Stanford Exploration Project, Report 73, November 18, 1997, pages 415–425

Nonstationarity and conjugacy: utilities for data patch work

Jon F. Claerbout[1]

keywords: *filtering, two-dimensional, linear, algorithm*

ABSTRACT

I design and explain programs for parceling a planar data set into an array of patches (small planes) that may overlap. The conjugate takes the patches and adds them back together into a data plane. Because of overlap, the original data set is not recovered unless weighting functions are used. Any weighting function can be used in any patch. I provide a seamless reconstruction with code in which you can incorporate any linear operator. Finally, I code time and space variable 2-D filtering using invariant filtering in patches, taking care that filters do not run off the boundaries of each patch and using triangular weighting functions to merge patches.

INTRODUCTION

There are many reasons for cutting data planes or image planes into pieces (patches), operating on the pieces, and then putting them back together again. The earth's dip varies with lateral location and depth. The dip spectrum and spatial spectrum thus also varies. The dip itself is important in all earth mapping, and its spectrum plays an important role (inverse covariance matrix) in estimating any earth properties. Chapter 4 of PVI (Claerbout, 1992) shows code to handle nonstationarity and uses it for dip estimation. There I did not acknowledge the conjugacy aspect of parceling data into patches (small planes). Additionally, now having significantly more experience with nonstationarity, I revise the general looping procedure and repackage code in more reusable form. The utility code here should have many applications. (I am planning uses related to monoplane-rejection regression and other 2-D prediction-error filters.)

[1]**email:** not available

PARCELING

A plane of information, either data or an image, say `wall(n1,n2)` will be parcelled up into an array of overlapping windows each of size `(w1,w2)`. You choose the number of windows by specifying `(k1,k2)`. The fraction `(w1*k1)/n1` measures overlap on the 1-axis. We can traverse all the windows, grabbing them from the wall, or putting them back, using subroutine `patch()` on this page. To run this traverse, we need only looping variables `(j1,j2)` to run from `(1,1)` to `(k1,k2)`.

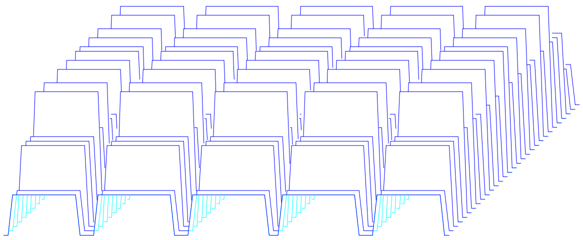
```
# patch ---- copy a patch of information from a wall into a window.
#
subroutine patch(          conj,add, j1,j2, k1,k2, wall, n1,n2, wind, w1,w2)
integer i1,i2,s1,s2,d1,d2, conj,add, j1,j2, k1,k2,       n1,n2,       w1,w2
real                                             wall( n1,n2),wind( w1,w2)
call conjnull(             conj,add,             wall, n1*n2, wind, w1*w2)

if( k2 != 1) {  s2 = 1.5 + (n2 - w2) * (j2-1.)/(k2-1.)}    else { s2= 1}
if( k1 != 1) {  s1 = 1.5 + (n1 - w1) * (j1-1.)/(k1-1.)}    else { s1= 1}

do i2= 1, w2 {  d2= i2 + s2 - 1
do i1= 1, w1 {  d1= i1 + s1 - 1
        if( conj == 0 )
                wind( i1,i2) = wind( i1,i2) + wall( d1,d2)
        else
                wall( d1,d2) = wall( d1,d2) + wind( i1,i2)
        }}
return; end
```

In subroutine `patch()` on the current page I redesigned the looping structure used in PVI. Now the patches are assured of abutting all edges of the given data plane even when `n1/w1` and `n2/w2` are not integers. With noninteger ratios, spacing of patches is slightly uneven but you will realize it does not matter after you see how easy it is to reassemble seamlessly the full plane from the patches.

