Stacking operators

Many geophysical algorithms can be cast in the form of a summation over some trajectory in a 2-D space. Familiar examples are NMO-stack, Kirchhoff migration and LMO-stack. In an earlier chapter **??** we defined moveout operators that performed NMO, LMO etc. All that we need to implement the desired algorithms is to combine these moveout operators with a stacking operator that sums over one dimension. For example NMO-stack is NMO followed by summation over the offset dimension. In our C++ classes all of these stacking operators can be coded using a chain of a moveout operator and a stacking operator `fopStack` (see man page **??**).

There is no "official" definition of which operator of an operator pair is the operator itself and which is the adjoint. We like to think of the modeling operation itself as *the* operator. In this case the forward operation for NMO-stack is to begin from a model that is the stacked zero-offset trace and spray this trace to all offsets. The adjoint operation is the common-midpoint stacking procedure. On the other hand, the industry machinery keeps churning away at many processes that have well-known names, so people often think of one of them as *the* operator. Industrial data-processing operators are typically adjoints to modeling operators. In this document we always try to define the forward operator as the modeling operator.

LINEAR MOVEOUT AND STACKING

[1]The linear moveout and stacking of a data space is done by the `fopLMOstk` operator. This operator is constructed as a chain of two operators: `fopLMO` and `fopStack`. The adjoint method takes a CMP gather as input, does a linear moveout of the data and finally stacks it. The header file `foplmostk.h` (page 2) shows how the class is define and `foplmostk` (page 2) implements the constructors. Figure 1 shows an LMO stacked CMP gather.

```
class fopLMOstk : public fopChain {
  public:

    // Contructors for fopNMOstk
    // Take time Axis, offset Axis, and slowness (1./vel)
    // parameter, as either a value or an array of
    // values.
    fopLMOstk(Axis &, Axis &, float);
    fopLMOstk(Axis &, Axis &, float a[]);
```

---

[1] Hector Urdaneta

```
};


fopLMOstk::fopLMOstk(Axis & t, Axis & x, float s) {

num=2;
ops = new foperator*[2];
ops[0] = new fopLMO( t, x, s );
ops[1] = new fopStack( x, 1);

}
fopLMOstk::fopLMOstk(Axis & t, Axis & x, float s[]) {

num=2;
ops = new foperator*[2];
ops[0] = new fopLMO( t, x, s );
ops[1] = new fopStack( x, 1);

}
```

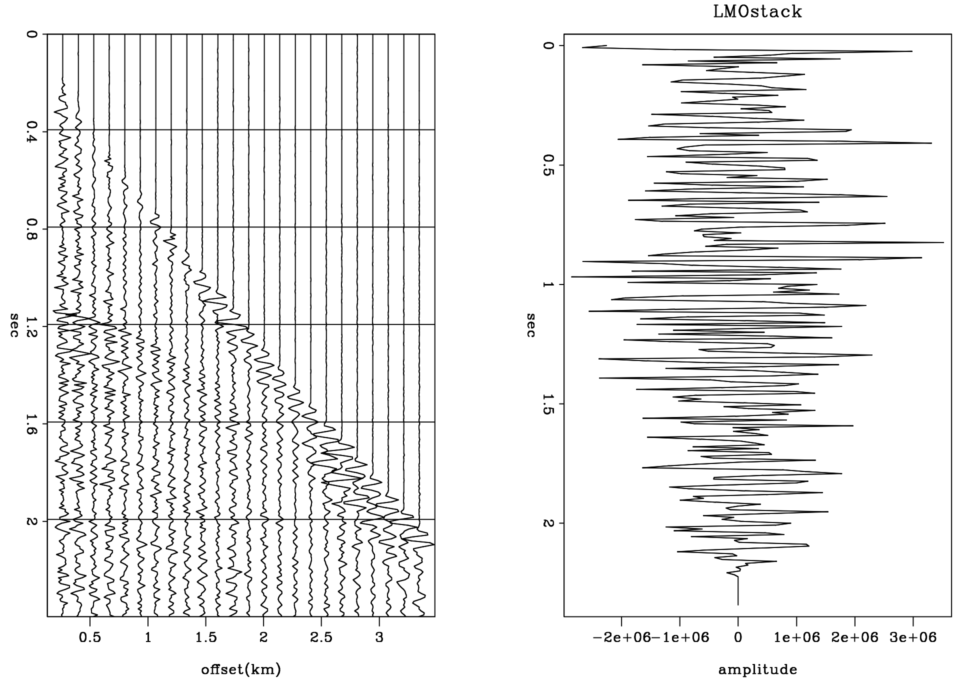


Figure 1: A common midpoint gather before (left) and after (right) LMO stacking at water velocity. stacking-wglmostk [NR]

NMO AND STACK

[2] NMO stack is the process of applying normal moveout to all the traces in a CMP gather and then summing the traces. We implement NMO stack as a composition of two previously defined operators, the NMO operator “fopNMO” and the stacking operator “fopStack”. To avoid the nuisance of having to create two operators each time we want to perform NMO and stack we create a new class, derived from the

---

[2] Dave Nichols

fopchain class, that builds a chain of the two operators for us. The code for this is almost exacty the same as the code for the LMO stack class (page 2) except that the fopLMO operator is replaced by a `fopNMO` operator (page **??**). Note that, as usual, the forward operation is the modeling operation. The adjoint is the familiar CMP stack.

Let $\mathbf{A}$ denote NMO-stack modeling (modeling from one zero offset stack trace to a full gather with hyperbolic moveout), and let the NMO-stack be defined by invoking the adjoint of the operator. Figure 2 shows a model trace $\mathbf{x}$ on the left and the result of applying the forward operator $\mathbf{A}\,\mathbf{x}$ on the right, this is our synthetic CMP gather, $\mathbf{y}$.

