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.

Many geophysical map estimation problems appear to be multidimensional, but actually they are not. To see the tip of the iceberg, consider this example: On a two-dimensional cartesian mesh, the function $\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$

has the autocorrelation $\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$.

Likewise, on a one-dimensional cartesian mesh,

the function $\mathbf{b} = (1,1,0,0,\cdots,0,1,1)$

has the autocorrelation $\mathbf{r} = (1,2,1,0,\cdots,0,2,4,2,0,\cdots,1,2,1)$.

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

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.

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.

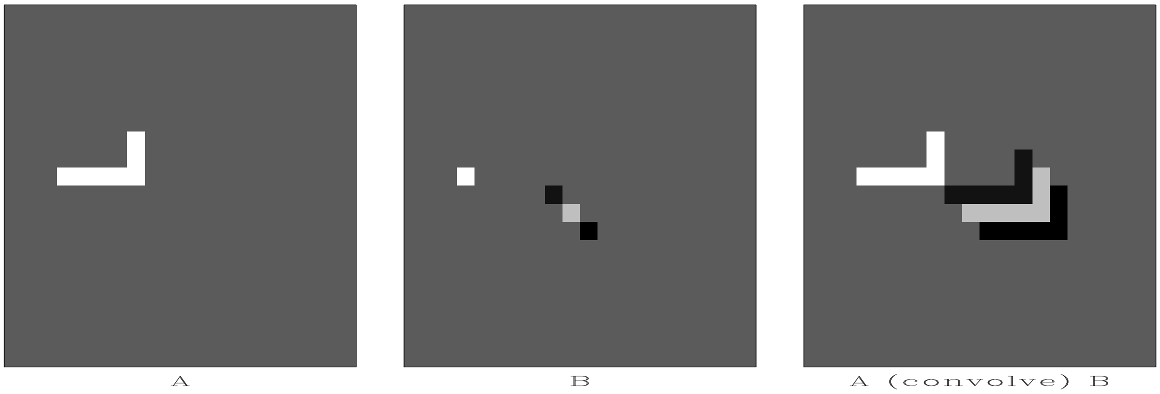

Figure 4.1: Two-dimensional convolution as performed in one dimension by module `helicon` hlx-diamond90 [ER]

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 $\partial_{xx} + \partial_{yy}$. 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.

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.

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

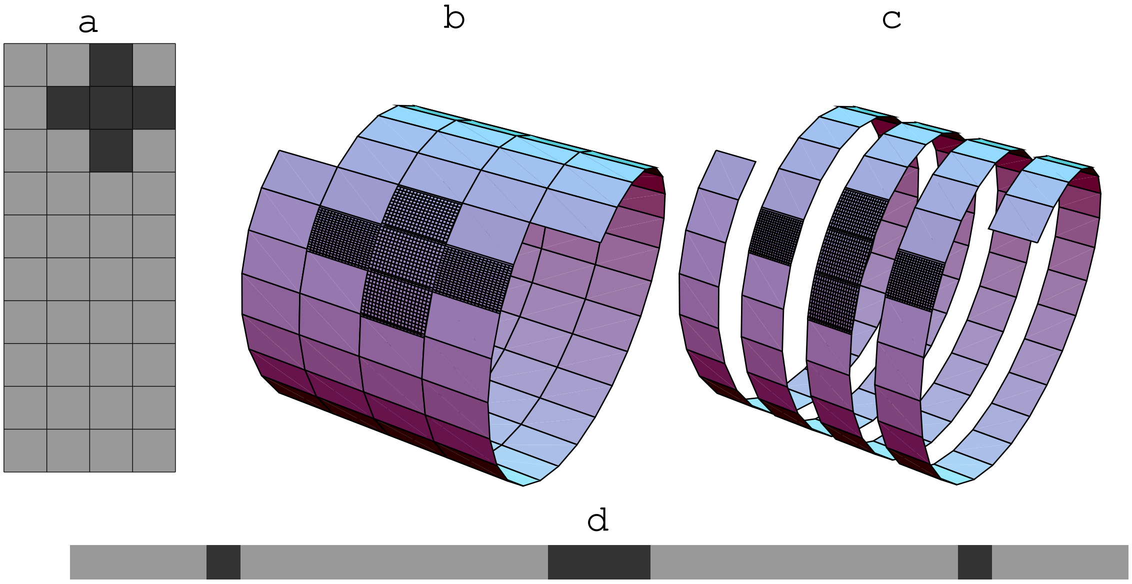

Figure 4.2: Filtering on a helix. The same filter coefficients overlay the same data values if the 2-D coils are unwound into 1-D strips. (*Mathematica* drawing by Sergey Fomel) hlx-sergey-helix [CR]

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 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. 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.

4.1.3 Examples of simple 2-D recursive filters

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

$$\frac{\partial}{\partial x} = \begin{bmatrix} 1 & -1 \end{bmatrix} \tag{4.3}$$

$$\frac{\partial}{\partial y} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \tag{4.4}$$

Convolving a data plane with the stencil (1.3) forms the x-derivative of the plane. Convolving a data plane with the stencil (1.4) forms the y-derivative of the plane. On the other hand, *deconvolving* with (1.3) integrates data along the x-axis for each y. Likewise, deconvolving with (1.4) integrates data along the y-axis for each x. Next we look at a fully two-dimensional operator (like the cross derivative ∂_{xy}).

A nontrivial two-dimensional convolution stencil is

$$\begin{bmatrix} 0 & -1/4 \\ 1 & -1/4 \\ -1/4 & -1/4 \end{bmatrix} \tag{4.5}$$

We will 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 (1.1) and

(1.2). Let us manufacture the simple data plane shown on the left in Figure 1.3. Beginning with a zero-valued plane, we add in a copy of the filter (1.5) 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 1.3 is the result of deconvolution by the filter (1.5) using the one-dimensional equation (1.2). 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. The filtering in Figure 1.3 ran along a helix from left

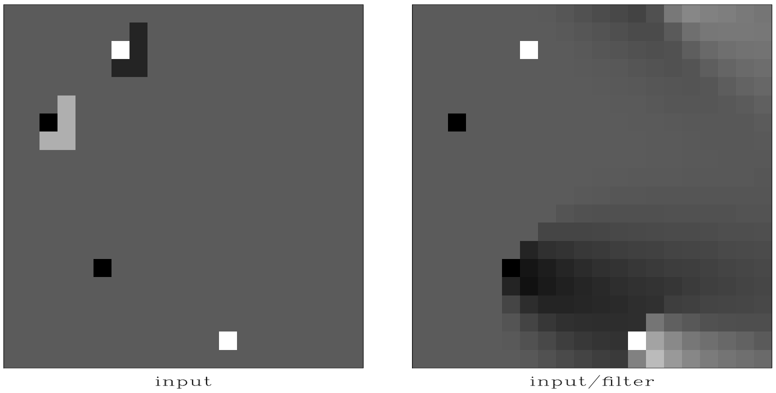

Figure 4.3: Illustration of 2-D deconvolution. Left is the input. Right is after deconvolution with the filter (1.5) as preformed by by module `polydiv` hlx-wrap90 [ER]

to right. Figure 1.4 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 symmetical 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, no evidence remains of any of the boundaries where we have wrapped and truncated.

