ are images, being neither matrices or vectors. Recall from Chapter **??** that a filter is a signal packed into a matrix to make a filter operator.

Let the time reversed version of $\mathbf{h}$ be denoted $\mathbf{h}^{\mathrm{T}}$. This notation is consistent with an idea from Chapter **??** that the adjoint of a filter matrix is another filter matrix with a reversed filter. In engineering, it is conventional to use the asterisk symbol "$*$" to denote convolution. Thus, the idea that the autocorrelation of a signal $\mathbf{h}$ is a convolution of the signal $\mathbf{h}$ with its time reverse (adjoint) can be written as $\mathbf{h}^{\mathrm{T}} * \mathbf{h} = \mathbf{h} * \mathbf{h}^{\mathrm{T}} = \mathbf{r}$.

Wind the signal $\mathbf{r}$ around a vertical-axis helix to see its 2-dimensional shape $\mathbf{R}$:

$$\mathbf{r} \quad \rightarrow \text{helical boundaries} \quad \begin{array}{|r|r|r|}\hline & -1 & \\ \hline -1 & 4 & -1 \\ \hline & -1 & \\ \hline \end{array} \quad = \quad \mathbf{R} \tag{12}$$

This 2-D image (which can be packed into a filter operator) is the negative of the finite-difference representation of the Laplacian operator, generally denoted $\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2}$. Now for the magic: Wind the signal $\mathbf{h}$ around the same helix to see its 2-dimensional shape $\mathbf{H}$

$$\mathbf{H} \quad = \quad \begin{array}{|c|c|c|c|c|c|c|c|c|}\hline & & & & 1.791 & -.651 & -.044 & -.024 & \cdots \\ \hline \cdots & -.044 & -.087 & -.200 & -.558 & & & & \\ \hline \end{array} \tag{13}$$

In the representation (13), we see the coefficients diminishing rapidly away from maximum value 1.791. My claim is that the 2-dimensional autocorrelation of (13) is (12). You verified this idea previously when the numbers were all ones. You can check it again in a few moments if you drop the small values, say 0.2 and smaller.

Physics on a helix can be viewed through the eyes of matrices and numerical analysis. This presentation is not easy, because the matrices are so huge. 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 minus the Laplacian operator $-(\partial_{xx} + \partial_{yy})$ into a matrix. For a 4×3 plane, that matrix is shown in equation (14).

$$
-\nabla^2 \quad = \quad \left[\begin{array}{rrrr|rrrr|rrrr}
4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
-1 & 4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & -1 & 4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & -1 & 4 & h & \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdot \\
\hline
-1 & \cdot & \cdot & h & 4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot & \cdot \\
\cdot & -1 & \cdot & \cdot & -1 & 4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot \\
\cdot & \cdot & -1 & \cdot & \cdot & -1 & 4 & -1 & \cdot & \cdot & -1 & \cdot \\
\cdot & \cdot & \cdot & -1 & \cdot & \cdot & -1 & 4 & h & \cdot & \cdot & -1 \\
\hline
\cdot & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & h & 4 & -1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & -1 & 4 & -1 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & -1 & 4 & -1 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & -1 & 4
\end{array}\right] \tag{14}
$$

The 2-dimensional matrix of coefficients for the Laplacian operator is shown in (14), 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 4×3 array.) Notice that the partitioning becomes transparent for the helix, $h = -1$. With the partitioning thus invisible, the matrix simply represents 1-dimensional convolution and we have an alternative analytical approach, 1-dimensional Fourier transform. We often need to solve sets of simultaneous equations with a matrix similar to (14). The method we use is triangular factorization.

Although the autocorrelation $\mathbf{r}$ has mostly zero values, the factored autocorrelation $\mathbf{a}$ has a great number of nonzero terms. Fortunately, the coefficients seem to be shrinking rapidly towards a gap in the middle, so truncation (of those middle coefficients) seems reasonable. I wish I could show you a larger matrix, but all I can do is to pack the signal $\mathbf{a}$ into shifted columns of a lower triangular matrix $\mathbf{A}$ like this:

$$
\mathbf{A} \;=\; \left[\begin{array}{rrrrrrrrrrrr}
1.8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
-.6 & 1.8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\ddots & -.6 & 1.8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
-.2 & \ddots & -.6 & 1.8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
-.6 & -.2 & \ddots & -.6 & 1.8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & -.6 & -.2 & \ddots & -.6 & 1.8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & -.6 & -.2 & \ddots & -.6 & 1.8 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & -.6 & -.2 & \ddots & -.6 & 1.8 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & -.6 & -.2 & \ddots & -.6 & 1.8 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & -.6 & -.2 & \ddots & -.6 & 1.8 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -.6 & -.2 & \ddots & -.6 & 1.8 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -.6 & -.2 & \ddots & -.6 & 1.8
\end{array}\right] \tag{15}
$$

If you allow me some truncation approximations, I now claim that the Laplacian represented by the matrix in equation (14) is factored into two parts $-\nabla^2 = \mathbf{A}^{\mathrm{T}}\mathbf{A}$, which are upper and lower triangular matrices whose product forms the autocorrelation seen in equation (14). Recall that triangular matrices allow quick solutions of simultaneous equations by backsubstitution, which is what we are doing with our deconvolution program.

Spectral factorization produces not merely a causal wavelet with the required autocorrelation. It produces one that is stable in deconvolution. Using $\mathbf{H}$ in 1-dimensional polynomial division, we can solve many formerly difficult problems very rapidly. Consider the Laplace equation with sources (Poisson's equation). Polynomial division and its reverse (adjoint) gives us $\mathbf{p} = (\mathbf{q}/\mathbf{H})/\mathbf{H}^{\mathrm{T}}$, which means we have solved $\nabla^2 \mathbf{p} = -\mathbf{q}$ by using polynomial division on a helix. Using the 7 coefficients shown, the cost is 14 multiplications (because we need to run both ways) per mesh point. An example is shown in Figure 10.