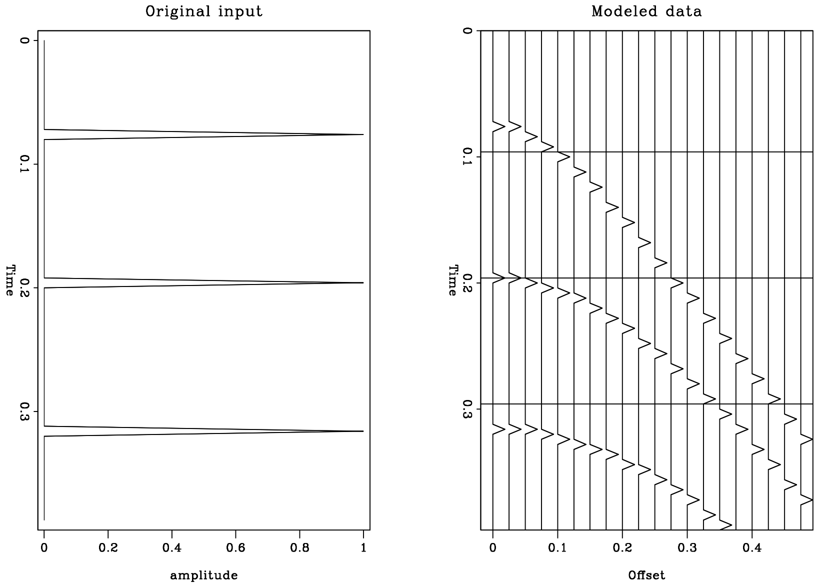


Figure 2: Left frame = input, right frame = modeling. stacking-stackout [NR]

Figure 3 shows the result of adjoint processing on the synthetic CMP gather. The left frame is $\mathbf{A}'\mathbf{y} = \mathbf{A}'\mathbf{A}\mathbf{x}$; we have not recovered our input model, $\mathbf{x}$, because the NMO-stack operator is not unitary. The right frame shows the result of modeling the processed trace $\mathbf{A}\,\mathbf{A}'\,\mathbf{y}$.

We can use a least-squares inverse solver to improve on the adjoint processing. This procedure minimizes $||\mathbf{A}\,\mathbf{x}\;-\;\mathbf{y}||^2$. Figure 4 shows the true model and several iterations of an iterative inverse procedure. After ten iterations the exact result has been recovered.

In Figure 5 the result of modeling the inverted result is shown, it completeley recovers the input to the inversion process. It is interesting to contrast this result with the inversion of the NMO operator in section **??**. In that inversion we observed a large null space in the operator and the original model was not recovered. One major reason that this does not happen with NMO-stack is that we have reduced the number of variables in the model by a factor of twenty (the number of offsets we stack over).

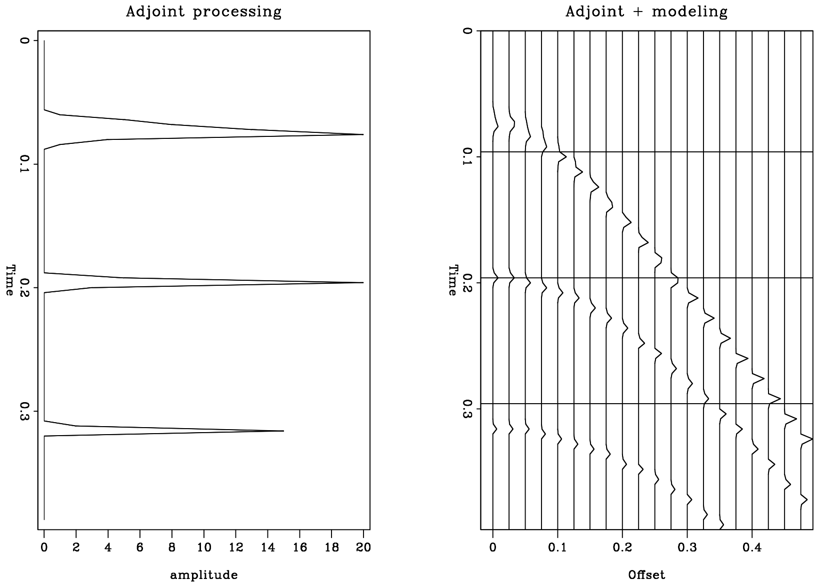


Figure 3: Left frame = adjoint processing, right frame = adjoint processing + modeling. stacking-stackadj [NR]

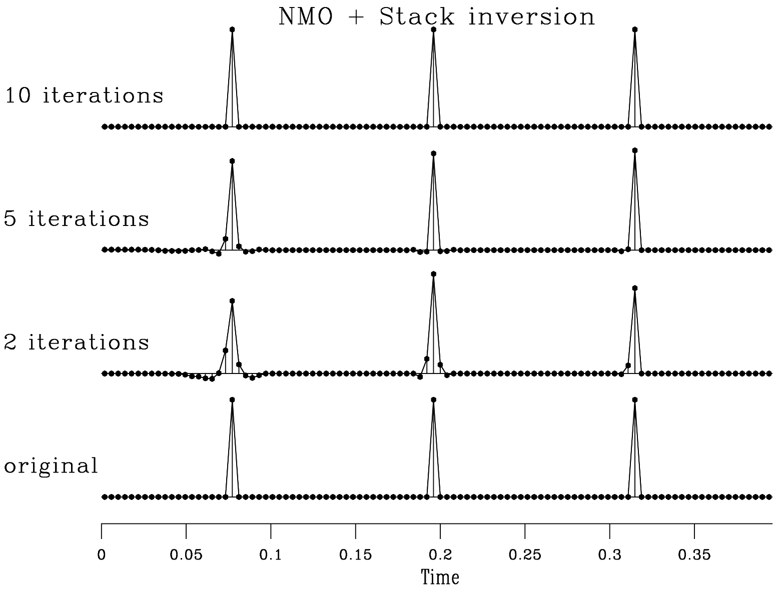


Figure 4: Iterations of inverse processing stacking-stackiters [NR]