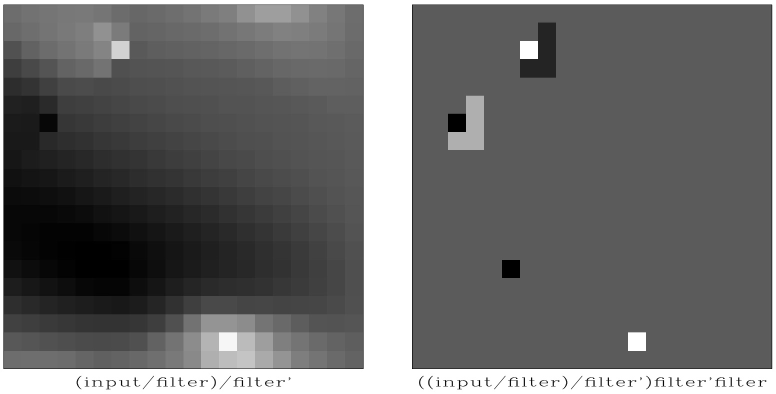

Figure 4.4: Recursive filtering backwards (leftward on the space axis) is done by the *adjoint* of 2-D deconvolution. Here we see that 2-D *deconvolution* compounded with its adjoint is exactly inverted by 2-D *convolution* and its adjoint. hlx-hback90 [ER]

Chapter 5 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.

In seismology we often have occasion to steer summation along beams. Such an impulse response is shown in Figure 1.6. 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 1.5.

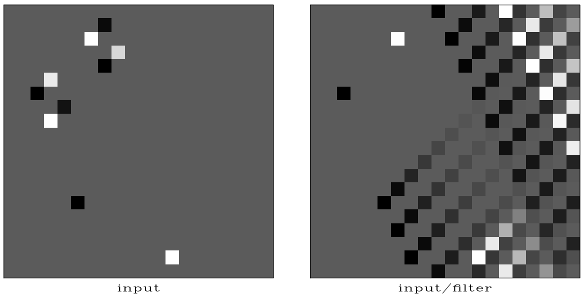

Figure 4.5: A simple low-order 2-D filter whose inverse contains plane waves of two different dips. One of them is spatially aliased. hlx-waves90 [ER]

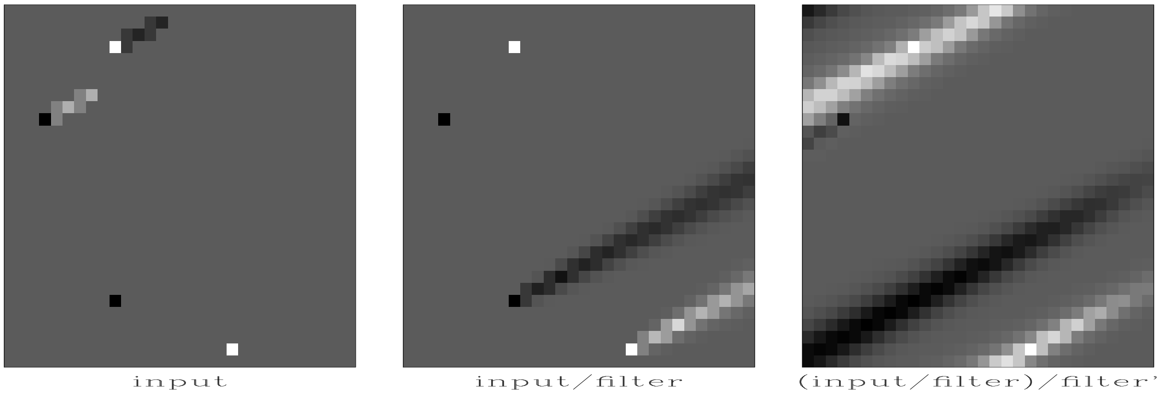

Figure 4.6: A simple low-order 2-D filter whose inverse times its inverse adjoint, is approximately a dipping seismic arrival. hlx-dip90 [ER]

4.1.4 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 8 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

$$\begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix} \xrightarrow{\text{helical boundary conditions}} (1, 0, \ ...\ 0, 1, -4, 1, 0, \ ...\ 0, 1). \tag{4.6}$$

The first filter coefficient in equation (1.6) is $+1$ as implicit to module `helicon`. To apply the Laplacian on a 1000×1000 mesh requires the filter inputs:

```
 i     lag(i)    flt(i)
---    ------    -----
 1       999       1
 2      1000      -4
 3      1001       1
 4      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 will be 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 will not be computed (either because of boundary conditions or because of missing inputs). The filter vectors are the size of the nonzero filter coefficents (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. By default, the logical vector is not allocated but is set to `null` with the `nullify` operator and ignored.

```
module helix {                                    # DEFINE helix filter type
  type filter {
    real,    dimension( :), pointer :: flt       # (nh) filter coefficients
    integer, dimension( :), pointer :: lag       # (nh) filter lags
    logical, dimension( :), pointer :: mis       # (nd) boundary conditions
  }
contains
  subroutine allocatehelix( aa, nh ) {            # allocate a filter
    type( filter) :: aa
    integer       :: nh                           # count of filter coefs (excl 1)
    allocate( aa%flt( nh), aa%lag( nh))           # allocate filter and lags.
    nullify( aa%mis)                              # set null pointer for "mis".
    aa%flt = 0.                                   # zero filter coef values
  }
  subroutine deallocatehelix( aa) {               # destroy a filter
    type( filter) :: aa
    deallocate( aa%flt, aa%lag)                   # free memory
    if( associated( aa%mis))                      # if logicals were allocated
```

```
        deallocate( aa%mis)                     # free them
  }
}
```

For those of you with no Fortran 90 experience, the "`%`" appearing in the helix module denotes a pointer. Fortran 77 has no pointers (or everything is a pointer). The C, C++, and Java languages use "`.`" to denote pointers. C and C++ also have a second type of pointer denoted by "`->`". The behavior of pointers is somewhat different in each language. Never-the-less, the idea is simple. In module `helicon` on page 8 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(2)` which in the example above refers to the value 4.0 in the center of the laplacian operator. For data sets like above with 1000 points on the 1-axis, this value 4.0 occurs after 1000 lags, thus `aa%lag(2)=1000`.

Our first convolution operator `tcai1` on page 9 was limited to one dimension and a particular choice of end conditions. With the helix and Fortran 90 pointers, the operator `helicon` on page 8 is a *multidimensional* filter with considerable flexibility (because of the `mis` vector) to work around boundaries and missing data.

```
module helicon {                        # Convolution, inverse to deconvolution.
#              Requires the filter be causal with an implicit "1." at the onset.
use helix
type( filter) :: aa
#%  _init( aa)
#%  _lop ( xx,  yy)
integer iy, ix, ia
if( adj)               # zero lag
        xx += yy
else
        yy += xx
do ia = 1, size( aa%lag) {
      do iy = 1  + aa%lag( ia), size( yy) {
            if( associated( aa%mis)) { if( aa%mis( iy)) cycle}
            ix = iy - aa%lag( ia)
            if( adj)
                        xx(ix) += yy(iy) * aa%flt(ia)
            else
                        yy(iy) += xx(ix) * aa%flt(ia)
            }
      }
}
```

The code fragment `aa%lag(ia)` corresponds to `ib-1` in `tcai1` on page 9.

Operator `helicon` did the convolution job for Figure 1.1. As with `tcai1` on page 9 the adjoint of filtering is filtering backwards which means unscrewing the helix.

The companion to convolution is deconvolution. The module `polydiv` on page 9 is essentially the same as `polydiv1` on page 19, but here it was coded using our new `filter` type in module `helix` on page 19 which will simplify 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)`.