The start `(s1,s2)` of the first window on the wall of data is at `(1,1)` and the last start is at the end of the wall `(n1,n2)` less the window width plus one. An exceptional case arises when the user specifies `k1=1` which means the 1-axis contains only one window. Then unless `w1==n1`, the window cannot fit both the beginning and the end of the wall so it is arbitrarily fit only at the beginning. As in any code mixing integers with floats, to guard against having a floating point number, say 99.9999, rounding down to 99 instead of up to 100, the rule is to always add .5 to a floating point number the moment before converting it to an integer.

We should always use subroutine `patch()` for parceling and avoid copying any of its messy internals into our application code. Thus when a better method of parceling comes along, we have it all in one place and can change it all at once. Figure 1 shows an example.

Figure 1: A plane of identical values after parceling into patches and then adding all the patches. Results are shown for `n1=100 w1=17 k1=5 n2=30 w2=6 k2=11` . For these parameters, there is gapping on the horizontal axis and overlap on the depth axis. jon4.patch-parcel [ER]



WEIGHTING AND RECONSTRUCTING

The conjugate of extracting all the patches is adding them back. Because of overlaps, the conjugate is not the inverse. In many applications, inverse parcelling is required, i.e. patching things back together seamlessly. This can be done with weighting functions. You can have any weighting function you wish and I will provide you a reconstruction like

$$\tilde{\mathbf{d}} \quad = \quad [\mathbf{W}_{\rm wall}\mathbf{P}'\mathbf{W}_{\rm wind}\mathbf{P}]\mathbf{d} \tag{1}$$

where $\mathbf{d}$ is your initial data, $\tilde{\mathbf{d}}$ is the reconstructed data, $\mathbf{P}$ is the parcelling operator, $\mathbf{P}'$ is conjugate parcelling (adding the patches). $\mathbf{W}_{\rm wind}$ is your chosen weighting function in the window, and $\mathbf{W}_{\rm wall}$ is the weighting function for the whole wall. You specify any $\mathbf{W}_{\rm wind}$ you like, and subroutine `mkwallwt()` below builds the weighting function $\mathbf{W}_{\rm wall}$ that you need to apply to your wall of reconstructed data to undo the nasty effects of the overlap of windows and the shape of your window-weighting function. You do not need to change your window weighting function when you increase or decrease the amount of overlap between windows because $\mathbf{W}_{\rm wall}$ takes care of it. The method is to add the weights of each window onto the wall using conjugate `patch()` on the facing page, and finally to invert the sum wherever it is non-zero. (You lose data wherever the sum is zero).

```
# make wall weights from window weights.
#
subroutine mkwallwt( k1,k2, windwt, w1,w2,  wallwt, n1,n2)
integer i1,i2,       k1,k2,         w1,w2,          n1,n2
real                       windwt( w1,w2), wallwt( n1,n2)
call null(                                 wallwt, n1*n2)
do i1= 1, k1 {
do i2= 1, k2 {
        call patch( 1, 1, i1,i2, k1,k2, wallwt, n1,n2, windwt,w1,w2 )
        }}
do i2= 1, n2 {
do i1= 1, n1 {
        if(  wallwt(i1,i2) != 0.)
             wallwt(i1,i2)  = 1. / wallwt(i1,i2)
        }}
```

```
return; end
```

No matrices are needed to show this method succeeds because data values are never mixed with one another. An equation for any reconstructed data value $\tilde{d}$ as a function of the original value d and the weights w_i that hit d is $\tilde{d} = (\sum_i w_i d)/\sum_i w_i = d$.

