Processing local-stationary data

ABSTRACT

Stationarity of a data set guarantees spatially invariant statistical properties, such as mean, variance, dip or spectrum. Many image processing schemes assume stationary data and consequently do not apply to nonstationary data. However, these processes can be generalized to apply to local-stationary data sets, a type of nonstationary data. We split the local-stationary data into its stationary components, process them individually using the stationary process, and finally merge the components into a single output.

A process or image is called stationary, if the statistics – mean, variance, spectrum – of any subset accurately describe the statistics of the entire data. Stationarity is so helpful that scientists assume it, even when a given data is only approximately stationary. Unfortunately, many interesting things in life are nonstationary and predictions and prejudices based on small subsets are notoriously unreliable.

Stationarity and nonstationarity are well-known properties in statistics and image processing (**?**). Gabor's windowed Fourier transform (**?**) and the later short-time Fourier transform (**?**) are early techniques that separate nonstationary images into stationary components. The components compactly localize events simultaneously in the time and frequency domain. In recent years, the time-frequency analysis and its applications of image compression, edge and feature detection, and texture analysis was considerably reformulated by multiresolution analysis, pyramid algorithms, and particularly wavelet transforms (Castleman, 1996). Local stationarity is a special case of nonstationarity: A local-stationary data set can be split into smaller stationary subsets (Castleman, 1996).

In this article, I first briefly define and illustrate stationarity and local-stationarity. Then I describe an algorithm that processes local-stationary data. by patching. The approach generalizes Claerbout's (1992) patching method and encapsulate it into a set of easy-to-use, object-oriented classes.

Definition of stationarity

A random process $y(\mathbf{x})$ is strict-sense stationary if the joint distribution of any set of samples does not depend on the sample's placement. Consequently, first order cumulative distribution functions, e.g., mean and variance, of $y(\mathbf{x})$ are constant. Furthermore, second order cumulative distribution functions (such as autocorrelation and autocovariance) depend only on the distance in placement, $\mathbf{x_2} - \mathbf{x_1}$. For example, a Gaussian process is strict-sense stationary since it is completely specified by its mean and covariance function.

If the mean is constant and the autocovariance is a function that depends only on the distance in placement, then we call the data wide-sense stationary or simply stationary.

Strict-sense stationary implies wide-sense stationary.

Examples of nonstationarity

A typical seismic trace, as shown in Figure 1, is nonstationary since it includes parts of low and high variance, albeit its mean is approximately zero. A shot gather, as shown in Figure 2, is nonstationary since each individual trace is usually nonstationary. Furthermore, the hyperbolic shape of the reflection events causes spatial spectral variations.