```
module polydiv {                          # Helix polynomial division
use helix
integer                         :: nd
type( filter)                   :: aa
real, dimension (nd), allocatable :: tt
#%  _init ( nd, aa)
#%  _lop  ( xx, yy)
integer  ia, ix, iy
tt = 0.
if( adj) {
        do ix= nd, 1, -1 {
                tt( ix) = yy( ix)
                do ia = 1, size( aa%lag) {
                        iy = ix + aa%lag( ia);     if( iy > nd)  next
                        tt( ix) -=  aa%flt( ia) * tt( iy)
                        }
                }
        xx += tt
 } else {
        do iy= 1, nd {
                tt( iy) = xx( iy)
                do ia = 1, size( aa%lag) {
                        ix = iy - aa%lag( ia);      if( ix < 1)  next
                        tt( iy) -=  aa%flt( ia) * tt( ix)
                        }
                }
        yy += tt
        }
}
```

EXERCISES:

1 Observe the matrix matrix (1.4) which corresponds to subroutine `tcai1` on page 9. What is the matrix corresponding to `helicon` on page 8?

4.1.5 Causality in two-dimensions

In one dimension, most filters of interest have a short memory. Significant filter coefficients are concentrated shortly after $t = 0$. The favorite example in Physics is the damped harmonic oscillator, all of which is packed into a two-lag filter (second order differential equation). 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 (1.1) and (1.2) work as desired and expected. These filters are called “minimum phase”. There are also filters for which (1.2) is a disaster numerically, the feedback process diverging to infinity.

Divergent cases correspond to physical processes that require boundary conditions. Equation (1.2) only allows for initial conditions. I oversimplify by trying to collapse an entire book (FGDP) into a few sentances by saying here that for any fixed 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.

Now let us turn to two dimensions. Filters of interest will correspond to energy concentrated near the end of a helix. Let us examine the end of a helix. At the very end, as in the 1-D case, is a coefficient with the numerical value 1.0. Keeping only coefficients within two mesh points in any direction from the 1.0, we copy the coefficients from near the end of the helix to a cartesian mesh like this:

$$
\begin{array}{ccccccccccc}
h & c & 0 & & h & c & \cdot & & \cdot & \cdot & 0 \\
p & d & 0 & & p & d & \cdot & & \cdot & \cdot & 0 \\
q & e & \mathbf{1} & = & q & e & \cdot & + & \cdot & \cdot & \mathbf{1} \\
s & f & a & & s & f & a & & \cdot & \cdot & \cdot \\
u & g & b & & u & g & b & & \cdot & \cdot & \cdot \\
 & & & & & & & & & & \\
 & 2-\mathrm{D\ filter} & & = & & \mathrm{variable} & & + & & \mathrm{constrained} &
\end{array}
\tag{4.7}
$$

where $a, b, c, ..., u$ are adjustable coefficients.

Which side of the little rectangular patch of coefficients we choose to place the 1.0 is rather arbitrary. The important matter is that as a matter of principle, the 1.0 is expected to lie along one side of the little patch. It is rarely (if ever) found at a corner of the patch. It is important that beyond the 1.0 (in whatever direction that may be) the filter coefficients must be zero because in one dimension, these coefficients lie before zero lag. Our foundations, the basic convolution-deconvolution pair (1.1) and (1.2) are applicable only to filters with all coefficients *after* zero lag.

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. Next we look at a particularly important autocorrelation function and see how 2-D is the same as 1-D.

4.2 FINITE DIFFERENCES ON A HELIX

The function

$$
\mathbf{r} \quad = \quad (-1, 0, \cdots, 0, -1, 4, -1, 0, \cdots, 0, -1) \tag{4.8}
$$

is an autocorrelation function. It is symmetrical about the "4" and its Fourier transform is positive for all frequencies. Digging out our old textbooks[1] we discover how to compute

[1] PVI or FGDP, for example, explain spectral factorization. More concisely in PVI, more richly in FGDP.

a causal wavelet with this autocorrelation. I used the "Kolmogoroff spectral-factorization method" to find this wavelet **h**:

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

Now for the magic: Wind the signal **r** around a vertical-axis helix to see its two-dimensional shape **R**:

$$\mathbf{R} \quad = \quad \begin{bmatrix} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{bmatrix} \xrightarrow{\text{helical boundaries}} (1,\, 0,\, ...\, 0,\, 1,\, -4,\, 1,\, 0,\, ...\, 0,\, 1) \quad = \quad \mathbf{r} \tag{4.10}$$

This 2-D filter 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 wind the signal **h** around the same helix to see its two-dimensional shape **H**

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

In the representation (1.11) we see the coefficients diminishing rapidly away from maximum value 1.791. My claim is that the autocorrelation of (1.11) is (1.10). You verified this idea earlier 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.

Since the autocorrelation of **H** is $\mathbf{H}'\mathbf{H} = \mathbf{R} = -\nabla^2$ is a second derivative, the operator **H** must be something like a first derivative. As a geophysicist, I found it natural to compare the operator $\frac{\partial}{\partial y}$ with **H** by applying them to a local topographic map. The result shown in Figure 1.7 is that **H** enhances drainage patterns whereas $\frac{\partial}{\partial y}$ enhances mountain ridges.

The operator **H** has curious similarities and differences with the familiar gradient and divergence operators. In two-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 **H** that maps one field to *one* field. Being a one-to-one transformation (unlike gradient and divergence) the operator **H** is potentially invertible by deconvolution (recursive filtering).

I have chosen the name[2] "helix derivative" or "helical derivative" for the operator **H**. A telephone pole has a narrow shadow behind it. The helix integral (middle frame of Figure 1.8) and the helix derivative (left frame) show shadows with an angular bandwidth approaching 180°.

Our construction makes **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 and with value zero at the origin. 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.

[2]Any fact this basic should be named in some earlier field of mathematics or theoretical physics. 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.

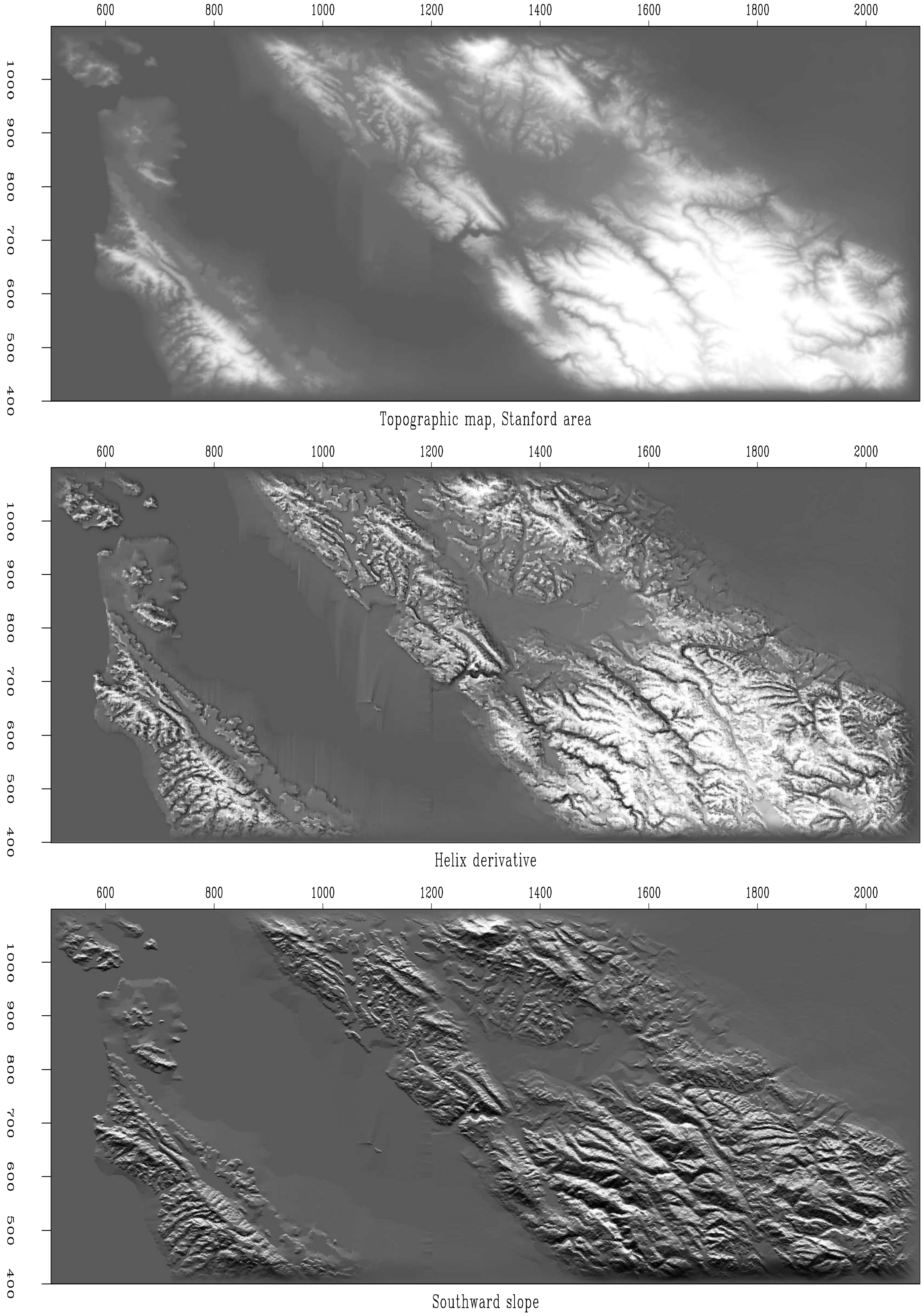

Figure 4.7: Topography, helical derivative, slope south. hlx-helocut90 [ER,M]

The rotationally invariant cone 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.

This is where the story all comes together. One-dimensional theory, either the old Kolmogoroff spectral factorization, or the new Wilson-Burg spectral-factorization method produces not merely a causal wavelet with the required autocorrelation. It produces one that is stable in deconvolution. Using $\mathbf{H}$ in one-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 $\mathbf{p} = (\mathbf{q}/\mathbf{H})/\mathbf{H}'$ which means that we have solved $\nabla^2 \mathbf{p} = -\mathbf{q}$ 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 1.8.

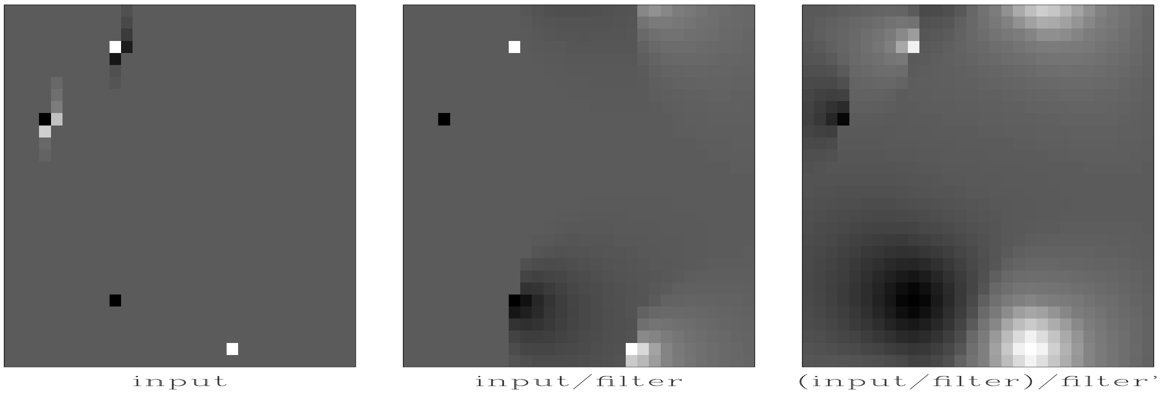

Figure 4.8: Deconvolution by a filter whose autocorrelation is the two-dimensional Laplacian operator. Amounts to solving the Poisson equation. Left is $\mathbf{q}$; Middle is $\mathbf{q}/\mathbf{H}$; Right is $(\mathbf{q}/\mathbf{H})/\mathbf{H}'$. hlx-lapfac90 [ER]