To demonstrate the program, I begin by making a random weighting function for each window with subroutine `mkrandwt()`. The only requirement is that the numbers be positive. Zero valued weights are OK too, but you would lose data there unless it was picked up by an overlapping window. The general strategy allows different weights in different windows. That flexibility adds clutter, however, so here we work with the same weighting function in each window.

```
# Put positive random numbers in a window.
#
subroutine mkrandwt( windwt, w1,w2)
integer i1,i2,              w1,w2,     iseed
real                windwt( w1,w2),    rand01
                                       iseed = 1992
do i1= 1, w1 {
do i2= 1, w2 {
                windwt(i1,i2) =  rand01( iseed)
                }}
return; end
```

Typically, we apply some linear operator, say $\mathbf{F}$ in each patch. Mathematically, this is:

$$\tilde{\mathbf{d}} \quad = \quad [\mathbf{W}_{\mathrm{wall}}\mathbf{P}'\mathbf{W}_{\mathrm{wind}}\mathbf{F}\mathbf{P}]\,\mathbf{d} \quad = \quad [\mathtt{Idempatch}]\,\mathbf{d} \tag{2}$$

The first code of this nature we will examine (subroutine `idempatch()`) uses the trivial identity matrix as the linear operator $\mathbf{F}$. (Later, after you understand the clutter, we go on to install 2-D filtering as $\mathbf{F}$.) To perform the weighting operation we use an adaptation of subroutine `diag()` on page 423 from PVI. To show you where to install data processing with any linear operator, I included some trivial processing, multiplying by `1.0`. This is done by subroutine `ident()` on page 423, also adapted from PVI. Since the composite operator is also a linear operator I also include code for the conjugate. Including the conjugate nearly doubles the length of the code which means that you hardly need to think about the second half which is a mirror image.

```
# idempatch --- cut up section and paste back together, idempotently.
#
subroutine idempatch( conj, add, w1,w2, k1,k2, data, n1,n2, fout)
integer i1,i2,        conj, add, w1,w2, k1,k2,       n1,n2
real                                          data( n1,n2),fout( n1,n2)
temporary real pdat(w1,w2),   copy(n1,n2)
temporary real pfou(w1,w2), wallwt(n1,n2), windwt(w1,w2)
call conjnull(        conj, add,              data, n1*n2, fout, n1*n2)
```

```
call mkrandwt(        windwt, w1,w2)
call mkwallwt( k1,k2, windwt, w1,w2,  wallwt, n1,n2)
if( conj == 0 ) {
        do i2= 1, k2 {
        do i1= 1, k1 {
                call patch( 0, 0,  i1,i2, k1,k2, data,n1,n2, pdat,w1,w2 )
                call ident( 0, 0,   1.0,  w1*w2, pdat,       pfou       )
                call diag(  0, 0, windwt, w1*w2, pfou,       pfou       )
                call patch( 1, 1,  i1,i2, k1,k2, copy,n1,n2, pfou,w1,w2 )
                }}
        call diag(          0, 1, wallwt, n1*n2, copy,       fout       )
        }
else {
        call diag(          1, 0, wallwt, n1*n2, copy,       fout       )
        do i2= 1, k2 {
        do i1= 1, k1 {
                call patch( 0, 0,  i1,i2, k1,k2, copy,n1,n2, pfou,w1,w2 )
                call diag(  1, 0, windwt, w1*w2, pfou,       pfou       )
                call ident( 1, 0,   1.0,  w1*w2, pdat,       pfou       )
                call patch( 1, 1,  i1,i2, k1,k2, data,n1,n2, pdat,w1,w2 )
                }}
        }
return; end
```

Figure 2 shows the result when a plane of identical values is passed through the `idempatch()` subroutine. The result is constant on the 2-axis which confirms that despite the random function as a weight, `idempatch()` performs as advertised and that there is adequate sampling on the 2-axis. On the 1-axis the output is constant except for being zero in gaps because the windows do not overlap on the 1-axis. Curiously, the conjugate operation produces the same result, i.e. we can do the operations in reverse order.