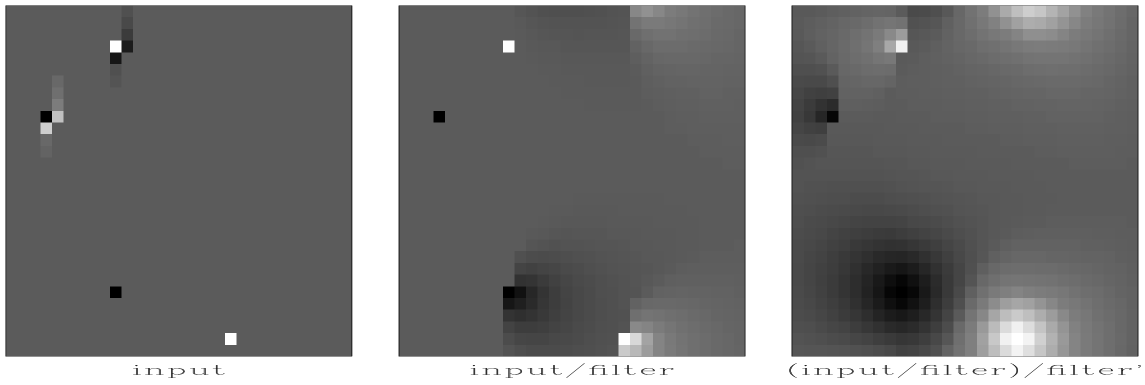

Figure 10: Deconvolution by a filter with autocorrelation being the 2-dimensional Laplacian operator. Amounts to solving the Poisson equation. Left is $\mathbf{q}$; Middle is $\mathbf{q}/\mathbf{H}$; Right is $(\mathbf{q}/\mathbf{H})/\mathbf{H}^{\mathrm{T}}$.

Figure **??** contains both the helix derivative and its inverse. Contrast those filters to the x- or y-derivatives (doublets) and their inverses (axis-parallel lines in the (x, y)-plane). Simple derivatives are highly directional, whereas, the helix derivative is only slightly directional achieving its meagre directionality entirely from its phase spectrum.

HELIX LOW-CUT FILTER

Because the autocorrelation of $\mathbf{H}$ is $\mathbf{H}^{\mathrm{T}} * \mathbf{H} = \mathbf{R} = -\nabla^2$ is a second derivative, the operator $\mathbf{H}$ must be something like a first derivative. As a geophysicist, I found it natural to compare the operator $\frac{\partial}{\partial y}$ with $\mathbf{H}$ by applying the helix derivative $\mathbf{H}$ to a local topographic map. The result shown in Figure 11 is that $\mathbf{H}$ enhances drainage patterns whereas $\frac{\partial}{\partial y}$ enhances mountain ridges.

The operator $\mathbf{H}$ has curious similarities and differences with the familiar gradient and divergence operators. In 2-dimensional physical space, the gradient maps one field to *two* fields (north slope and east slope). The factorization of $-\nabla^2$ with the helix gives us the operator $\mathbf{H}$ that maps one field to *one* field. Being a one-to-one transformation (unlike gradient and divergence), the operator $\mathbf{H}$ is potentially invertible by deconvolution (recursive filtering).

I have chosen the name "helix derivative" or "helical derivative" for the operator $\mathbf{H}$. A