Figure 1.8 contains both the helix derivative and its inverse. Contrast them 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.

4.2.1 Matrix view of the helix

Physics on a helix can be viewed thru the eyes of matrices and numerical analysis. This 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 the Laplacian operator $\partial_{xx} + \partial_{yy}$

into a matrix. For a 4×3 plane, that matrix is shown in equation (1.12).

$$
- \nabla^2 \quad = \quad
\left|
\begin{array}{cccc|cccc|cccc}
\hline
4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
-1 & 4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & -1 & 4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & -1 & 4 & h & \cdot & \cdot & -1 & \cdot & \cdot & \cdot & \cdot \\
\hline
-1 & \cdot & \cdot & h & 4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot & \cdot \\
\cdot & -1 & \cdot & \cdot & -1 & 4 & -1 & \cdot & \cdot & -1 & \cdot & \cdot \\
\cdot & \cdot & -1 & \cdot & \cdot & -1 & 4 & -1 & \cdot & \cdot & -1 & \cdot \\
\cdot & \cdot & \cdot & -1 & \cdot & \cdot & -1 & 4 & h & \cdot & \cdot & -1 \\
\hline
\cdot & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & h & 4 & -1 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & -1 & 4 & -1 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & -1 & 4 & -1 \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -1 & \cdot & \cdot & -1 & 4 \\
\hline
\end{array}
\right|
\tag{4.12}
$$

The two-dimensional matrix of coefficients for the Laplacian operator is shown in (1.12), 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 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 (1.12). 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, but fortunately they seem to be converging rapidly (in the middle) so truncation (of the 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

A like this:

$$
\mathbf{A} \;=\; \left[\begin{array}{cccccccccccc}
1.8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
-.6 & 1.8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
0.0 & -.6 & 1.8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
-.2 & 0.0 & -.6 & 1.8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
-.6 & -.2 & 0.0 & -.6 & 1.8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & -.6 & -.2 & 0.0 & -.6 & 1.8 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & -.6 & -.2 & 0.0 & -.6 & 1.8 & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & -.6 & -.2 & 0.0 & -.6 & 1.8 & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & -.6 & -.2 & 0.0 & -.6 & 1.8 & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & -.6 & -.2 & 0.0 & -.6 & 1.8 & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -.6 & -.2 & 0.0 & -.6 & 1.8 & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & -.6 & -.2 & 0.0 & -.6 & 1.8
\end{array}\right] \tag{4.13}
$$

If you will allow me some truncation approximations, I now claim that the laplacian represented by the matrix in equation (1.12) is factored into two parts $-\nabla^2 = \mathbf{A}'\mathbf{A}$ which are upper and lower triangular matrices whose product forms the autocorrelation seen in (1.12). Recall that triangular matrices allow quick solutions of simultaneous equations by backsubstitution. That is what we do with our deconvolution program.

4.3 SPECTRAL FACTORIZATION

Spectral factorization is an abstract and deep mathematical subject that manifests itself many ways in the world of computational physics and engineering. I'll point to several possibilities before we pursue one of them to the level of a useful algorithm.

4.3.1 Cholesky decomposition

The world of positive-definite matrices contains an important and delightfully simple algorithm called the "Cholesky decomposition". This algorithm takes a positive-definite matrix $\mathbf{Q}$ and factors it into a triangular matrix times its transpose, say $\mathbf{Q} = \mathbf{T}'\mathbf{T}$. Equation (1.23) is an example of a banded triangular matrix.

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 $\mathbf{T}'$. You will find that 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 they 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 $\mathbf{Q} = \mathbf{T}'\mathbf{D}\mathbf{T}$ 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 backsubstitutions: (We looked at a special case of back-substitution with equation (1.23).) For example, we often encounter simultaneous equations of the form $\mathbf{B}'\mathbf{B}\mathbf{m} = \mathbf{B}'\mathbf{d}$. Suppose the positive-definite matrix $\mathbf{B}'\mathbf{B}$ has been factored into triangle form $\mathbf{T}'\mathbf{T}\mathbf{m} = \mathbf{B}'\mathbf{d}$. To find $\mathbf{m}$ we first backsolve $\mathbf{T}'\mathbf{x} = \mathbf{B}'\mathbf{d}$ 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 backsubstitution is filtering down a helix and the second is filtering back up it. Poisson's equation $\nabla^2\mathbf{p} = -\mathbf{q}$ requires boundary conditions which 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.

Being a matrix method, the Cholesky method of factorization has a cost proportional to the cube of the size of the matrix. Because our problems are very large and because the Cholesky method does not produce a useful result if we stop part way to completion, we look further. The Cholesky method is a powerful method but it does more than we require. The Cholesky method does not require band matrices, yet these matrices are what we generally find in applications, so we seek methods that take advantage of the special properties of band matrices.

4.3.2 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. They can often be terminated early with a reasonable partial result. 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. Two Toeplitz methods, the Levinson method and the Burg method are described in my first textbook, FGDP.

4.3.3 Kolmogoroff spectral factorization

With Fourier analysis we find a method of spectral factorization that is as fast as Fourier transformation, namely $N \log N$ for a matrix of size N. This is very appealing. An earlier version of this book included such an algorithm. Pedagogically, I didn't like it in this book because it requires lengthy off-topic discussions of Fourier analysis which are already found in both my first book FGDP and my third book PVI.

The weakness of the Kolmogoroff method is related to its strength. Fourier methods strictly require the matrix to be a band matrix. A matrix that is "almost" a band matrix adds complexity beyond my textbooks.

4.4 WILSON-BURG SPECTRAL FACTORIZATION

(If you are new to this material, you should pass over this section.) Spectral factorization is the job of taking a power spectrum and from it finding a causal (zero before zero time) filter with that spectrum. Methods for this task (there are many) not only produce a causal wavelet, but they typically produce one whose convolutional inverse is also causal. (This is called the "minimum phase" property.) In other words, with such a filter we can do stable deconvolution. Here I introduce a new method of spectral factorization that looks particularly suitable for the task at hand. I learned this new method from John Parker Burg who attributes it to an old paper by Wilson (I find Burg's explanation, below, much clearer than Wilson's.)

Below find subroutine `lapfac()` which was used in the previous section to factor the Laplacian operator. To invoke the factorization subroutine, you need to supply one side of an autocorrelation function. For example, let us specify the negative of the 2-D Laplacian (an autocorrelation) in a vector `n` = 256×256 points long.