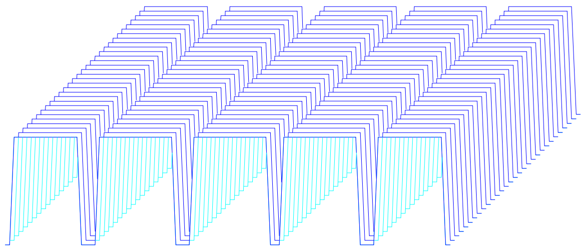


Figure 2: A plane of identical values passed through subroutine `idempatch()`. Results are shown for the same parameters as Figure 1. jon4.patch-idempatch [ER]

2-D FILTERING

A way to do time and space variable filtering is to do invariant filtering within each patch. A convenient subroutine for two-dimensional filtering is `cinloi()` on page 424 from PVI page 192 (or SEP-71). For this no-end-effect convolution routine I built a triangular weighting routine `cinloiwt()` that tapers from the center of the patch of the filter's *outputs* towards the edges. This is complicated by (1) the constraint that the filter must not move off the edge of the inputs and (2) the alignment of the input and the output. You need not study the details of `cinloiwt()` because we have seen that the shape of the weighting function is not crucial to fitting things back together. (If you insist on looking at the details, first notice in `cinloi()` on page 424 that when the filter index `b1` equals `lag1` that the output 1-axis pointer `y1` matches the input pointer `x1` and likewise for the 2-axis.)

```
# cinloiwt --- get triangle tent weights for 2-D convolution (cinloi()) output
#
subroutine cinloiwt( a1,a2, lag1,lag2, windwt, w1,w2)
integer              a1,a2, lag1,lag2,         w1,w2,     i1,i2, s1,s2, e1,e2
real                                   windwt( w1,w2)
real mid1,mid2, wide1,wide2, x, y
                            call null( windwt, w1*w2)
s1= 1+a1-lag1;  e1= w1-lag1+1;  mid1=(e1+s1)/2.;  wide1=(e1-s1+1.)/2.
s2= 1+a2-lag2;  e2= w2-lag2+1;  mid2=(e2+s2)/2.;  wide2=(e2-s2+1.)/2.

do i1= s1, e1 {     x = abs((i1-mid1)/wide1)
do i2= s2, e2 {     y = abs((i2-mid2)/wide2)
#       windwt(i1,i2) = amax1( 0., 2. - abs(x+y) - abs(x-y) )          # Cheop
        windwt(i1,i2) = amax1( 0., 1. - abs(x))  *
                        amax1( 0., 1. - abs(y))
        }}
return; end
```

In applications where triangle weights are needed on the *inputs* (or where we can work on a patch without having interference with edges) we can get "triangle tent" weights from `cinloiwt()` by setting filter dimensions and lags to unity such as shown in Figure 3. Triangle weighting functions can sum to a constant if the

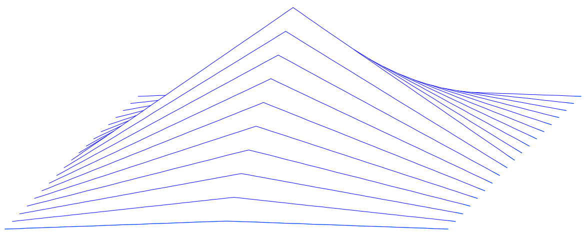


Figure 3: Window weights from `cinloiwt()` with `w1=61, w2=19` `a1=1, a2=1, lag1=1, lag2=1` . $\boxed{\text{jon4.patch-windwt}}$ [ER]

spacing is such that the midpoint of one triangle is at the beginning of the next. I imagined in 2-D that something similar would happen with shapes like Egyptian pyramids, $2 - |x - y| + |x + y|$. Instead, this property is possessed by the tent-like shapes in Figure 3 with the equation $(1 - |x|)(1 - |y|)$ as is shown in Figure 4 where, to add interest, I separated the windows by a little more than the precisely matching distance. In practice we may chose window shapes and overlaps for other reasons than the constancy of the sum of weights because `mkwallwt()` on page 417 accounts for that.

Inserting the calls to internal convolution `cinloi()` on page 424 and `cinloiwt()` on the current page into `idempatch()` on page 418 gives the subroutine `cinloip()` on

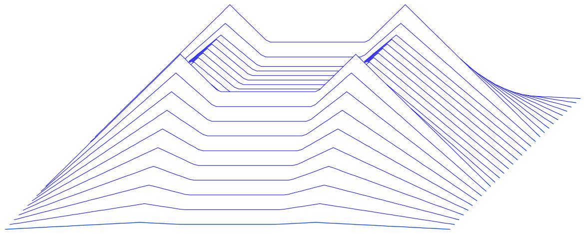


Figure 4: (Inverse) wall weights with `n1=100, w1=61, k1=2, n2=30, w2=19, k2=2` jon4.patch-wallwt [ER]

the current page that does the time and space variable filtering by invariant filtering in patches.