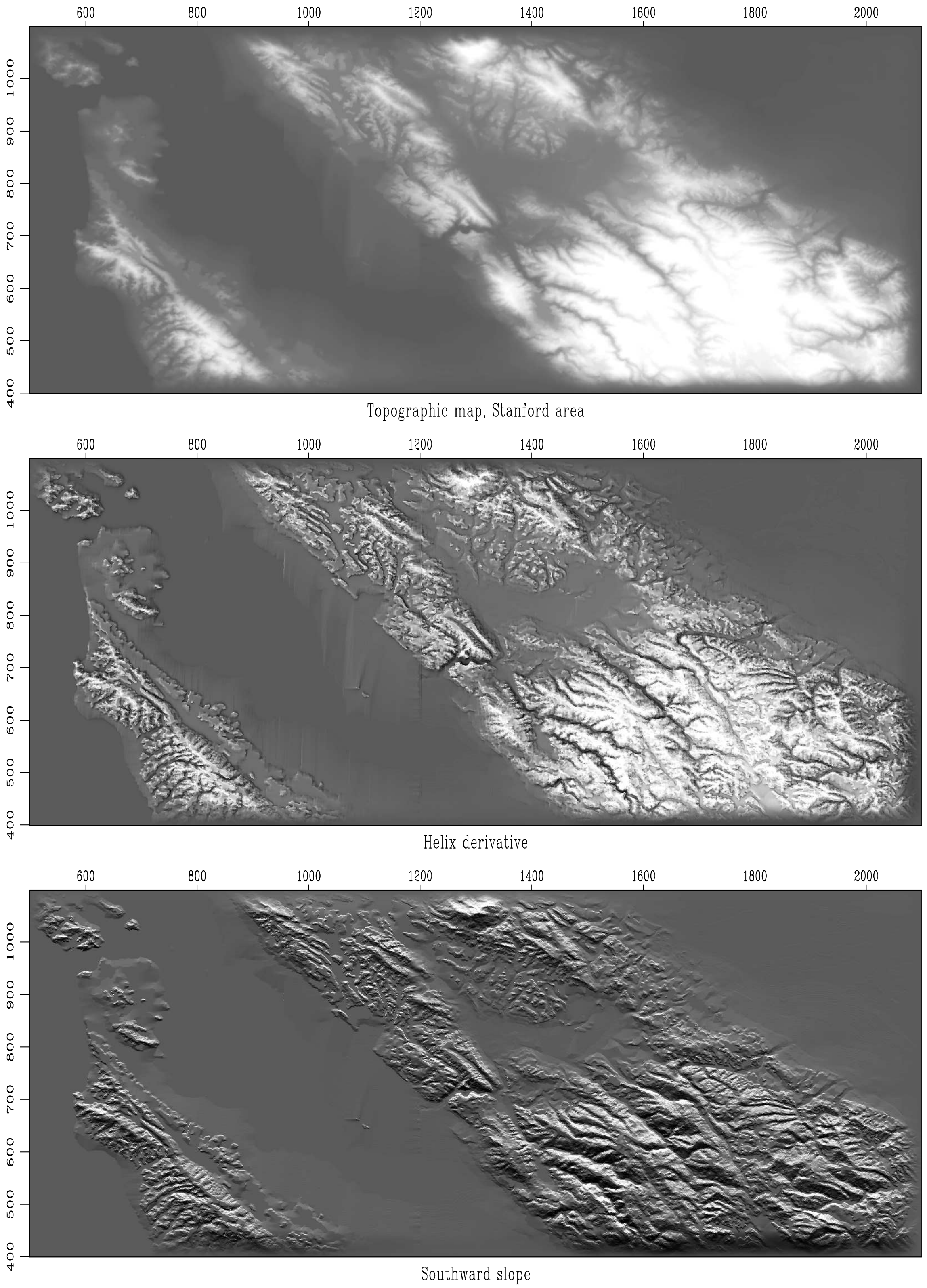

Figure 11: Topography, helical derivative, slope south.

flag pole has a narrow shadow behind it. The helix integral (middle frame of Figure **??**) and the helix derivative (left frame) show shadows with an angular bandwidth approaching 180°.

Our construction makes $\mathbf{H}$ have the energy spectrum $k_x^2 + k_y^2$, so the magnitude of the Fourier transform is $\sqrt{k_x^2 + k_y^2}$. It is a cone centered at the origin with there the value zero. By contrast, the components of the ordinary gradient have amplitude responses $|k_x|$ and $|k_y|$ that are lines of zero across the (k_x, k_y)-plane.

The rotationally invariant cone in the Fourier domain contrasts sharply with the nonrotationally invariant helix derivative in (x, y)-space. The difference must arise from the phase spectrum. The factorization (13) is nonunique because causality associated with the helix mapping can be defined along either x- or y-axes; thus the operator (13) can be rotated or reflected.

In practice, we often require an isotropic filter. Such a filter is a function of $k_r = \sqrt{k_x^2 + k_y^2}$. It could be represented as a sum of helix derivatives to integer powers.

If you want to see some tracks on the side of a hill, you want to subtract the hill and see only the tracks. Usually, however, you do not have a very good model for the hill. As an expedient, you could apply a low-cut filter to remove all slowly variable functions of altitude. In Chapter **??** we found the Sea of Galilee in Figure **??** to be too smooth for viewing pleasure, so we made the roughened versions in Figure **??**, a 1-dimensional filter that we could apply over the x-axis or the y-axis. In Fourier space, such a filter has a response function of k_x or a function of k_y. The isotropy of physical space tells us it would be more logical to design a filter that is a function of $k_x^2 + k_y^2$. In Figure 11 we saw that the helix derivative $\mathbf{H}$ does a nice job. The Fourier magnitude of its impulse response is $k_r = \sqrt{k_x^2 + k_y^2}$. There is a little anisotropy connected with phase (which way should we wind the helix, on x or y?), but it is not nearly so severe as that of either component of the gradient, the two components having wholly different spectra, amplitude $|k_x|$ or $|k_y|$.

Improving low-frequency behavior

It is nice having the 2-D helix derivative, but we can imagine even nicer 2-D low-cut filters. In 1-D, we designed a filter with an adjustable parameter, a cutoff frequency. In 1-D, we compounded a first derivative (which destroys low frequencies) with a leaky integration (which undoes the derivative at all other frequencies). The analogous filter in 2-D would be $-\nabla^2/(-\nabla^2 + k_0^2)$, which would first be expressed as a finite difference $(-Z^{-1} + 2.00 - Z)/(-Z^{-1} + 2.01 - Z)$ and then factored as we did the helix derivative.

We can visualize a plot of the magnitude of the 2-D Fourier transform of the filter equation (13). It is a 2-D function of k_x and k_y and it should resemble $k_r = \sqrt{k_x^2 + k_y^2}$. The point of the cone $k_r = \sqrt{k_x^2 + k_y^2}$ becomes rounded by the filter truncation, so k_r does not reach zero at the origin of the (k_x, k_y)-plane. We can force it to vanish at zero frequency by subtracting .183 from the lead coefficient 1.791. I did not do that subtraction in Figure 12, which explains the whiteness in the middle of the lake. I gave up on playing with both k_0 and filter length; and now, merely play with the sum of the filter coefficients.