```
rr(1)     =  4.
rr(2)     = -1.
rr(1+256) = -1.
```

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 {                              # Factor 2-D Laplacian.
  use wilson
contains
  function lapfac2( eps, n1, na)   result (aa) {
    type( filter)                        :: aa, lap
    real,                  intent( in) :: eps
    integer,               intent( in) :: n1, na
    integer                            :: i
    real                               :: a0, lap0
    call allocatehelix( lap, 2)        # laplacian filter
    lap0 = 4. + eps                    # zero lag coeff.
    lap%lag = (/ 1, n1 /)              # lag(1)= 1; lag(2)=n1  # one side only
    lap%flt = -1.                      # flt(1)=-1; flt(2)=-1
    call allocatehelix( aa, 2*na)      # laplacian derivative
    aa%flt = 0.;                       # probably done already in allocation.
    do i = 1, na {
       aa%lag( i   ) =      i          # early lags (first row)
       aa%lag( na+i) = n1 + i - na     # late lags (second row)
       }
    call wilson_init( 10 * n1 )
    call wilson_factor( 20, lap0, lap, a0, aa)
    call wilson_close()
    call deallocatehelix( lap)
    }
}
```

Subroutine `lapfacn()` has its main job done by subroutine `wilson_factor()` on page 19 shown after the Wilson-Burg theory.

4.4.1 Wilson-Burg theory

Newton's iteration for square roots

$$a_{t+1} \quad = \quad \frac{1}{2}\left(a_t + \frac{s}{a_t}\right) \tag{4.14}$$

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

$$a_{t+1} - \sqrt{s} \quad \sim \quad (a_t - \sqrt{s})^2 \quad = \quad a_t^2 - 2a_t\sqrt{s} + s \quad > \quad 0 \tag{4.15}$$

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 (1.15) 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 (1.15) becomes itself the Newton iteration (1.14).

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 (1.15).

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

$$2\,\frac{a_{t+1}}{a_t} \quad = \quad 1 + \frac{s}{a_t^2} \tag{4.16}$$

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

$$\frac{\bar{A}_{t+1}(1/Z)}{\bar{A}_t(1/Z)} + \frac{A_{t+1}(Z)}{A_t(Z)} \quad = \quad 1 + \frac{S(Z)}{\bar{A}_t(1/Z)\,A_t(Z)} \tag{4.17}$$

Now we are ready for the algorithm: Compute the right side of (1.17) 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.