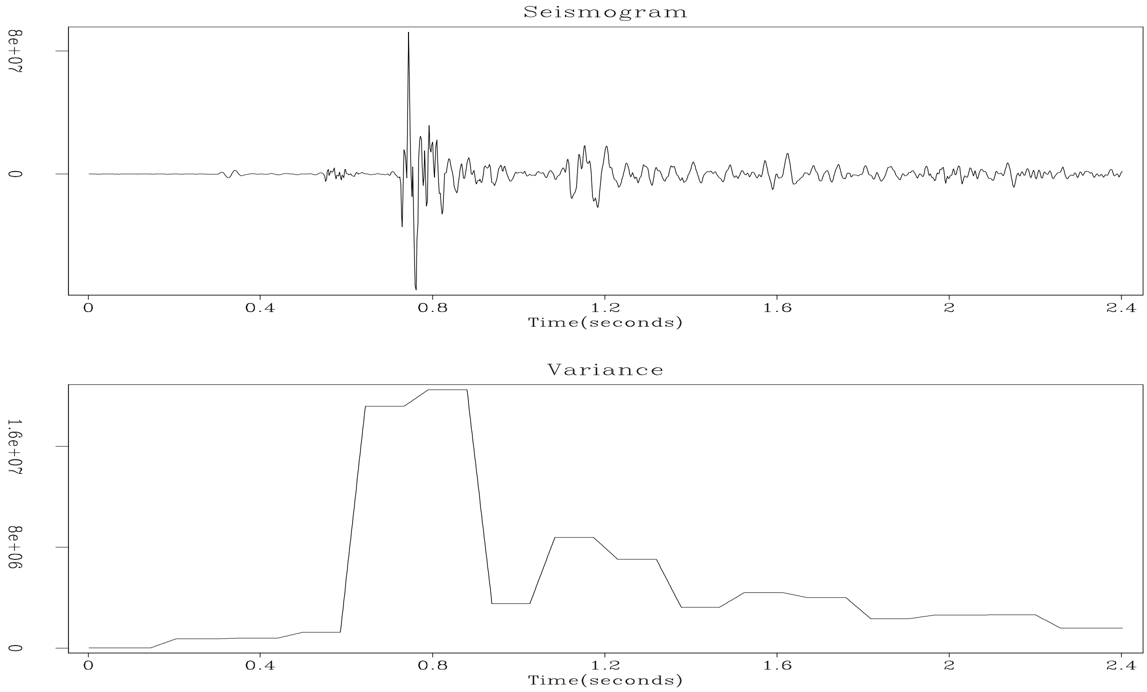


Figure 1: Nonstationary seismogram. The seismogram in the top panel has three distinct regions: The relatively quiet period before the event, the event, and the ringing of the coda. The bottom panel displays an estimate of the signal's local variance. $\boxed{\text{nStatVar}}$ [ER]

Many synthetic data sets are stationary since they are generated by simple statistical models. In practice, data sets are often nonstationary. However, the decision if a data set is assumed stationary or nonstationary depends on an application's tolerance for deviations from strict stationarity.

MATHEMATICS

A local-stationary data set is a nonstationary data set that consists of small stationary regions. For example the shot gather of Figure 2 shows approximately constant dips in small local data patches, albeit the dip varies considerably over the entire image. Local-stationarity suggests to split an image into individual stationary subsets. Each subset can then be treated as a stationary data with its individual statistical characterization. Finally, the individually

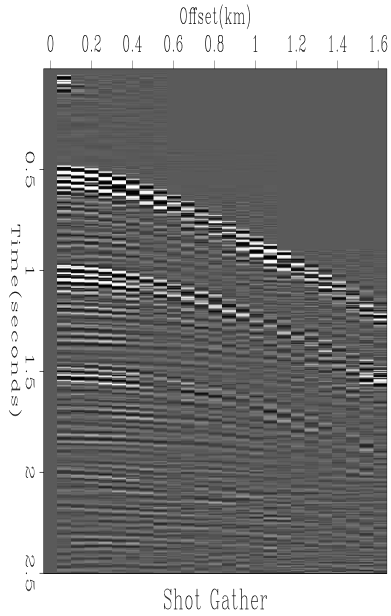


Figure 2: Shot gather. Shot gathers are nonstationary, because they show regions of distinct variance: The early quite zone, the hyperbolic reflections and the noisy coda at the bottom. gather [ER]

treated subsets are merged into a single result image (Castleman, 1996; Claerbout, 1997). My object-oriented implementation of the patching approach hides all patching details from the programmer. Additionally, my implementation offers the programmer a choice of specifying the patch size and overlap of either the input or output patches.

Splitting and merging

As illustrated by Figure 3, the local-stationary algorithm splits the given nonstationary data set $\mathbf{d}$ into predefined stationary patches $\mathbf{d}_i$:

$$\begin{bmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \\ \mathbf{d}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{P}_1 \\ \mathbf{P}_2 \\ \mathbf{P}_3 \end{bmatrix} \mathbf{d}. \tag{1}$$

Each patch is then processed individually by a local stationary operator $\mathbf{F}_i : \Omega_{d_i} \rightarrow \Omega_{m_i}$:

$$\begin{bmatrix} \mathbf{m}_1 \\ \mathbf{m}_2 \\ \mathbf{m}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{F}_1 & & \\ & \mathbf{F}_2 & \\ & & \mathbf{F}_3 \end{bmatrix} \begin{bmatrix} \mathbf{d}_1 \\ \mathbf{d}_2 \\ \mathbf{d}_3 \end{bmatrix}.$$