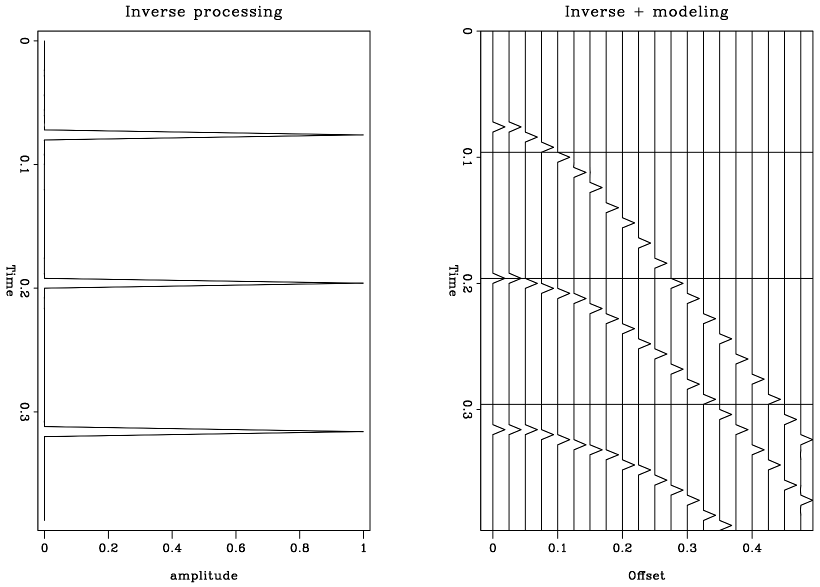


Figure 5: Left frame = inverse processing, right frame = inverse processing + modeling stacking-stackinv [NR]

KIRCHHOFF OPERATOR

[3]The Kirchhoff poststack operator (Kirchhoff moveout + Stack over offset) is implemented as a composition of two previously defined operators, the KMO operator `fopKMO` and the stacking operator `fopStack`. To avoid the nuisance of having to create two operators each time we want to perform KMO and Stack, we derive a new class `fopKMOstksingle`, from class `fopChain`. For convenience we create an intermediate class `fopKMOstksingle`, that handles only the moveout and stack for a single midpoint gather (or cube). More complicated output geometries, such as poststack 2-D panels or 3-D cubes, produce their operators by chaining the single gather operator.

```
class fopKMOstksingle : public fopChain {
  public:

    // Contructors for fopKMOstk
    // Take time Axis, offset Axis, midpoint Axis and slowness (1./vel)
    // parameter, as either a value or an array
    // values.
    fopKMOstksingle( Axis &, Axis &, float m, float a);
    fopKMOstksingle( Axis &, Axis &, float m, float a[]);

};
```

All that is required to implement this new class is to write a constructor that builds the two separate operators and stores them internally in the `fopChain` class. Application of the two operators forward and adjoint is handled by the parent class so they can inherit its behavior. We do not need to reimplement the code.

```
#include <fopkmostksingle.h>
```

---

[3] Martin Karrenbach

```
#include <fopkmo.h>
#include <fopstack.h>

// Operator to do KMO and stack (as adjoint) on a single trace.

fopKMOstksingle::fopKMOstksingle(Axis & t,Axis & x, float m, float s) {

  num = 2;
  ops = new foperator*[num];
  ops[0] = new fopKMO( t, x, s ,m );
  ops[1] = new fopStack( x, 1);
}

fopKMOstksingle::fopKMOstksingle(Axis & t,Axis & x, float m ,float s[]) {

  num = 2;
  ops = new foperator*[num];
  ops[0] = new fopKMO( t, x, s ,m );
  ops[1] = new fopStack( x, 1);
}
```

Let $\mathrm{KMOSTK_s}$ denote Kirchhoff modeling from one zero-offset trace to a full gather with hyperbolic moveout. By invoking the adjoint the operator will sum all moved-out traces into a single stacked trace. Thus `fopKMOstksingle` performs the following operation on some given input data set:

$$\mathrm{KMOSTK_s} = \mathrm{STACK}^t \quad \mathrm{KMO} \tag{1}$$

What we consider the normal Kirchhoff stack operator is here the adjoint composite operator $\mathrm{KMOSTK_S}^t$.

Poststack Kirchhoff Operator

The previously introduced class `fopKMOstksingle` can be used for building the more complicated operator which models or migrates a whole poststack panel. Again we chain operators which loop over the midpoint axes and sum the data.

The poststack Kirchhoff operator needs three axes to work on: the time axis, the unmigrated midpoint axis and the migrated midpoint axis. We construct it in the familiar manner:

```
class fopKMOstk : public fopChain {
  public:

    // Contructors for fopKMOstk
    // Take time Axis, offset Axis, midpoint Axis and slowness (1./vel)
    // parameter, as either a value or an array
    // values.
    fopKMOstk( Axis &, Axis &, Axis &, float a);
    fopKMOstk( Axis &, Axis &, Axis &, float a[]);

};
```

The actual work is done in the following operator chain:

$$\mathrm{KIRCH} = \mathrm{KMOSTK_s} \quad \mathrm{MERGE}^t \tag{2}$$

The forward input space is the "time–migrated midpoint" space. We introduce a new dimension the "unmigrated midpoint" axis, which will contain our modeling results.

An operator array is constructed from the input space, with each subspace containing one midpoint. Each of those subspaces is moved out and stacked using `fopKMOstksingle`. The final step is the stack over migrated midpoint axis to obtain the output section. The output is "time–unmigrated midpoint" space. The forward operation thus implements a Kirchhoff modeling process. The adjoint operation is to start with "time" and "unmigrated midpoint" space and applying the adjoint operator chain, ending up with the "time" and "migrated midpoint" space, a migrated dataset.

```
fopKMOstk::fopKMOstk(Axis & t,Axis & m, Axis & x, float s) {

   // make an array of kmostack operators
   fopArray * kmostkops = new fopArray( 1, m.length );

   for ( int i=0; i<m.length; i++ ){
       fopKMOstksingle * kmostack = new fopKMOstksingle( t, x, m.value(i), s );
       kmostkops->set( 0, i, kmostack );
   }
   fopMerge   * mergeop   = new fopMerge( m );
   fopAdjoint * makepanel = new fopAdjoint( mergeop );

   num = 2;
   ops = new foperator*[num];
   ops[0] = kmostkops;
   ops[1] = makepanel;
}
fopKMOstk::fopKMOstk(Axis & t,Axis & x, Axis & m ,float s[]) {

   // make an array of kmostack operators
   fopArray * kmostkops = new fopArray( 1, m.length );

   for ( int i=0; i<m.length; i++ ){
       fopKMOstksingle * kmostack = new fopKMOstksingle( t, x, m.value(i), s );
       kmostkops->set( 0, i, kmostack );
   }
   fopMerge   * mergeop   = new fopMerge( m );
   fopAdjoint * makepanel = new fopAdjoint( mergeop );

   num = 2;
   ops = new foperator*[num];
   ops[0] = kmostkops;
   ops[1] = makepanel;
}
```