(Parenthetically, for those people familiar with the idea of minimum phase (if not, see FGDP or PVI), we show that $A_{t+1}(Z)$ is minimum phase: Both sides of (1.17) 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 (1.17) 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 (1.17) 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.)

The input of the program is the spectrum $S(Z)$ and the output is the factor $A(Z)$, a function with the spectrum $S(Z)$. I mention here that in later chapters of this book, the factor $A(Z)$ is known as the inverse Prediction-Error Filter (PEF). In the Wilson-Burg code below, $S(Z)$ and $A(Z)$ are Z-transform polynomials but their lead coefficients are extracted off, so for example, $A(z) = (a_0) + (a_1 Z + a_2 Z^2 + \cdots)$ is broken into the two parts `a0` and `aa`.

```
module wilson { # Wilson's factorization
  use helicon
  use polydiv
  integer,                          private :: n
  real, dimension( :), allocatable, private :: auto, bb, cc, b, c
contains
  subroutine wilson_init( nmax) {
    integer, intent (in) :: nmax;    n = nmax
    allocate ( auto( 2*n-1), bb( 2*n-1), cc( 2*n-1), b(n), c(n))
    }
  subroutine wilson_factor( niter, s0, ss, a0, aa, verb) {
    integer,       intent( in)  :: niter # Newton iterations
    real,          intent( in)  :: s0    # autocorrelation zero lag
    type( filter), intent( in)  :: ss    # autocorrelation, other lags
    real,          intent( out) :: a0    # factor, zero lag
    type( filter)               :: aa    # factor, other lags
    logical,       intent( in)  :: verb
    optional                    :: verb
    real                        :: eps
    integer                     :: i, stat
    auto = 0.;  auto( n) = s0;  b( 1) =1.       # initialize
    auto( n+ss%lag) =  ss%flt                   # autocorrelation
    auto( n-ss%lag) =  ss%flt                   # symmetrize input auto.
    call helicon_init( aa)                      # multiply polynoms
    call polydiv_init( 2*n-1, aa)               # divide   polynoms
    do i = 1, niter {
       stat= polydiv_lop(.false.,.false., auto, bb)  # bb = S/A
       stat= polydiv_lop( .true.,.false., cc,   bb)  # cc = S/(AA')
       b( 2:n) = 0.5*( cc( n+1:2*n-1  ) +
                       cc( n-1:1    :-1)) / cc( n)     # b = plusside(1+cc)
       eps = maxval( abs( b(2:n)));   # "L1 norm"
       if (present (verb)) { if (verb) write (0,*) i, eps}
       if( eps < epsilon( a0))  break                  # convergence
       stat= helicon_lop( .false., .false., b, c)      # c = A b
       aa%flt = c( 1+aa%lag)                           # put on helix
       }
    a0 = sqrt( cc( n))
    }
  subroutine wilson_close () {
    deallocate( auto, bb, cc, b, c)
    call polydiv_close()
    }
}
```

EXERCISES:

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

$$\begin{array}{rrr} -1 & -4 & -1 \\ -4 & 20 & -4 \\ -1 & -4 & -1 \end{array}$$

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

2 Fomel's factorization: A simple trick to avoid division in square root computation is to run Newton's method on the inverse square root instead. The iteration is then $R' = \frac{1}{2}R(3 - R^2X^2)$ where R converges (quadratically) to $1/\sqrt{X^2}$. To get the square root, just multiply R by X^2. This leads to a reciprocal version of the Wilson-Burg algorithm. $A'/A + \bar{A}'/\bar{A} = 3 - A\bar{A}S$ Here is how it can work: Construct an inverse autocorrelation — for example, an ideal isotropic smoother; make a guess for A (min-phase roughener); iterate: (1) compute $3 - AA * S$, (2) take its causal part, (3) convolve with A to get A'. Each iteration involves just 3 convolutions (could be even done without helix).

4.5 HELIX LOW-CUT FILTER

When an image is spoiled by excessive low frequencies, we can cut them out by taking a gradient or a helix derivative.

Neither the gradient nor the helix derivative has an adjustable cutoff frequency. A slightly more complicated form allows us to build a lowcut filter with an adjustable cut-off frequency. Recall the one-dimensional causal lowcut filters (1.26) that we invented earlier. These can be thought of as the solution to a spectral factorization problem:

$$\frac{\omega^2}{\omega^2 + \epsilon^2} \quad \approx \quad \frac{1 - Z^{-1}}{1 - \rho Z^{-1}} \; \frac{1 - Z}{1 - \rho Z} \tag{4.18}$$

With this idea, we can use the helix to build a two-dimensional lowcut filter. We use a numerator $\mathbf{H}$ that is a causal factor of $-\nabla^2$ and we use a denominator that is a causal factor of $-\nabla^2 + \epsilon^2$.

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 the lead coefficient 1.79, so it seems the "zero" at zero frequency actually only drops 90% of the way to zero. Thus, although I coded a filter response to be the factorization of $-\nabla^2/(\nabla^2 + \epsilon^2)$, I then abandoned it. Instead, I choose the depth of the hole at zero frequency by adjusting the number of filter coefficients in the representation of the helix derivative.