Finally, an interpolation operator $\mathbf{Q} = [\mathbf{Q}_1, \mathbf{Q}_2, \mathbf{Q}_3]$ combines the processed patches $\mathbf{m}_i$ to a single output quilt $\mathbf{m}$:

$$\mathbf{m} = \begin{bmatrix} \mathbf{Q}_1 & \mathbf{Q}_2 & \mathbf{Q}_3 \end{bmatrix} \begin{bmatrix} \mathbf{m}_1 \\ \mathbf{m}_2 \\ \mathbf{m}_3 \end{bmatrix}. \tag{2}$$

In summary, the patching algorithm is a compounding of windowing $\mathbf{P}^T = [\mathbf{P}_1, \mathbf{P}_2, \mathbf{P}_3]$, local imaging $\mathbf{F} = Diag(\mathbf{F}_1, \mathbf{F}_2, \mathbf{F}_3)$, and merging $\mathbf{Q}^T = [\mathbf{Q}_1, \mathbf{Q}_2, \mathbf{Q}_3]$:

$$\mathbf{m} = \mathbf{Q}^T \mathbf{F} \mathbf{P} \mathbf{d}.$$

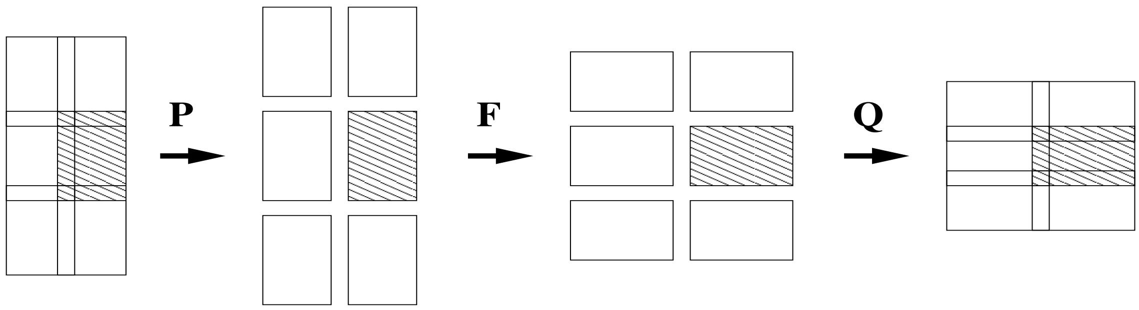


Figure 3: Patching scheme of local-stationary data. The algorithm extracts data patches from the data quilt. It images each patch by the corresponding stationary operator. Finally, it merges the individual patches to a single output quilt. The isolation of the individual patches makes this algorithm suitable for nonlinear, data-dependent operators. quiltPatch [NR]

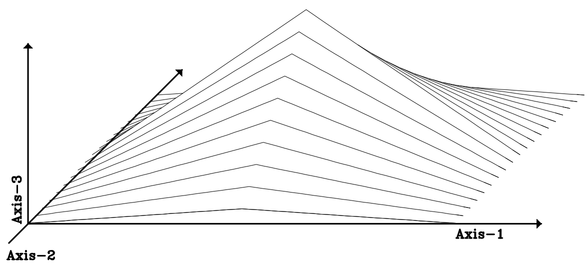
This algorithm applies each stationary operator exclusively to its appropriate patch. The individual patches can conveniently be used to estimate the desired operator, should it be dependent on its stationary input data. If, for example, the operators $\mathbf{F}_i$ are to deconvolve an unknown local blur, the isolated stationary patches conveniently permit the estimation of the optimal coefficients for the local deconvolution filter. However, this algorithm has to manipulate the space information of its various patches and quilts carefully: a task, the vector spaces of my Java software library *Jest* excel at.

If the operator $\mathbf{F}$ has identical domain and range, the initial windowing step (1) and the final merging step (2) could be implemented by adjoint operators: a column of truncation operators $\mathbf{P}_i$ and a row of zero padding operators $\mathbf{Q}_i = \mathbf{P}_i^T$. In general, the internal stationary operators $\mathbf{F}_i$, however, causes the range and domain of the splitting and interpolation operation to differ. Furthermore, a row of zero-padding operators amounts to stacking of the individual patches into the output quilt. To avoid visible boundaries at the edges of the patches, we usually prefer more sophisticated interpolation schemes.