The basic rule of computer program management is that we should have no duplicated code. Otherwise improvements might be made to one copy but not the other, and inconsistency could grow. It is an aggravation to me, and a defect in the Fortran language that I must present to you two copies of nearly identical code, `idempatch()` on page 418 and `cinloip()` on the current page, and when you or I prepare the next application, we cannot link this code from a library, but we will need to create another copy changing the call to `ident()` to a call to our next application subroutine. I plan to study the possibility of using interlude subroutines that pick up the required variables through common and the use of other languages.

Finally is an example of filtering a plane of uniform constants with an impulse function. The impulse function is surrounded by zeros so the filter output patches are smaller than the input patches in Figure 2. In Figure 5, both axes need more window density.

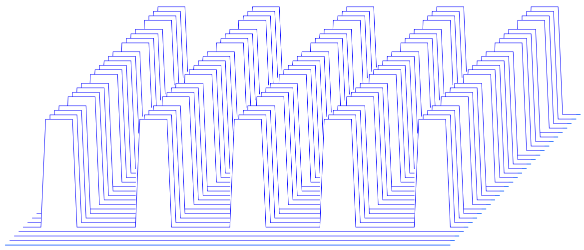


Figure 5: Output of `cinloi()` with the same parameters as Figures 1 and 2. The filter parameters are `a1=11 a2=5 lag1=6 lag2=1` . Thus, windows are centered on the 1-axis and pushed back out the 2-axis. jon4.patch-cinloip [ER]

```
# cinloip --- apply time and space variable 2-D filter, conjugate to input.
#
subroutine cinloip( conj,add, w1,w2,lag1,lag2, aa,a1,a2,k1,k2, data,n1,n2, fout)
integer i1,i2,      conj,add, w1,w2,lag1,lag2,    a1,a2,k1,k2,      n1,n2
real                                aa( a1,a2, k1,k2), data(n1,n2), fout(n1,n2)
temporary real pdat(w1,w2),   copy(n1,n2)
temporary real pfou(w1,w2), wallwt(n1,n2), windwt(w1,w2)
call conjnull(      conj,add,                             data,n1*n2,  fout,n1*n2)
call cinloiwt( a1,a2, lag1,lag2,             windwt,w1,w2)
call mkwallwt( k1,k2, windwt, w1,w2, wallwt,n1,n2)
if( conj == 0 ) {
        do i1= 1, k1 {
```

```
        do i2= 1, k2 {
                call patch(  0, 0, i1,i2, k1,k2, data,n1,n2, pdat,w1,w2 )
                call cinloi( 0, 0, lag1,lag2, a1,a2, aa(1,1,i1,i2),
                                          w1,w2, pdat,       pfou       )
                call diag(   0, 0,windwt, w1*w2, pfou,       pfou       )
                call patch(  1, 1, i1,i2, k1,k2, copy,n1,n2, pfou,w1,w2 )
                }}
        call diag(           0, 1,wallwt, n1*n2, copy,       fout       )
        }
else {  call diag(           1, 0,wallwt, n1*n2, copy,       fout       )
        do i1= 1, k1 {
        do i2= 1, k2 {
                call patch(  0, 0, i1,i2, k1,k2, copy,n1,n2, pfou,w1,w2 )
                call diag(   1, 0,windwt, w1*w2, pfou,       pfou       )
                call cinloi( 1, 0, lag1,lag2, a1,a2, aa(1,1,i1,i2),
                                          w1,w2, pdat,       pfou       )
                call patch(  1, 1, i1,i2, k1,k2, data,n1,n2, pdat,w1,w2 )
                }}
        }
return; end
```

EXERCISES

Some familiar operations can be seen in a new light when done in patches.