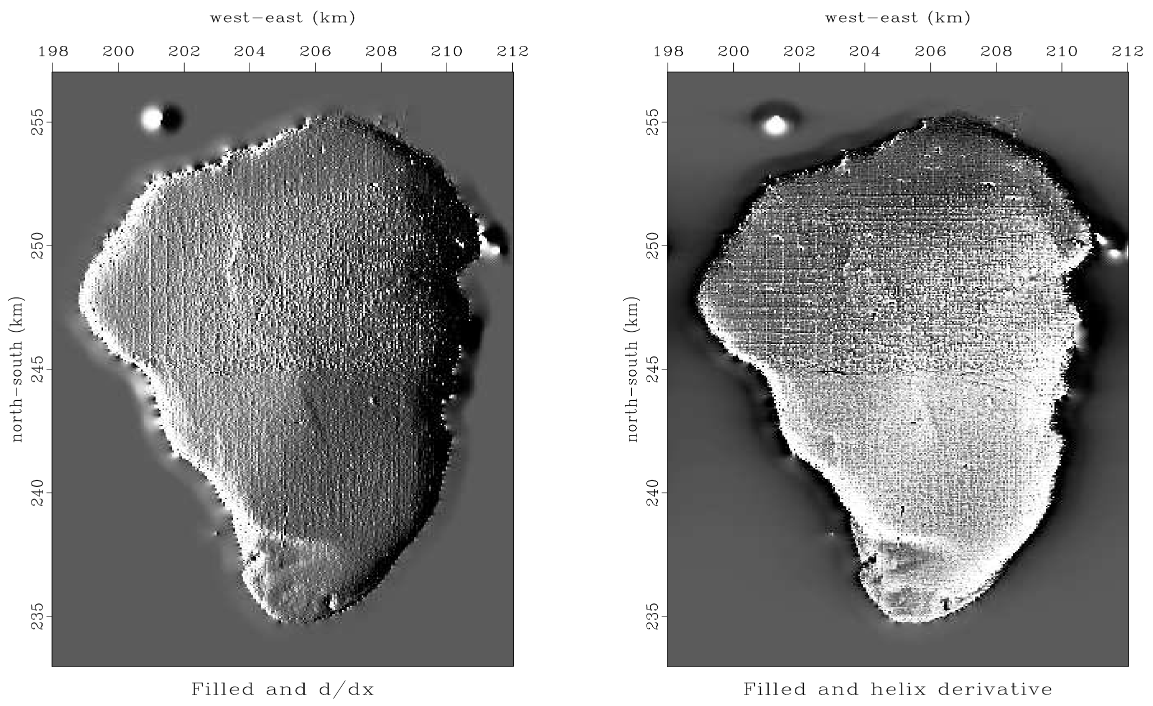

Figure 12: Galilee roughened by gradient and by helical derivative.

Filtering mammograms

I prepared a half dozen medical X-rays like Figure 13. The doctor brought her young son to my office one evening to evaluate the results. In a dark room, I would show the original X-ray on a big screen and then suddenly switch to the helix derivative. Every time I did this, her son would exclaim "Wow!" The doctor was not so easily impressed, however. She was not accustomed to the unfamiliar image. Fundamentally, the helix derivative applied to her data does compress the dynamic range making weaker features more readily discernible. We were sure of this from theory and various geophysical examples. The subjective problem was her unfamiliarity with our display. I found that I could always spot anomalies more quickly on the filtered display, but then I would feel more comfortable when I would discover those same anomalies also present (though less evident) in the original data. Retrospectively, I felt the doctor would likely have been equally impressed had I used a spatial low-cut filter instead of the helix derivative. This simpler filter would have left the details of her image unchanged (above the cutoff frequency), altering only the low frequencies, thereby allowing me to increase the gain.

First, I had a problem preparing Figure 13. It shows the application of the helix derivative to a medical X-ray. The problem was that the original X-ray was all positive values of brightness, so there was a massive amount of spatial low frequency present. Obviously, an x-derivative or a y-derivative would eliminate the low frequency, but the helix derivative did not. This unpleasant surprise arises because the filter in equation (13) was truncated after a finite number of terms. Adding up the terms actually displayed in equation (13), the sum comes to .183, whereas, theoretically the sum of all the terms should be zero. From the

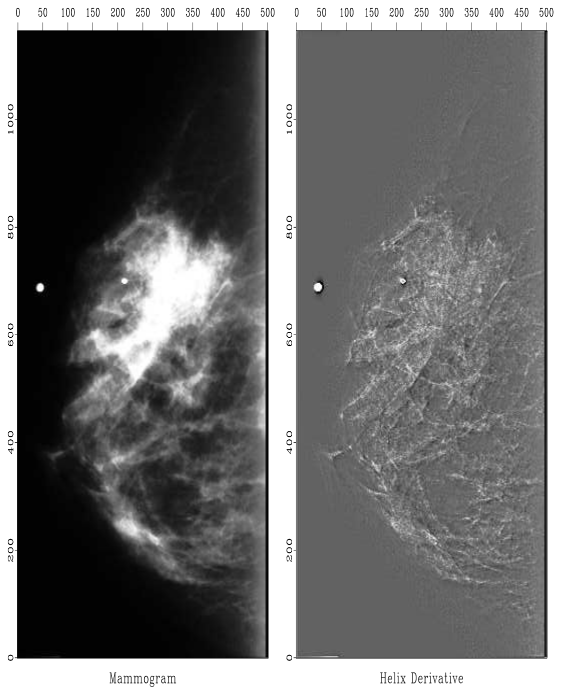

Figure 13: Mammogram (medical X-ray). The cancer is the “spoked wheel.” (I apologize for the inability of paper publishing technology to exhibit a clear grey image.) The tiny white circles are metal foil used for navigation. The little halo around a circle exhibits the impulse response of the helix derivative.

ratio of .183/1.791, we can say the filter pushes zero-frequency amplitude 90% of the way to zero value. When the image contains very much zero-frequency amplitude, more coefficients are needed. I did use more, but simply removing the mean saved me from needing a costly number of filter coefficients.

A final word about the doctor. As she was about to leave my office she suddenly asked if I had scratched one of her X-rays. We were looking at the helix derivative, and it did seem to show a big scratch. What should have been a line was broken into a string of dots. I apologized in advance and handed her the original film negatives, which she proceeded to inspect. “Oh,” she said, “Bad news. There are calcification nodules along the ducts.” So, the scratch was not a scratch, instead an important detail had not been noticed on the original X-ray. Times have changed since then. Nowadays, mammography has become digital; and appropriate filtering is defaulted in the presentation.