Interpolation

If done carelessly, the merging of individually processed patches in equation (2) can cause visible patch boundaries in the final image. I implemented Claerbout's(1994) interpolation scheme, which linearly interpolates overlapping patches.

Figure 4: Patch weight. The weight function is applied to individual patches to prepare the patch for the linear interpolation of neighboring, overlapping patches. patchwt [ER]



Interpolation requires the application of weights to each data element. The weighting is easily implemented as the element-wise scaling of two vectors: one contains the data that is to be interpolated, the other one each element's corresponding weight. Algebraically, weighting of a vector amounts to a matrix multiplication by a diagonal matrix . The diagonal elements are the elements of the weight vector.

To interpolate the area of overlap linearly, each patch $\mathbf{m}_i$ is weighted by a patch $\mathbf{w}_i$ of identical size but tent-like amplitudes (Figure 4). Equation (3) expresses the weighting as the multiplications of the diagonal matrices $\mathbf{W}_i$. After zero padding and stacking the weighted patches by the row of $\mathbf{Q}_i$ operators, the operator $\mathbf{W}_m$ is another weighting operator that compensates for the variable amount of contributions every output element received:

$$\mathbf{m} = \mathbf{W}_m \begin{bmatrix} \mathbf{Q}_1 & \mathbf{Q}_2 & \mathbf{Q}_3 \end{bmatrix} \begin{bmatrix} \mathbf{W}_1 & & \\ & \mathbf{W}_2 & \\ & & \mathbf{W}_3 \end{bmatrix} \begin{bmatrix} \mathbf{m}_1 \\ \mathbf{m}_2 \\ \mathbf{m}_3 \end{bmatrix} \tag{3}$$

How do we compute the operator $\mathbf{W}_m$? The same row of zero-padding operators $\mathbf{Q}_i$ that add the weighted patches $\mathbf{m}_i$ in equation (3) adds the patch weights into a data quilt $\hat{\mathbf{w}}_m$.

$$\hat{\mathbf{w}}_m = \begin{bmatrix} \mathbf{Q}_1 & \mathbf{Q}_2 & \mathbf{Q}_3 \end{bmatrix} \begin{bmatrix} \mathbf{W}_1 & & \\ & \mathbf{W}_2 & \\ & & \mathbf{W}_3 \end{bmatrix} \begin{bmatrix} \mathbf{1} \\ \mathbf{1} \\ \mathbf{1} \end{bmatrix} \tag{4}$$

The elements of $\hat{\mathbf{w}}_m$ measure the weighted contributions each quilt element received. To balance the variable number of contributions, each element of the output quilt is scaled by the inverse of the corresponding element of the weight quilt $\hat{\mathbf{w}}_m$. This final weighting ensures that all weights applied to a single output sample sum up to one. Algebraically, the weighting is again expressed as a matrix multiplication by a diagonal matrix $\mathbf{W}_m$ in equation (3). The weights $\mathbf{W}_i$ and $\mathbf{W}_m$ can be interpreted as a heuristic preconditioning by row and column weighting (Golub and Van Loan , 1989).

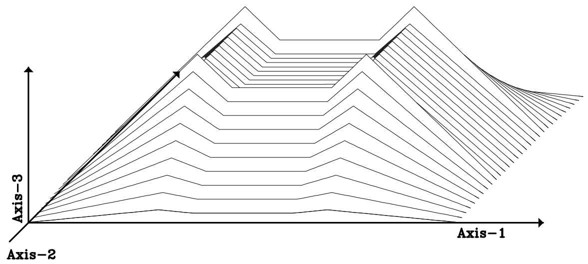
A user can modify the interpolation by supplying any alternative weight function $\mathbf{w}_i$ for the individual patches. The algorithm will apply the correct corresponding weight operator $\mathbf{W}_m$. Figure 5 shows the weight vector $\hat{\mathbf{w}}_m$ that corresponds to four slightly separated tent weights $\mathbf{w}_i$ as shown in Figure 4.