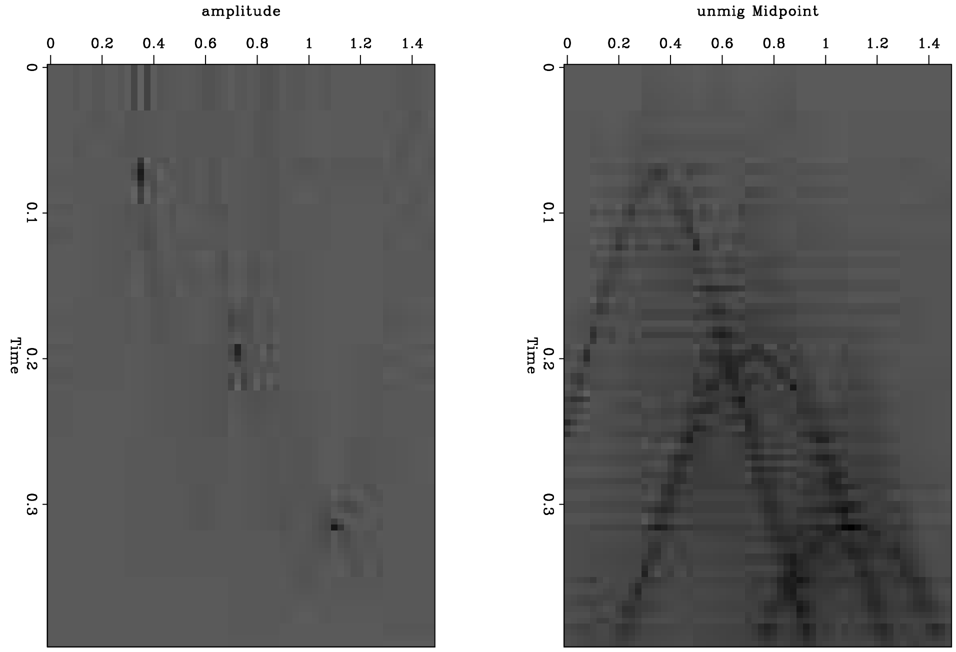
The following Figures 6 and 7 show results of Kirchhoff modeling and migration using the chained operators.

VELOCITY ANALYSIS

[4] An important transformation in exploration geophysics takes the data as a function of shot-receiver offset and transforms it to data as a function of apparent velocity. Data is summed along hyperbolas of many velocities. This important industrial process is the adjoint to another that may be easier to grasp: data is synthesized by a superposition of many hyperbolas. The hyperbolas have various asymptotes (velocities) and various tops (apexes).

---

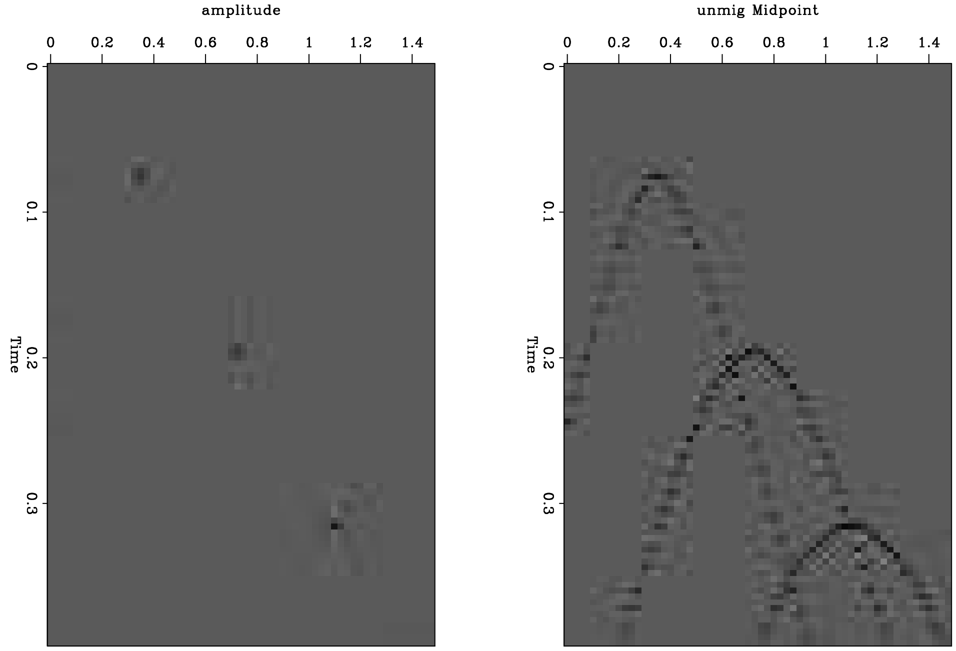
[4] Hector Urdaneta



Adjoint processing

Adjoint + modeling

Figure 6: Left frame = adjoint processing, right frame = adjoint processing + modeling. stacking-stackadjp [NR]



Inverse processing

Inverse + modeling

Figure 7: Left frame = inverse processing, right frame = inverse processing (10 iteration) + modeling. stacking-stackinvp [NR]

The construction of the velocity analysis operator, is done by chaining the NMO stack operator (see section) and the merge operator (see man pages **??**). The adjoint operator takes a set of linearly increasing slownesses values that are put into an axis. The adjoint operator applies first, for each slowness value, the NMO stack operator, which in turn, does the normal moveout and the stacking of the data. When this process is finished, we end up having one-dimensional data sets for each individual velocity. The merge operator finishes the job, putting all these one-dimensional spaces into a two-dimensional space where the new axis is a slowness axis.