Smoothing program

Parcel the data. In each patch, find the mean value. Replace each value by the mean value. Reconstruct the wall.

Smoothing while filling missing data

This is like smoothing, but set window weights to zero where there is no data. Since there will be a different set of weights in each window, you will need to make a simple generalization to `mkwallwt()` on page 417.

Gain control

Divide data into patches. Compute the square root of the sum of the squares of all the data in the patch. Divide all values in the patch by this amount. Reassemble patches.

REFERENCES

Claerbout, J. F., 1992, Earth Soundings Analysis: Processing Versus Inversion: Blackwell Scientific Publications.

APPENDIX

Below are reprinted subroutines from the current version of PVI. Since the paper version of PVI was published, I modified `ident()` and `diag()` so that their outputs can be overlaid on their inputs.

```
subroutine null(     xx, n)
integer i, n;   real xx( n)
do i= 1, n
        xx(i) = 0.
return; end
```

```
subroutine conjnull( conj, add, x, nx,  y, ny )
integer ix, iy,      conj, add,    nx,     ny
real                           x( nx), y( ny )
if( add == 0 )
        if( conj == 0 )
                do iy= 1, ny
                        y(iy) = 0.
        else
                do ix= 1, nx
                        x(ix) = 0.
return; end
```

```
subroutine ident( conj, add, epsilon, n, pp,    qq   )
integer i,        conj, add,          n
real                        epsilon,    pp(n), qq(n)  # equivalence (pp,qq) OK
if( conj == 0 ) {
        if( add == 0 ) { do i=1,n {  qq(i) =         epsilon * pp(i) } }
        else           { do i=1,n {  qq(i) = qq(i) + epsilon * pp(i) } }
        }
else {  if( add == 0 ) { do i=1,n {  pp(i) =         epsilon * qq(i) } }
        else           { do i=1,n {  pp(i) = pp(i) + epsilon * qq(i) } }
        }
return; end
```

```
subroutine diag( conj, add, lambda,n,  pp,    qq)
integer i,       conj, add,       n                  # equivalence (pp,qq) OK
real                        lambda(n), pp(n), qq(n)
if( conj == 0 ) {
        if( add == 0 ) { do i=1,n {  qq(i) =         lambda(i) * pp(i) } }
        else           { do i=1,n {  qq(i) = qq(i) + lambda(i) * pp(i) } }
```

```
        }
else {  if( add == 0 ) { do i=1,n {  pp(i) =         lambda(i) * qq(i) } }
        else           { do i=1,n {  pp(i) = pp(i) + lambda(i) * qq(i) } }
        }
return; end


# CINLOI --- Convolution INternal with Lags.  Output is conjugate with INPUT.
#
subroutine cinloi( conj, add, lag1,lag2,nb1,nb2,bb,  n1,n2, xx, yy)
integer            conj, add, lag1,lag2,nb1,nb2,     n1,n2        # lag=1 causal
real                                     bb(nb1,nb2), xx(n1,n2), yy(n1,n2)
integer y1,y2, x1,x2, b1,b2
call conjnull(     conj, add,                        xx,n1*n2,  yy,n1*n2 )
if( conj == 0 )
        do b2=1,nb2 { do y2= 1+nb2-lag2, n2-lag2+1 {  x2= y2 - b2 + lag2
        do b1=1,nb1 { do y1= 1+nb1-lag1, n1-lag1+1 {  x1= y1 - b1 + lag1
                                yy(y1,y2) = yy(y1,y2) + bb(b1,b2) * xx(x1,x2)
                                }} }}
else
        do b2=1,nb2 { do y2= 1+nb2-lag2, n2-lag2+1 {  x2= y2 - b2 + lag2
        do b1=1,nb1 { do y1= 1+nb1-lag1, n1-lag1+1 {  x1= y1 - b1 + lag1
                                xx(x1,x2) = xx(x1,x2) + bb(b1,b2) * yy(y1,y2)
                                }} }}
return; end
```