Figure 6 demonstrates the effect of the weighted interpolation on the example in Figure 1. The top panel shows the input trace. The bottom panel shows an estimate of the signal's time variable variance. The weight $\mathbf{w}_m$, shown in the second panel, correct the brute stack, shown in the third panel, to the final result in the bottom panel.

IMPLEMENTATION AND INTERFACE

The programmer interface of the Patch classes successfully encapsulates and hides the somewhat complex implementation and invocation details of the underlying patch enumeration, windowing, weighting, and merging. The program `NStatVarPush` on the following page

Figure 5: Superposition of overlapping patch weights. The shown quilt vector is used to balance the weighted patch contributions of the linear interpolation of neighboring, overlapping patches. The shown quilt vector is the result of zero padding and stacking of four slightly separated patch weights. The amplitudes indicate the number of weighted contributions each image quilt element received. The weighting operator $\mathbf{W}_m$ applies the inverses of these weight quilt elements to the corresponding image quilt elements. quiltwt [ER]



shows the instantiation and invocation of the patch method. In general, The programmer chooses among two patch operators: PatchPush or PatchPull. PatchPush defines the patches as subsets of the input, while PatchPull defines them as subsets of the output. The initialization of the patch operators requires three arguments: the patch size and overlap, the input or output space, and an operator factory object. The patch operators implement Jest's operator interface (Schwab and Schroeder, 1997) and,consequently, the standard `image()` method accepts the input quilt and returns the processed output quilt.

```
package nstat.var;
import  jam.operator.*;
import  rsf.vector.*;
import  rsf.operator.*;
import  sep.io.ArgumentKeeper;
import nstat.patch.*;

public class NStatVarPush {
  public static void main(String[] args) {
    ArgumentKeeper argKeeper = new ArgumentKeeper(args);    // read args
    Rsf   iput = RsfFactory.newRsf     (System.in         ); // read input quilt
    float olap = argKeeper.getFloat    ("overlap"  , "0.5");
    int[] psze = argKeeper.getIntArray("patchsize", "321");

    OperatorFactory  opFac = new StatVarFactory();
    PatchPushOp      pchOp = new PatchPushOp((RsfSpace) iput.getSpace(),
                                          olap, psze, opFac);
    Rsf oput = (Rsf) pchOp.image(iput);                    // image quilt
    oput.write("pushOput.H"); iput.write("pushIput.H");   // write output quilt
  }
}
```