A graph of inheritances for the velan operator is shown in Figure 8. Notice how you can also construct a velocity analysis operator, for the LMO operator (see next section), in the same way as for the NMO operator.

```
class fopVelan : public fopChain {
  public:
      //    time       offset    slowness
    fopVelan(Axis & t,  Axis & x,  Axis& s  );
};


fopVelan::fopVelan(Axis & t, Axis & x, Axis& s) {

int len = s.length;

// make a ( 1, len ) array of nmostack operators
fopArray * nmostkops = new fopArray(  1, len );

for( int i=0; i<len; i++ ){
    float slow = s.value(i);
    // build a single nmostack operator
    fopNMOstk * stkop = new fopNMOstk(t, x, slow );
    // put it in the array
    nmostkops->set( 0, i, stkop );
}

 // make a Merge operator
fopMerge * mergeop = new fopMerge( s );

 // make its adjoint
fopAdjoint * adjmerge = new fopAdjoint( mergeop );

// make a chain of the two
num=2;
ops = new foperator*[2];
ops[0] = nmostkops;
ops[1] = adjmerge;

}
```

The constructors for the `fopVelan` class are defined in the header file `fopvelan.h` (page 9). `fopvelan` (page 9) shows the coding of the `fopVelan` constructor.

An example of applying the `fopVelan` operator to field data is shown in Figure 9.

Figures 9 to 12 show examples of modeling, processing and inverse processing for the velocity analysis operator.

SLANT STACK AND PRECONDITIONED INVERSION

Figure missing !

Figure 8: Graph of dependencies for the velocity analysis operator. Class relationships are depicted according to the Booch Method: an arrow indicates inheritance and double lines with a solid dot connect a class with the class it contains stacking-dependencies [NR]

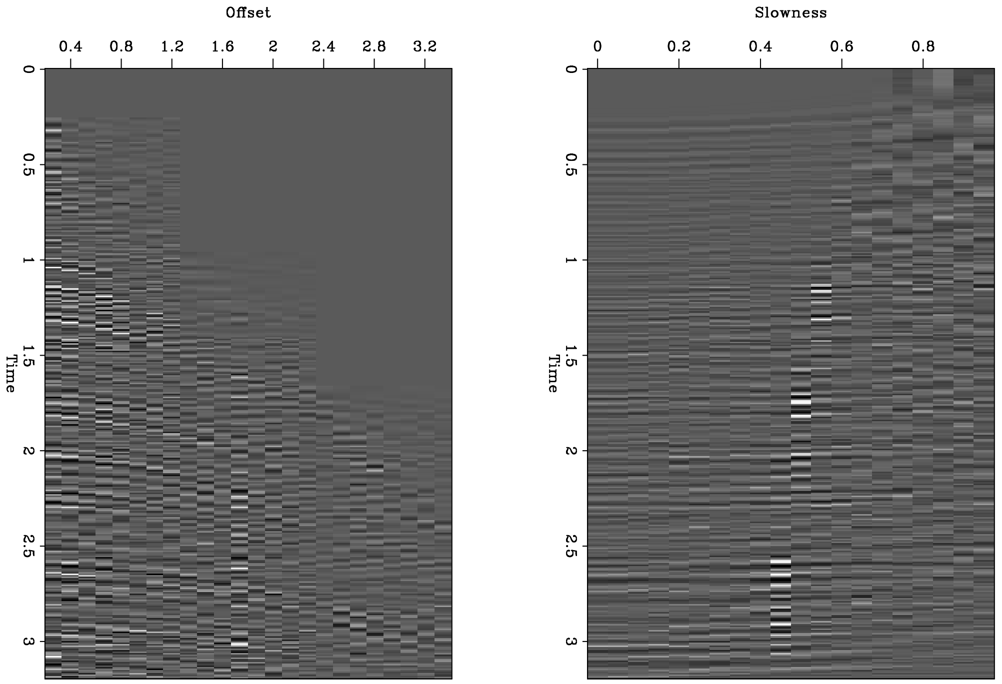


Figure 9: Transformation of data as a function of offset (left) to data as a function of slowness (right) using the `fopVelan` operator. stacking-mutvel [NR]

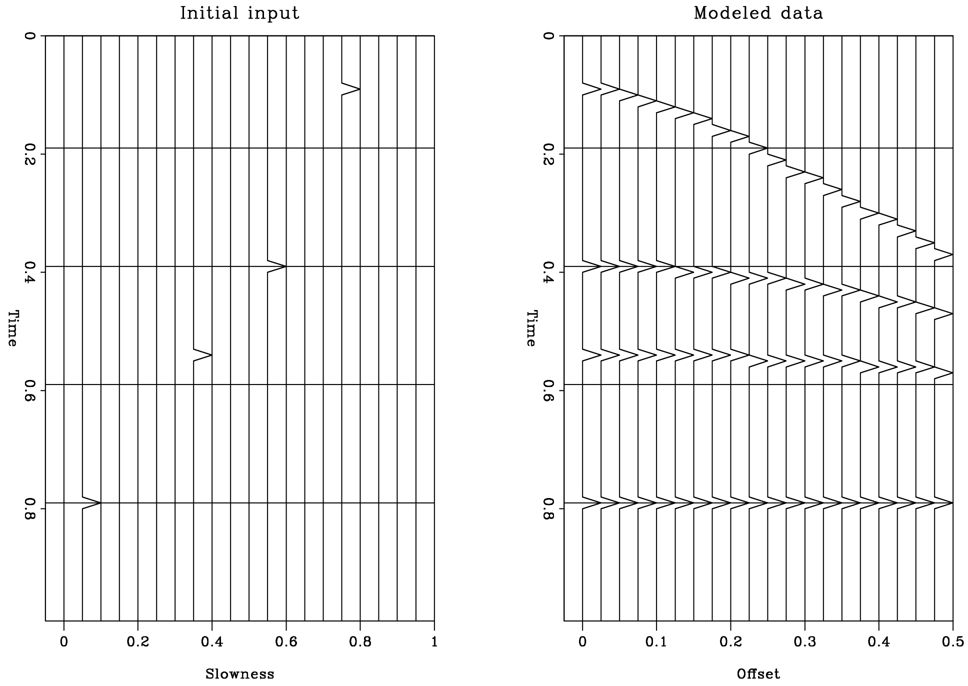


Figure 10: The left frame shows three events in a time-slowness space. The right frame shows the spreading of this events across the offset axis, made by the velocity analysis operator applied forwards. stacking-velanmod [NR]