In preparing an illustration for here, I learned one more lesson. The scratch was a small part of a big image, so I enlarged a small portion of the mammogram for display here. The very process of selecting a small portion followed by scaling the amplitude between maximum and minimum darkness of printer ink had the effect enhancing the visibility of the scratch on the mammogram. Now, Figure 14 shows the two calcification nodule strings perhaps even clearer than on the helix derivative.

SUBSCRIPTING A MULTIDIMENSIONAL HELIX

Basic utilities transform back and forth between multidimensional matrix coordinates and helix coordinates. The essential module used repeatedly in applications later in this book is `createhelix`. We begin here from its intricate underpinnings.

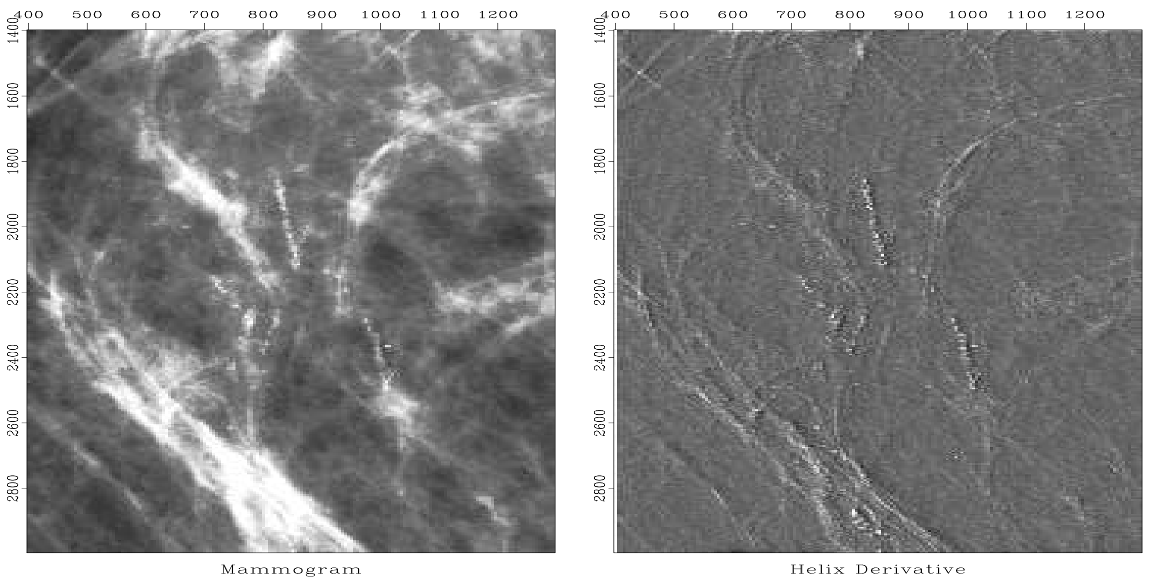

Figure 14: Not a scratch. Reducing the (x, y)-space range of the illustration allowed boosting the gain, thus making the nonscratch more prominent. Find both strings of calcification nodules.

Fortran77 has a concept of a multidimensional array being equivalent to a 1-dimensional array. Given that the hypercube specification `nd=(n1,n2,n3,...)` defines the storage `dimension` of a data array, we can refer to a data element as either `dd(i1,i2,i3,...)` or `dd( i1 +n1*(i2-1) +n1*n2*(i3-1) +...)`. The helix says to refer to the multidimensional data by its equivalent 1-dimensional index (sometimes called its vector subscript or linear subscript).

The filter, however, is a much more complicated story than the data: First, we require all filters to be causal. In other words, the Laplacian does not fit very well, because it is intrinsically noncausal. If you really want noncausal filters, you need to provide your own time shifts outside the tools supplied here. Second, a filter is usually a small hypercube, say `aa(a1,a2,a3,...)` and would often be stored as such. For the helix we must store it in a special 1-dimensional form. Either way, the numbers `na= (a1,a2,a3,...)` specify the dimension of the hypercube. In cube form, the entire cube could be indexed multidimensionally as `aa(i1,i2,...)` or it could be indexed 1-dimensionally as `aa(ia,1,1,...)` or sometimes `aa[ia]` by letting `ia` cover a large range. When a filter cube is stored in its normal "tightly packed" form, the formula for computing its 1-dimensional index `ia` is:

```
ia = i1 +a1*i2 +a1*a2*i3 + ...
```

When the filter cube is stored in an array with the same dimensions as the data, `data[n3][n2][n1]`, the formula for `ia` is:

```
ia = i1 +n1*i2 +n1*n2*i3 + ...
```

The following module `decart` contains two subroutines that explicitly provide us the transformations between the linear index `i` and the multidimensional indices `ii= (i1,i2,...)`. The two subroutines have the logical names `cart2line` and `line2cart`.

api/c/decart.c

```
void sf_line2cart(int dim          /* number of dimensions */,
                  const int* nn /* box size [dim] */,
                  int i         /* line coordinate */,
                  int* ii       /* cartesian coordinates [dim] */)
/*< Convert line to Cartesian >*/
{
    int axis;

    for (axis = 0; axis < dim; axis++) {
        ii[axis] = i%nn[axis];
        i /= nn[axis];
    }
}

int sf_cart2line(int dim          /* number of dimensions */,
                 const int* nn /* box size [dim] */,
                 const int* ii /* cartesian coordinates [dim] */)
/*< Convert Cartesian to line >*/
{
    int i, axis;

    if (dim < 1) return 0;

    i = ii[dim-1];
    for (axis = dim-2; axis >= 0; axis--) {
        i = i*nn[axis] + ii[axis];
    }
    return i;
}
```

The Fortran linear index is closely related to the helix. There is one major difference, however, and that is the origin of the coordinates. To convert from the linear index to the helix lag coordinate, we need to subtract the Fortran linear index of the "1.0" usually taken at `center= (1+a1/2, 1+a2/2, ..., 1)`. (On the last dimension, there is no shift, because nobody stores the volume of zero values that would occur before the 1.0.) The `decart` module fails for negative subscripts. Thus, we need to be careful to avoid thinking of the filter's 1.0 (shown in Figure 8) as the origin of the multidimensional coordinate system although the 1.0 is the origin in the 1-dimensional coordinate system.