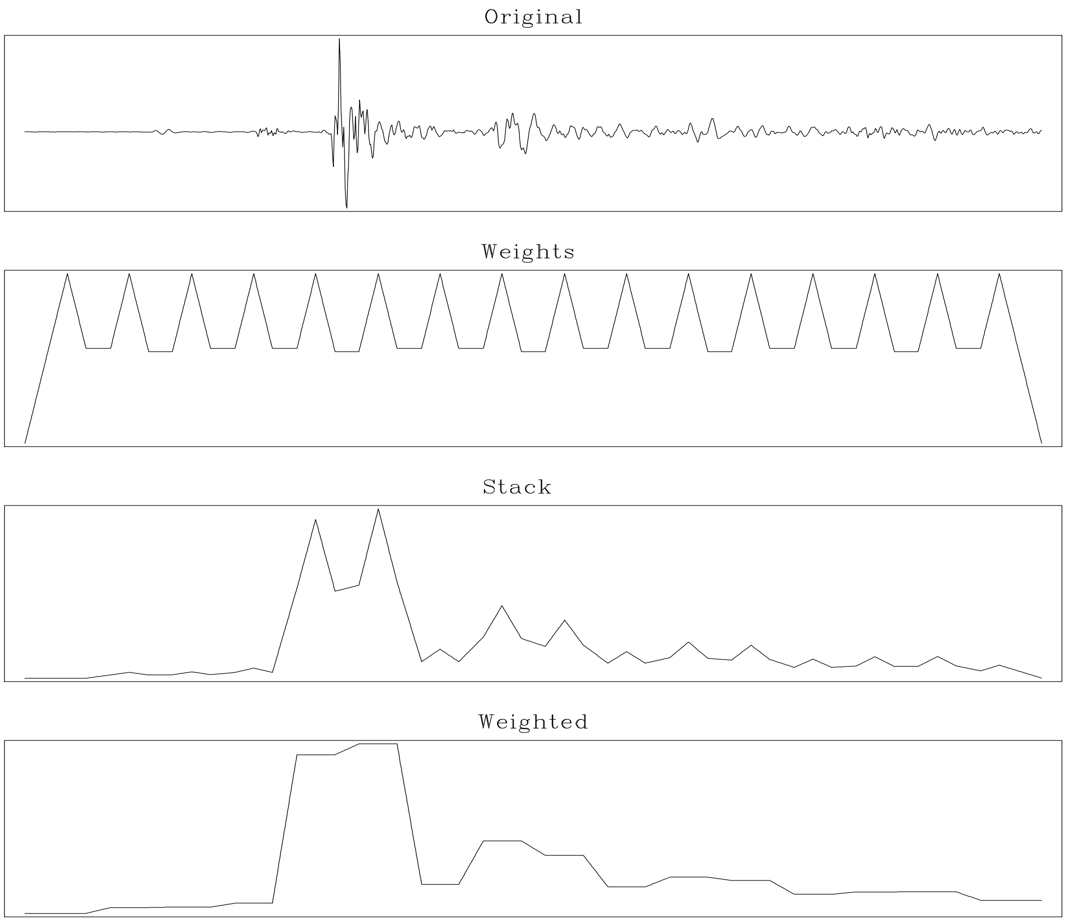


Figure 6: Variance computation of nonstationary seismogram. The top panel shows a seismic trace. The bottom panel shows the local variance of that trace. The two traces in the center panels display intermediate results of the algorithm. The quilt weight in the second panel is used to normalize the weighted stack in the third panel. $\boxed{\text{nStatVarEnum}}$ [ER]

Table 1: Some Operator Factory methods (syntax, effect).

`xspc = fac.createDomain(yspc)`	creates domain space for given range space
`yspc = fac.createRange( xspc)`	creates range space for given domain space
`oper = fac.getOp(xspc,yspc)`	creates operator given domain and range

The patch operators require an operator factory to generate the stationary operators needed to process each stationary patch. Such a factory is usually easily programmed, because most of its methods are simply wrappers for the instantiation methods of the stationary operator. Why are such wrappers necessary? A factory standardizes instantiation methods, which cannot be standardized directly by inheritance from an abstract interface (**?**). Additionally, the operator factory supplies methods that return the domain given a range and vice versa. Table 1 shows the basic methods of the operator factory interface.

The programmer-friendly patch operators encapsulate the tedious and error-prone computations that split, process, and merge nonstationary data sets[1]. Internally, the patch operators invoke a class called PatchEnumeration that organizes the patch processing. The enumeration computes the actual patch location. The patch size is kept constant, but the overlap may slightly vary in order to fit the given patch size into the given input quilt size. The patch location is defined by the patch size and a patch anchor, the lowest-index vertex of the patch on the quilt space. The PatchEnumeration object directs a Window operator $\mathbf{P}_i$ to extract the various patches. For efficiency, the Window operator can change its domain and range without being instantiated. The operator handles minor rounding errors in the spaces' offset and increment intelligently, so that the patch's space remains a subspace of the quilt space. For each extracted patch, the patch operator requests a corresponding operator $\mathbf{F}_i$ from the operator factory. After processing the patch, the patch operator creates an interpolation operator $\mathbf{Q}_i$. The interpolation operator weights the output patch and adds it to the output quilt $\mathbf{m}$. The same interpolation operator images an identity vector and adds it to the weight quilt $\hat{\mathbf{w}}_m$. After the last patch is processed, the patch operator divides the output quilt's elements by the corresponding elements of the weight quilt. The encapsulation of the patch operator greatly owes its implementation to Jest's novel and careful formulation of domains, ranges, and vector spaces.

EQUIVALENCE OF FILTER- AND DATA INTERPOLATION

To ensure smoothly space-variant filtering, we overlap neighboring patches, process each patch individually, and interpolate the resulting patch vectors. This patching operation may look ad hoc: Shouldn't we apply a spatially varying operator instead? In the case of a local-stationary filtering operation, the interpolation of filters and the interpolation of data patches approximately yield the same result, if the interpolation weights vary slowly with respect to

[1]Most programmers and readers do not want to know the details of the patch operator implementation and can skip this paragraph. But for anyone who wants to take a peek ...

the filter size. Under such circumstances, the sequence of convolution and interpolation can be commuted. Similar commutations should be permissible for other stationary operators.