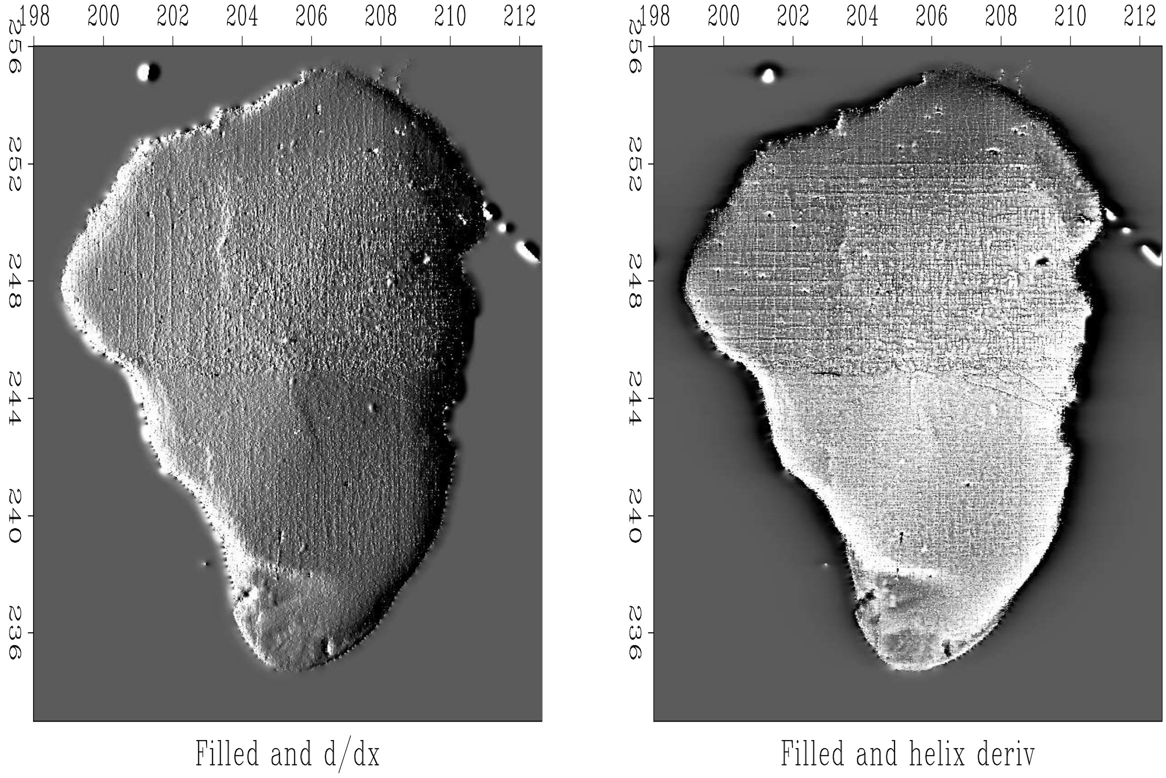

Figure 4.9: The depth of the Sea of Galilee roughened with a gradient and roughened with a helical derivative. hlx-helgal [ER,M]

Figure 1.9 shows the application of the helix derivative to the Sea of Galilee. In experimenting with preparing this figure I found it best to use a filter with a whopping 128 points. With smaller numbers of points, the middle of the lake tended to be a dome (or bowl) just like the lake itself. The reason is that the helix derivative is not fully eliminating the zero frequency. If computation time were a burden, we might simply prefer to begin from a smaller helix derivative representation but then artificially reduce the size of the zero lag input (or boost the other values).

An operator for applying the helix filter is `helderiv` on page 22.

```
module helderiv {
use lapfac
use helicon
type( filter), private :: aa
#%  _init( n1, na)
integer, intent (in) :: n1, na
aa = lapfac2( 0.001, n1, na)
call helicon_init( aa)
#% _lop (pp, qq)
integer stat1
stat1 = helicon_lop( adj, .false.,  pp, qq)
#% _close
call deallocatehelix( aa)
}
```

Figure 1.10 shows the application of the helix derivative to a medical X-ray. Experimentation with this figure showed that the helix filter need not be long (16 points was enough) but, since the inputs were all positive numbers, the output was best displayed with its mean removed because the short 16 point filter did not remove the mean.

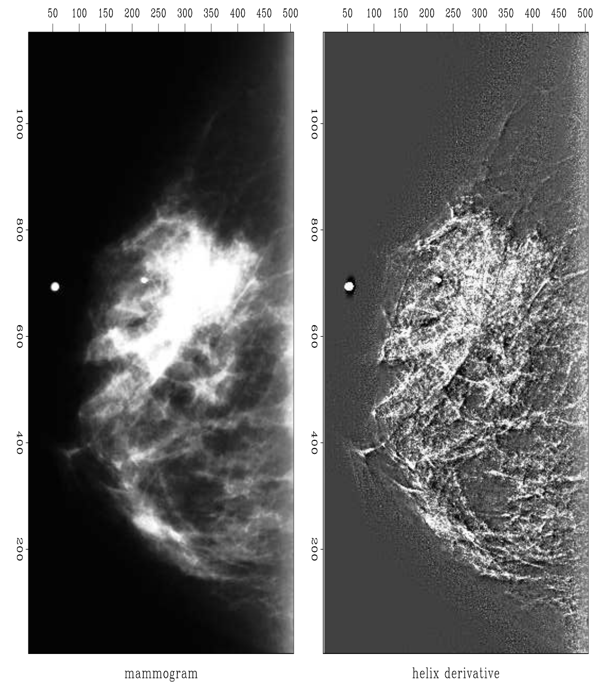

Figure 4.10: Mammogram (medical X-ray). The white circles are metal foil used for navigation. The cancer is the "spoked wheel." hlx-mam [ER,M]

4.6 THE MULTIDIMENSIONAL HELIX

Till now the helix idea was discussed as if it were merely a two-dimensional concept. Here we explore its multidimensional nature. Our main goal is to do multidimensional convolution with a one-dimensional convolution program. This allows us to do multidimensional deconvolution with a one-dimensional deconvolutional program which is "magic", i.e. many novel applications will follow.

We do multidimensional deconvolution with causal (one-sided) one-dimensional filters. Equation (1.7) shows such a one-sided filter as it appears at the end of a 2-D helix. Figure 1.11 shows it in three dimensions. The top plane in Figure 1.11 is the 2-D filter seen in equation (1.7). The top plane can be visualized as the area around the end of a helix. Above the top plane are zero-valued anticausal filter coefficients.

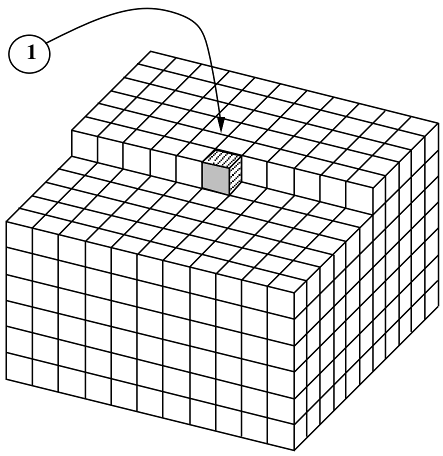

Figure 4.11: A 3-D causal filter at the starting end of a 3-D helix. hlx-3dpef [NR]

Geometrically, the three-dimensional generalization of a helix is like string on a spool, but that analogy does not illuminate our underlying conspiracy, which is to represent multidimensional convolution and deconvolution as one-dimensional.

4.7 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 `createhelixmod` on page 25. We begin here from its intricate underpinnings.

Fortran77 has a concept of a multidimensional array being equivalent to a one-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 one-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 doesn't fit very well, since it is intrinsically noncausal. If you really want noncausal filters, you will 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 one-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 one-dimensionally as `aa(ia,1,1,...)` or sometimes[3] `aa(ia)` by letting `ia` cover a large range. The formula for computing `ia` is

```
ia = i1 +a1*(i2-1) +a1*a2*(i3-1) + ...
```

for a cube stored in its normal "tightly packed" form, or if the cube were stored in an array with the same dimensions as the data, the formula for `ia` is

```
ia = i1 +n1*(i2-1) +n1*n2*(i3-1) + ...
```

The fortran compiler knows how to convert from the multidimensional cartesian indices to the linear index. We will need to do that, as well as the converse. Module `cartesian` below 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`.

```
module cartesian {       # index transform (vector to matrix) and its inverse
contains
  subroutine line2cart( nn, i, ii) {
    integer, dimension( :), intent( in) :: nn    # cartesian axes (n1,n2,n3,...)
    integer, dimension( :), intent(out) :: ii    # cartesn coords (i1,i2,i3,...)
    integer               , intent( in) :: i     # equivalent 1-D linear index
    integer                             :: axis, n123
    n123 = 1
    do axis = 1, size(  nn) {
        ii( axis) = mod( ( i-1)/n123, nn( axis)) + 1
        n123 = n123 * nn( axis)
        }
    }
  subroutine cart2line( nn, ii, i) {
    integer, dimension( :), intent( in) :: nn, ii
    integer                             :: i, axis, n123
    n123 = 1;    i = 1
    do axis = 1, size(  nn) {
        i = i + ( ii( axis)-1)*n123
        n123 = n123 * nn( axis)
        }
    }
}
```

[3]Some programming minutia: Fortran77 does not allow you to refer to an array by both its cartesian coordinates and by its linear subscript in the same subroutine. To access it both ways, you need a subroutine call, or you dimension it as `data(n1,n2,...)` and then you refer to it as `data(id,1,1,...)`. Fortran90 follows the same rule outside modules. Where modules use other modules, the compiler does not allow you to refer to data both ways, unless the array is declared as `allocatable`.