Even in 1-D (see the matrix in equation (**??**)), to define a filter *operator*, we need to know not only filter coefficients and a filter length, but we also need to know the data length. To define a multidimensional filter using the helix idea, besides the properties intrinsic to the filter, also the circumference of the helix, i.e., the length on the 1-axis of the data's hypercube as well as the other dimensions `nd=(n1,n2,...)` of the data's hypercube.

Thinking about convolution on the helix, it is natural to think about the filter and data being stored in the same way, that is, by reference to the data size. Such storeage would waste so much space, however, that our helix filter module `helix` instead stores the filter coefficients in one vector and the lags in another. The `i`-th coefficient value of the filter goes in `aa->flt[i]` and the `i`-th lag `ia[i]` goes in `aa->lag[i]`. The lags are the same as the Fortran linear index except for the overall shift of the 1.0 of a cube of data dimension `nd`. Our module for convolution on a helix, `helicon`. has already an implicit "1.0" at the filter's zero lag, so we do not store it. (It is an error to do so.)

Module `createhelix` allocates memory for a helix filter and builds filter lags along the helix from the hypercube description. The hypercube description is not the literal cube seen in Figure 8 but some integers specifying that cube: the data cube dimensions `nd`, likewise the filter cube dimensions `na`, the parameter `center` identifying the location of the filter's "1.0", and a `gap` parameter used in a later chapter. To find the lag table, module `createhelix` first finds the Fortran linear index of the `center` point on the filter hypercube. Everything before that has negative lag on the helix and can be ignored. (Likewise, in a later chapter, we see a `gap` parameter that effectively sets even more filter coefficients to zero so those extra lags can also be ignored.) Then, it sweeps from the center point over the rest of the filter hypercube calculating for a data-sized cube `nd`, the Fortran linear index of each filter element. Near the end of the code you see the calculation of a parameter `lag0d`, which is the count of the number of zeros that a data-sized Fortran array would store in a filter cube preceding the filter's 1.0. We need to subtract this shift from the filter's Fortran linear index to get the lag on the helix.

A filter can be represented literally as a multidimensional cube like equation (9) shows us in two dimensions or like Figure 8 shows us in three dimensions. Unlike the helical form, in literal cube form, the zeros preceding the "1.0" are explicitly present, so `lag0` needs to be added back in to get the Fortran subscript. To convert a helix filter `aa` to Fortran's multidimensional hypercube `cube(n1,n2,...)` is module `boxfilter`: The `boxfilter` module is normally used to display or manipulate a filter that was estimated in helical form (usually estimated by the least-squares method).

A reasonable arrangement for a small 3-D filter is `na=`{`5,3,2`} and `center=`{`3,2,1`}. Using these arguments, I used `createhelix` to create a filter. I set all the helix filter coefficients to 2. Then I used module `boxfilter` to put it in a convenient form for display. Finally, I printed it:

```
0.000  0.000  0.000  0.000  0.000
0.000  0.000  1.000  2.000  2.000
2.000  2.000  2.000  2.000  2.000
---------------------------------
2.000  2.000  2.000  2.000  2.000
2.000  2.000  2.000  2.000  2.000
2.000  2.000  2.000  2.000  2.000
```

Different data sets have different sizes. To convert a helix filter from one data size to another, we could drop the filter into a cube with module `cube`. Then, we could extract it with module `unbox` specifying any data set size we wish. Instead, we use module `regrid` user/gee/createhelix.c

```
sf_filter createhelix(int ndim    /* number of dimensions */,
                      int* nd     /* data size [ndim] */,
                      int* center /* filter center [ndim] */,
                      int* gap    /* filter gap [ndim] */,
                      int* na     /* filter size [ndim] */)
/*< allocate and output a helix filter >*/
{
    sf_filter aa;
    int ii[SF_MAX_DIM], na123, ia, nh, lag0a,lag0d, *lag, i;
    bool skip;

    for (na123 = 1, i=0; i < ndim; i++) na123 *= na[i];
    lag = (int*) alloca(na123*sizeof(int));

    /* index pointing to the "1.0" */
    lag0a = sf_cart2line (ndim, na, center);

    nh=0;
    /* loop over linear index. */
    for (ia = 1+lag0a; ia < na123; ia++) {
            sf_line2cart(ndim, na, ia, ii);

            skip = false;
            for (i=0; i < ndim; i++) {
                    if (ii[i] < gap[i]) {
                            skip = true;
                            break;
                    }
            }
            if (skip) continue;

            lag[nh] = sf_cart2line(ndim, nd, ii);
            nh++;
/* got another live one */
    }
    /* center shift for nd cube */
    lag0d = sf_cart2line(ndim,  nd, center);
    aa = sf_allocatehelix(nh); /* nh becomes size of filter */

    for (ia=0; ia < nh; ia++) {
            aa->lag[ia] = lag[ia] - lag0d;
            aa->flt[ia] = 0.;
    }

    return aa;
}
```

user/gee/boxfilter.c

```
void box (int dim              /* number of dimaneions */,
          const int *nd        /* data size [dim] */,
          const int *center    /* filter center [dim] */,
          const int *na        /* filter size [dim] */,
          const sf_filter aa   /* input filter */,
          int nc               /* box size */,
          float* cube          /* output box [nc] */)
/*< box it >*/
{
    int ii[SF_MAX_DIM];
    int j, lag0a, lag0d, id, ia;

    for (ia=0; ia < nc; ia++) {
        cube[ia] = 0.;
    }
    lag0a = sf_cart2line(dim, na, center);   /* 1.0 in na. */
    cube[lag0a] = 1.;                         /* place it. */
    lag0d = sf_cart2line(dim, nd, center);   /* 1.0 in nd. */
    for (j=0; j < aa->nh; j++) { /* inspect the entire helix */
        id = aa->lag[j] + lag0d;
        sf_line2cart(dim, nd, id, ii);   /* ii = cartesian  */
        ia = sf_cart2line(dim, na, ii); /* ia = linear in aa */
        cube[ia] = aa->flt[j];           /* copy the filter */
    }
}
```

prepared by Sergey Fomel which does the job without reference to an underlying filter cube. He explains his `regrid` module thus:

> Imagine a filter being cut out of a piece of paper and glued on another paper, which is then rolled to form a helix.
>
> We start by picking a random point (let us call it `rand`) in the cartesian grid and placing the filter so that its center (the leading 1.0) is on top of that point. `rand` should be larger than (or equal to) `center` and smaller than `min (nold, nnew)`, otherwise the filter might stick outside the grid (our piece of paper.) `rand=nold/2` will do (assuming the filter is small), although nothing should change if you replace `nold/2` with a random integer array between `center` and `nold - na`.
>
> The linear coordinate of `rand` is `h0` on the old helix and `h1` on the new helix. Recall that the helix lags `aa->lag` are relative to the center. Therefore, we need to add `h0` to get the absolute helix coordinate (`h`). Likewise, we need to subtract `h1` to return to a relative coordinate system.

user/gee/regrid.c

```c
void regrid( int dim         /* number of dimensions */,
             const int* nold /* old data size [dim] */,
             const int* nnew /* new data size [dim] */,
             sf_filter aa    /* modified filter */)
/*< change data size >*/
{
    int i, h0, h1, h, ii[SF_MAX_DIM];

    for (i=0; i < dim; i++) {
        ii[i] = nold[i]/2-1;
    }

    h0 = sf_cart2line( dim, nold, ii); /* midpoint lag on nold */
    h1 = sf_cart2line( dim, nnew, ii); /*              on nnew */
    for (i=0; i < aa->nh; i++) { /* for all filter coefficients */
        h = aa->lag[i] + h0;
        sf_line2cart( dim, nold, h, ii);
        aa->lag[i] = sf_cart2line( dim, nnew, ii) - h1;
    }
}
```

INVERSE FILTERS AND OTHER FACTORIZATIONS

Mathematics sometimes seems a mundane subject, like when it does the “accounting” for an engineer. Other times, as with the study of causality and spectral factorization, it brings unexpected amazing new concepts into our lives. There are many little-known, fundamental ideas here; a few touched on next.

Start with an example. Consider a mechanical object. We can strain it and watch it stress or we can stress it and watch it strain. We feel knowledge of the present and past stress history is all we need to determine the present value of strain. Likewise, the converse, history of strain should tell us the stress. We could say there is a filter that takes us from stress to strain; likewise, another filter takes us from strain to stress. What we have here is a pair of filters that are mutually inverse under convolution. In the Fourier domain, one is literally the inverse of the other. What is remarkable is that in the time domain, both are causal. They both vanish before zero lag $\tau = 0$.

Not all causal filters have a causal inverse. The best known name for one that does is "minimum-phase filter." Unfortunately, this name is not suggestive of the fundamental property of interest, "causal with a causal (convolutional) inverse." I could call it CCI. An example of a causal filter without a causal inverse is the unit delay operator—with Z-transforms, the operator Z. If you delay something, you cannot get it back without seeing into the future, which you are not allowed to do. Mathematically, $1/Z$ cannot be expressed as a polynomial (actually, a convergent infinite series) in positive powers of Z.

Physics books do not tell us where to expect to find transfer functions that are CCI. I think I know why they do not. Any causal filter has a "sharp edge" at zero time lag where it switches from nonresponsiveness to responsiveness. The sharp edge might cause the spectrum to be large at infinite frequency. If so, the inverse filter is small at infinite frequency. Either way, one of the two filters is unmanageable with Fourier transform theory, which (you might have noticed in the mathematical fine print) requires signals (and spectra) to have finite energy. Finite energy means the function must get really small in that immense space on the t-axis and the ω axis. It is impossible for a function to be small and its inverse be small. These imponderables become manageable in the world of Time Series Analysis (discretized time axis).

Uniqueness and invertability

Interesting questions arise when we are given a spectrum and find ourselves asking how to find a filter that has that spectrum. Is the answer unique? We will see it is not unique. Is there always an answer that is causal? Almost always, yes. Is there always an answer that is causal with a causal inverse (CCI)? Almost always, yes.

Let us have an example. Consider a filter like the familiar time derivative $(1, -1)$, except let us down weight the -1 a tiny bit, say $(1, -\rho)$ where $0 << \rho < 1$. Now, the filter $(1, -\rho)$ has a spectrum $(1 - \rho Z)(1 - \rho/Z)$ with autocorrelation coefficients $(-\rho, 1 + \rho^2, -\rho)$ that look a lot like a second derivative, but it is a tiny bit bigger in the middle. Two different waveforms, $(1, -\rho)$ and its time reverse both have the same autocorrelation. In principle, spectral factorization could give us both $(1, -\rho)$ and $(\rho, -1)$, but we always want only the one that is CCI, which is the one we get from Kolmogoroff. The bad one is weaker on its first pulse. Its inverse is not causal. Following are two expressions for the filter inverse to $(\rho, -1)$, the first divergent (filter coefficients at infinite lag are infinitely strong), the second convergent but noncausal.

$$\frac{1}{\rho - Z} \quad = \quad \frac{1}{\rho}\,(1 + Z/\rho + Z^2/\rho^2 + \cdots) \tag{16}$$

$$\frac{1}{\rho - Z} \quad = \quad \frac{-1}{Z}\,(1 + \rho/Z + \rho^2/Z^2 + \cdots) \tag{17}$$

(Please multiply each equation by $\rho - Z$, and see it reduce to $1 = 1$.)