I will demonstrate the equivalence of data and filter interpolation on a simple deconvolution example. The local-stationary data $z_0(\mathbf{x})$ has two stationary components. We find a normalized, smoothly varying, weighting function $w(\mathbf{x})$ that isolates the two stationary components. If we define the deconvolution filters for the two stationary components as $f_1(\mathbf{x})$ and $f_2(\mathbf{x})$, we can compute two filtered images

$$\begin{aligned} z_1(\mathbf{x}) &= f_1(\mathbf{x}) \star z_0(\mathbf{x}) \\ z_2(\mathbf{x}) &= f_2(\mathbf{x}) \star z_0(\mathbf{x}). \end{aligned} \tag{5}$$

The interpolation of the two filtered images yields the interpolation of the two deconvolved subsets:

$$\begin{aligned} z(\mathbf{x}) &= w(\mathbf{x}) z_1(\mathbf{x}) + (1 - w(\mathbf{x})) z_2(\mathbf{x}) \\ &= w(\mathbf{x})(f_1(\mathbf{x}) \star z_0(\mathbf{x})) + (1 - w(\mathbf{x}))(f_2(\mathbf{x}) \star z_0(\mathbf{x})). \end{aligned} \tag{6}$$

If, however, $w(\mathbf{x})$ is slowly varying compared to the size of the filters f_1 and f_2, the weight function can be assumed to be locally constant. Under this assumption, multiplication and convolution commute. Now, the filter – not the data – is interpolated:

$$\begin{aligned} z(\mathbf{x}) &= z_0(\mathbf{x}) \star [w(\mathbf{x}) f_1(\mathbf{x}) + (1 - w(\mathbf{x})) f_2(\mathbf{x})] \\ &= z_0(\mathbf{x}) \star f(\mathbf{x}), \end{aligned} \tag{7}$$

where

$$f(\mathbf{x}) = w(\mathbf{x}) f_1(\mathbf{x}) + (1 - w(\mathbf{x})) f_2(\mathbf{x}).$$

In terms of the two scales involved, the filter f varies smoothly on every scale, while the filter components f_1 and f_2 are locally constant.

DISCUSSION

Processing local-stationary data amounts to splitting, processing, and then merging stationary patches. The idea of patching is easily explained but its software implementation is tricky.

I redesigned Claerbout's original approach (Claerbout, 1992b) in an object-oriented framework. Due to Jest's handling of domain and range, the patch operators yield a high-level user interface that simply requires patch specifications and an operator factory object. The patches are specified by patch size and overlap in the domain *or* range. The operator factory generates a stationary operator given a specific domain and range patch.

Alternative approaches to process local-stationary data exist. In particular, I considered Claerbout's formulation of an underdetermined inverse problem that estimates an operator for each input sample. The discussion of filter versus data interpolation, discussed above, indicates that, for slowly varying local-stationary data, such a variable operator approach is equivalent to the chapter's patching scheme.

ACKNOWLEDGMENTS

Jon Claerbout's (1997) Fortran implementation provided the basic approach and test examples. Joel Schroeder implemented a preliminary version of the patching algorithm within the Jest framework.

REFERENCES

Castleman, K. R., 1996, Digital image processing: Prentice-Hall, Inc., New Jersey.

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

Claerbout, J. F., 1992b, Nonstationarity and conjugacy: Utilities for data patch work: SEP–**73**, 391–400.

Claerbout, J. F., 1994, Applications of Three-Dimensional Filtering: Stanford Exploration Project.

Claerbout, J. F., 1997, Geophysical exploration mapping: Environmental soundings image enhancement: Stanford Exploration Project.

Golub, G. H., and Van Loan, C. F., 1989, Matrix computations: The Johns Hopkins Press Ltd., London.

Schwab, M., and Schroeder, J. M., 1997, Jest, an optimization library: `http://sepwww.stanford.edu/sep/matt/jest`.