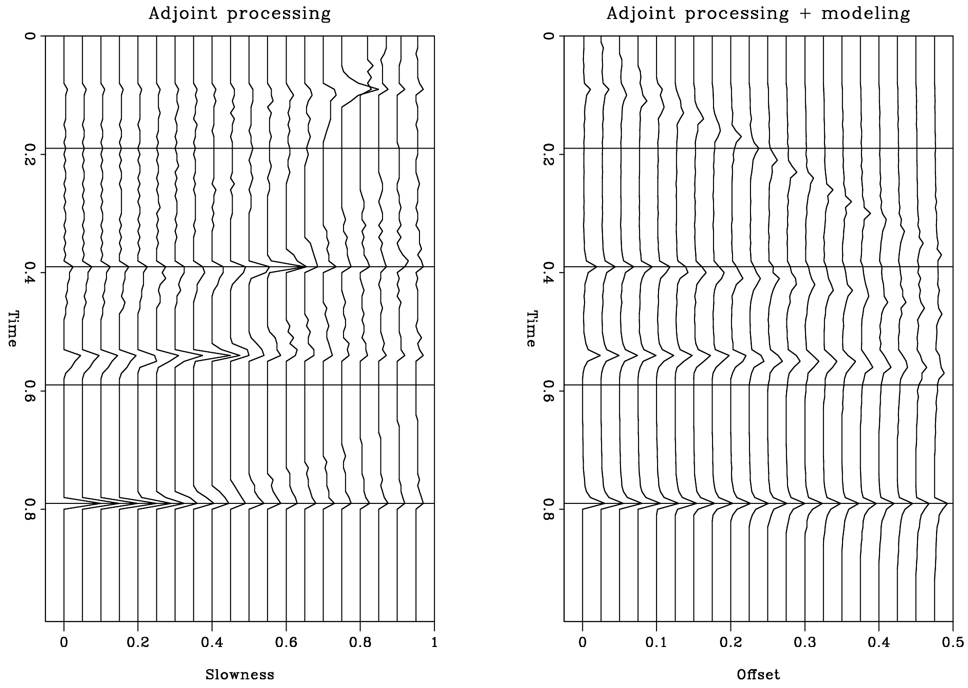


Figure 11: The left frame shows the result for the adjoint operator applied to the modeled data. The right frame shows the modeling operation applied to the adjoint processing data. The adjoint operation does not recover the modeled data. stacking-velanadj [NR]

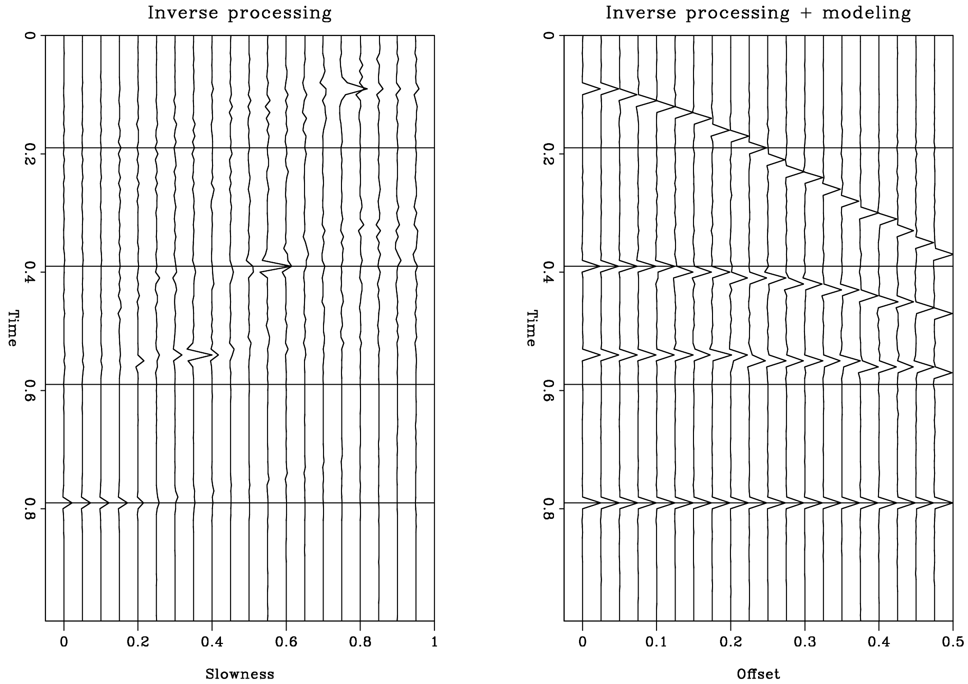


Figure 12: The left frame shows the inverse operator applied to the modeled data. The right frame shows modeling applied to the inverse processing, where even though the input does not reproduce Figure 10, the modeled data is recovered. stacking-velaninv [NR]

[5] The slant stack process, LMO and stack for a range of slowesses, is implemented in much the same way as the velocity analysis operator. It is a composition of an array of LMO-stack operators (class `fopLMOstk`) and a merge operator (class `fopMerge`).