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" which is 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 `cartesian` module fails for negative subscripts. Thus we need to be careful to avoid thinking of the filter's 1.0 (shown in Figure 1.11) as the origin of the multidimensional coordinate system although the 1.0 is the origin in the one-dimensional coordinate system.

Even in one dimension (see the matrix in equation (1.4)), 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, we also need to know 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 hypecube.

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. This would waste so much space, however, that our helix filter module `helix` on page 7 instead stores the filter coefficients in one vector and their 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` on page 8, 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 `createhelixmod` on page 25 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 1.11 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 `createhelixmod` 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 their lags can be ignored too.) 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.

```
module createhelixmod {                         # Create helix filter lags and mis
use helix
use cartesian
contains
  function createhelix( nd, center, gap, na)  result( aa) {
    type( filter)                       :: aa       # needed by helicon.
    integer, dimension(:), intent(in) :: nd, na   # data and filter axes
    integer, dimension(:), intent(in) :: center   # normally (na1/2,na2/2,...,1)
    integer, dimension(:), intent(in) :: gap      # normally ( 0,   0,  0,...,0)
    integer, dimension( size( nd))    :: ii       # cartesian indexes
    integer                             :: na123, ia, ndim, nh, lag0a,lag0d
    integer, dimension(:), allocatable:: lag
```

```
        nh= 0;   na123 = product( na);   ndim = size( nd)
    allocate(  lag( na123 ) )             # filter cube size
    call cart2line ( na, center, lag0a)   # lag0a = index pointing to the "1.0"
    do ia = 1+lag0a, na123 {              #    ia  is fortran linear index.
       call line2cart( na, ia, ii)        # ii(ia) is fortran array  indices.
       if( any( ii <= gap))    next       # ignore some locations
       nh = nh + 1                        # got another live one.
       call cart2line( nd, ii, lag(nh))   # get its fortran linear index
       }
    call cart2line(  nd, center, lag0d)   # lag0d is center shift for nd_cube
    call allocatehelix( aa, nh)           # nh becomes size of filter on helix.
    aa%lag = lag(1:nh) - lag0d;           # lag = fortran_linear_index - center
    aa%flt = 0.0;             deallocate( lag)
    }
}
```

Near the end of the code you see the calculation of a parameter `lag0d`. This is the count of the number of zeros that a data-sized fortran array would store in a filter cube before 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 (1.7) shows us in two dimensions or like Figure 1.11 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 multi-dimensional hypercube `cube(n1,n2,...)` is module `box`:

```
module box {         # Convert helix filter to hypercube: cube(na(1),na(2),...)
use helix
use cartesian
contains
  subroutine boxn( nd, center, na, aa, cube) {
    integer, dimension (:), intent( in)  :: nd, center, na    # (ndim)
    type( filter),          intent( in)  :: aa
    real,    dimension( :), intent( out) :: cube
    integer, dimension( size( nd))       :: ii
    integer                              :: j, lag0a, lag0d, id, ia
    cube = 0.;                          # cube=0
    call cart2line( na, center, lag0a)  # locate the 1.0 in the na_cube.
    cube( lag0a) = 1.                   # place it.
    call cart2line( nd, center, lag0d)  # locate the 1.0 in the nd_cube.
    do j = 1, size( aa%lag) {           # inspect the entire helix
       id = aa%lag( j) + lag0d          # index = helix_lag + center_d
       call line2cart( nd, id, ii)      # ii(id) = cartesian indices
       call cart2line( na,     ii, ia)  # ia(ii) = linear index in aa
       cube( ia) = aa%flt( j)           # copy the filter coefficient
       }
    }
}
```

The `box` module is normally used to display or manipulate a filter that was estimated in helical form (usually estimated by the least-squares method).

The inverse process to `box` is to convert a fortran hypercube to a helix filter. For this we have module `unbox`. It abandons all zero-valued coefficients such as those that should be zero before the box's 1.0. It abandons the "1.0" as well, because it is implicitly present in the helix convolution module `helicon` on page 8.

```
module unbox {                          # helixfilter aa = cube(a1,a2,...)
use helix
use cartesian
contains
  function unboxn( nd, center, na, cube)  result( aa) {
    type( filter)                        :: aa
    integer, dimension( :), intent( in) :: nd, center, na      # (ndim)
    real,    dimension( :), intent( in) :: cube                # cube(a1,a2,...)
    logical, dimension( size( cube))    :: keep                # keep(a1*a2*...)
    integer, dimension( size( nd))      :: ii                  # (ndim)
    integer                             :: ic, lag0a, lag0d, i, h
    call cart2line(  na, center, lag0a)
    call cart2line(  nd, center, lag0d)
    keep = ( abs( cube) > epsilon( cube))     # epsilon is a Fortran intrinsic
    keep( lag0a) = .false.                    # throw away the 1.0.
    call allocatehelix( aa, count( keep));  h = 0
    do ic = 1, size( cube) {                           # sweep cube
           if( keep( ic) ) {                h = h + 1  # only the keepers
                call line2cart(  na, ic, ii)          # ii(ic)= indices on na
                call cart2line(  nd,     ii, i)       # i     = index   on nd
                aa%lag( h) = i - lag0d                 # lag = index - center
                aa%flt( h) = cube( ic)                 # copy coefs.
                }
           }
    }
}
```

An example of using `unbox` would be copying some numbers such as the factored laplacian in equation (1.11) into a cube and then converting it to a helix.

A reasonable arrangement for a small 3-D filter is `na=(5,3,2)` and `center=(3,2,1)`. Using these arguments, I used `createhelixmod` on page 25 to create a filter. I set all the helix filter coefficients to 2. Then I used module `box` on page 26 to put it in a convenient form for display. After this conversion, the coefficient `aa(3,2,1)` is 1, not 2. 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` 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's 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.

```
module regrid {          # convert a helix filter from one size data to another
use helix
use cartesian
contains
  subroutine regridn( nold, nnew, aa) {
    integer, dimension (:), intent (in) :: nold, nnew  # old and new helix grid
    type( filter)                       :: aa
    integer, dimension( size( nold))    :: ii
    integer                             :: i, h0, h1, h
    call cart2line( nold, nold/2, h0)   # lag of any near middle point on nold
    call cart2line( nnew, nold/2, h1)   # lag                          on nnew
    do i = 1, size( aa%lag) {           # forall given filter coefficients
       h = aa%lag( i) + h0              # what is this?
       call line2cart( nold, h, ii)     #
       call cart2line( nnew,    ii, h)  #
       aa%lag( i) = h - h1              # what is this
    }
  }
}
```

Index