We begin with a power spectrum, and our goal is to find a CCI filter with that spectrum. If we input to the filter an infinite sequence of random numbers (white noise), we should output something with the original power spectrum.

We easily inverse Fourier transform the square root of the power spectrum, getting a symmetrical time function, but we need a function that vanishes before $\tau = 0$. On the other hand, if we already had a causal filter with the correct spectrum we could manufacture many others. To do so, all we need is a family of delay operators for convolution. A pure delay filter does not change the spectrum of anything—same for frequency-dependent delay operators. Here is an example of a frequency-dependent delay operator: First, convolve with (1,2) and then deconvolve with (2,1). Both these have the same amplitude spectrum, so the ratio has a unit amplitude (and nontrivial phase). If you multiply $(1 + 2Z)/(2 + Z)$, by its Fourier conjugate (replace Z by $1/Z$) the resulting spectrum is 1 for all ω.

Anything with a nature to delay is death to CCI. The CCI has its energy as close as possible to $\tau = 0$. More formally, my first book, *FGDP* proves the CCI filter has for all time τ more energy between $t = 0$ and $t = \tau$ than any other filter with the same spectrum.

Spectra can be factorized by an amazingly wide variety of techniques, each of which gives you a different insight into this strange beast. Spectra can be factorized by factoring polynomials, inserting power series into other power series, solving least squares problems, and by taking logarithms and exponentials in the Fourier domain. I have coded most of of these methods, and find each seemingly unrelated to the others.

Theorems in Fourier analysis can be interpreted physically in two different ways, one as given, and the other with time and frequency reversed. For example, convolution in one domain amounts to multiplication in the other. If we express the CCI concept with reversed domains, instead of saying the "energy comes as quick as possible after $\tau = 0$," we would say "the frequency function is as close to $\omega = 0$ as possible." In other words, it is minimally wiggly with time. Most applications of spectral factorization begin with a spectrum, a real, positive function of frequency. I once recognized the opposite case and achieved minor fame by starting with a real, positive function of space, a total **magnetic** field $\sqrt{H_x^2 + H_z^2}$ measured along the x-axis; and I reconstructed the magnetic field components H_x and H_z that were minimally wiggly in space (*FGDP*, page 61).

Cholesky decomposition

Conceptually, the simplest computational method of spectral factorization might be "Cholesky decomposition." For example, the matrix of (15) could have been found by Cholesky factorization of (14). The Cholesky algorithm takes a positive-definite matrix $\mathbf{Q}$ and factors it into a triangular matrix times its transpose, say $\mathbf{Q} = \mathbf{T}^{\mathrm{T}}\,\mathbf{T}$.

It is easy to reinvent the Cholesky factorization algorithm. To do so, simply write all the components of a 3×3 triangular matrix $\mathbf{T}$ and then explicitly multiply these elements times the transpose matrix $\mathbf{T}^{\mathrm{T}}$. You then find you have everything you need to recursively build the elements of $\mathbf{T}$ from the elements of $\mathbf{Q}$. Likewise, for a 4×4 matrix, etc.

The 1×1 case shows that the Cholesky algorithm requires square roots. Matrix elements are not always numbers. Sometimes, matrix elements are polynomials, such as Z-transforms. To avoid square roots, there is a variation of the Cholesky method. In this variation, we factor $\mathbf{Q}$ into $\mathbf{Q} = \mathbf{T}^{\mathrm{T}}\mathbf{D}\mathbf{T}$, where $\mathbf{D}$ is a diagonal matrix.

Once a matrix has been factored into upper and lower triangles, solving simultaneous equations is simply a matter of two back substitutions: (We looked at a special case of back substitution with Equation (**??**).) For example, we often encounter simultaneous equations of the form $\mathbf{B}^{\mathrm{T}}\,\mathbf{B}\mathbf{m} = \mathbf{B}^{\mathrm{T}}\,\mathbf{d}$. Suppose the positive-definite matrix $\mathbf{B}^{\mathrm{T}}\,\mathbf{B}$ has been factored into triangle form $\mathbf{T}^{\mathrm{T}}\,\mathbf{T}\mathbf{m} = \mathbf{B}^{\mathrm{T}}\,\mathbf{d}$. To find $\mathbf{m}$, first backsolve $\mathbf{T}^{\mathrm{T}}\,\mathbf{x} = \mathbf{B}^{\mathrm{T}}\,\mathbf{d}$ for the vector $\mathbf{x}$. Then, we backsolve $\mathbf{T}\mathbf{m} = \mathbf{x}$. When $\mathbf{T}$ happens to be a band matrix, then the first back substitution is filtering down a helix, and the second is filtering back up it. Polynomial division is a special case of back substitution.

Poisson's equation $\nabla^2\mathbf{p} = -\mathbf{q}$ requires boundary conditions, that we can honor when we filter starting from both ends. We cannot simply solve Poisson's equation as an initial-value problem. We could insert the Laplace operator into the polynomial division program, but the solution would diverge.

Toeplitz methods

Band matrices are often called Toeplitz matrices. In the subject of Time Series Analysis are found spectral factorization methods that require computations proportional to the dimension of the matrix squared. These calculations can often be terminated early with a reasonable partial result. Two Toeplitz methods, the Levinson method and the Burg method, are described in my first textbook, *FGDP*. Our interest is multidimensional data sets, so the matrices of interest are truely huge and the cost of Toeplitz methods is proportional to the square of the matrix size. Thus, before we find Toeplitz methods especially useful, we may need to find ways to take advantage of the sparsity of our filters.