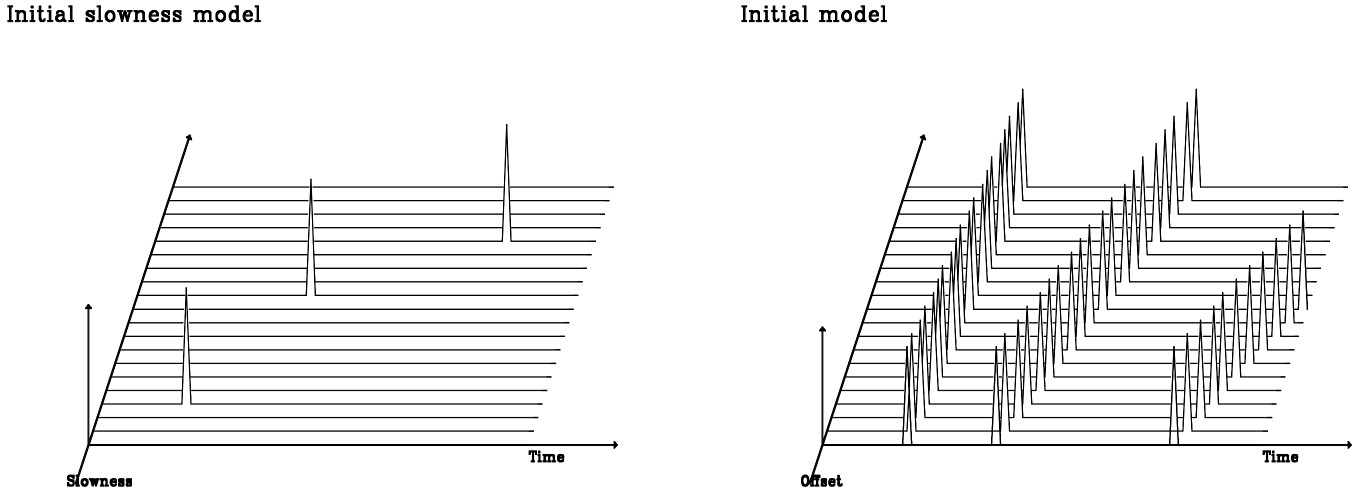
```
class fopSStack : public fopChain {
  public:
    //       time       offset    slowness
    fopSStack(Axis & t,  Axis & x,  Axis& s  );
};
```

All that is required to implement this new class is to write a constructor that builds the two operators and chains them together. I will not show this code here because it is identical to the code for constructing the `fopVelan` operator, except that it uses the `fopLMOstk` instead of the `fopNMOstk` operator internally (see `fopvelan` 9).

The forward operation of the slant stack operator models data from slowness-time space to offset-time space. In all the following figures the right hand plot is the result of applying the modeling operator to the left hand plot. Figure 13 shows the input, slowness-time, model on the left, it has three spikes in the slowness domain. The forward operator maps each spike to a linear event in offset-time space.

---

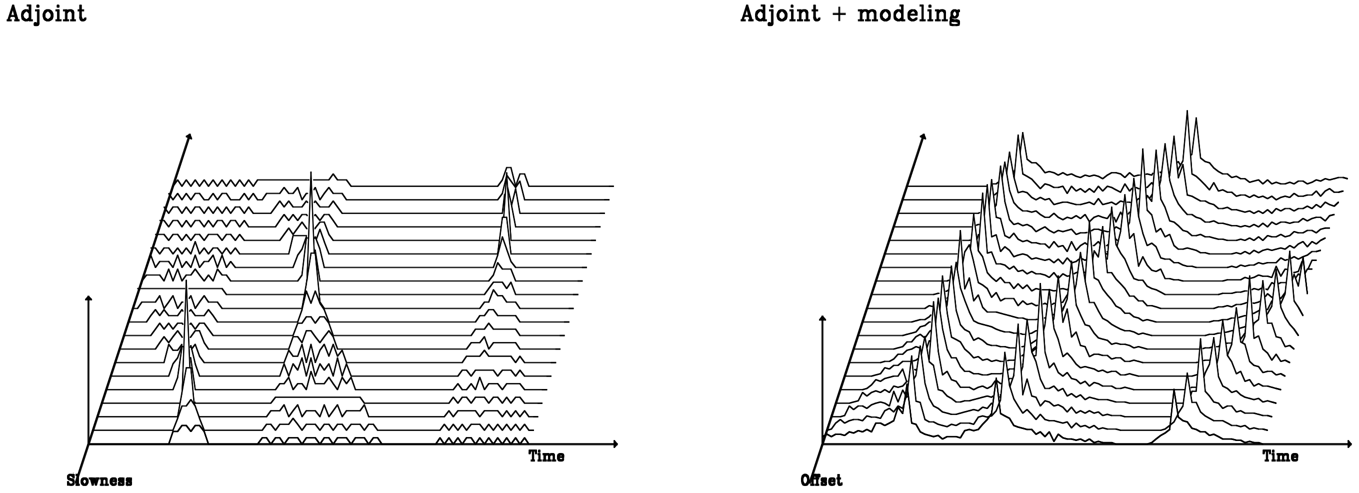
[5] Dave Nichols


Initial slowness model

Initial model

Figure 13: Left frame = input, right frame = modeling. stacking-out.1 [NR]

Adjoint processing of the modeled data is shown in figure 14, the three spikes have not been recovered, instead the familiar "cross" pattern is seen. The incomplete reconstruction is caused by two effects the absence of the "rho" filter and the finite aperture of the modeled data.


Adjoint

Adjoint + modeling

Figure 14: Left frame = adjoint processing, right frame = adjoint processing + modeling. stacking-adj.1 [NR]

Theory tells us that if the slant stack is performed by integration in an infinite medium (rather than summation in a finite medium) then the inverse of the forward operator is the adjoint operator plus the "rho" filter, the rho filter scales each frequency component by its magnitude. The lack of this frequency scaling is clearly seen when the adjoint processed data is modeled back to the offset-time space. Instead of the sharp linear event that was input a much lower frequency event is seen.

The "cross" in the result of adjoint processing is caused by the finite aperture of the input data. The slope of the two arms of the cross correspond to the minimum and maximum offset in the input. If the input data had infinite extent we would expect the cross to be absent.

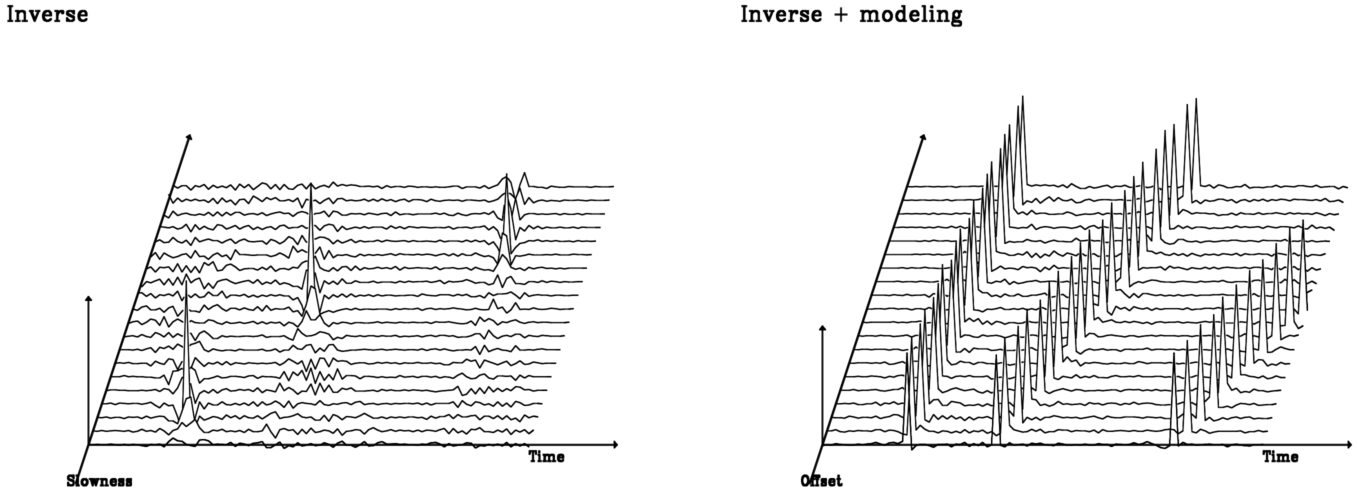


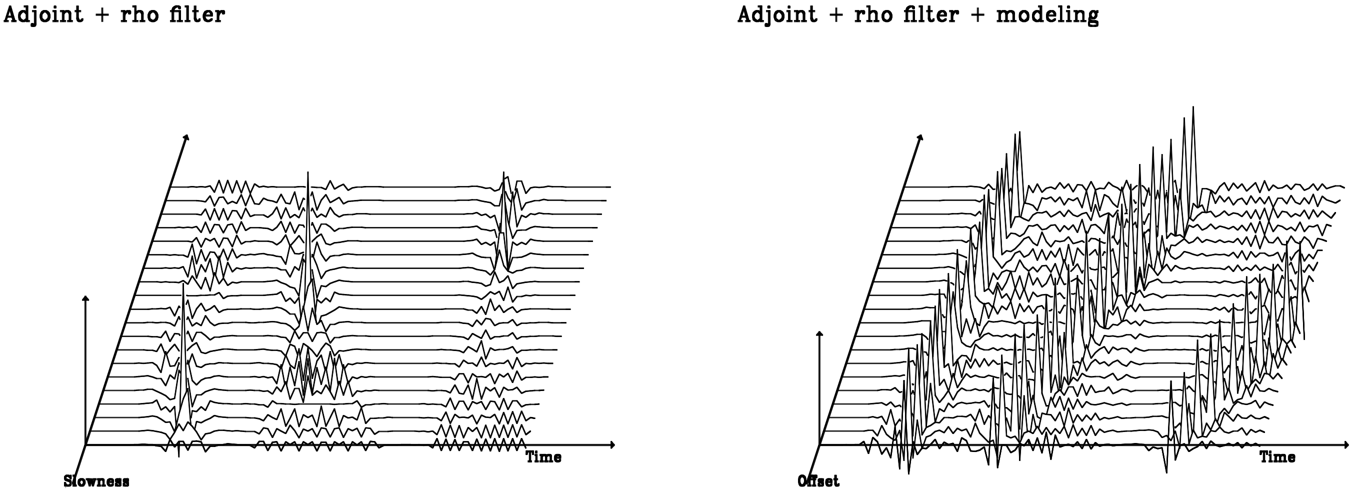
Figure 15: Left frame = inverse processing, right frame = inverse processing + modeling. stacking-inv.1 [NR]

Using a Hestenes solver to perform inverse processing removes almost all of these artifacts. Figure 15 shows the result of inverse processing after ten iterations. The result is much closer to the original model. When this data is modeled back to the offset-time domain it is a closer match to the input data. The iterative solver reported the following statistics from the inversion steps.

```
iteration: 0   residual norm: 5.40984   norm ratio: 0.757529
iteration: 1   residual norm: 3.60264   norm ratio: 0.504471
iteration: 2   residual norm: 2.54561   norm ratio: 0.356457
iteration: 4   residual norm: 1.5167    norm ratio: 0.212381
iteration: 6   residual norm: 1.06988   norm ratio: 0.149813
iteration: 8   residual norm: 0.858584  norm ratio: 0.120226
iteration: 10  residual norm: 0.700457  norm ratio: 0.0980836
```

The first step (which is essentially a scaled adjoint) produces a model that is a poor match to the input data. After 10 iterations there is still a residual with a norm that is 10% of the original but the fit to the data is much improved. Note that a 10% ratio in the norm means that all but 1% of the energy in the input has been modeled.

In an attempt to improve this result I decided to use the rho filter as a preconditioner for the solver. Figure **??** shows the result of using the adjoint operator with a rho filter. This still does not reconstruct the original data because of the finite aperture and finite sampling of the input. The slowness-time data still has the cross artifact in it because the adjoint has not corrected fot the finite aperture. The modeled output on the right now has a higher frequency content than the input rather than a lower one. This is because the rho filter has boosted the high frequencies too much, for finite aperture data the filter should be less powerful.


Adjoint + rho filter

Adjoint + rho filter + modeling

Figure 16: Left frame = adjoint processing + rho filter, right frame = adjoint processing + rho filter + modeling. stacking-adjrho.1 [NR]

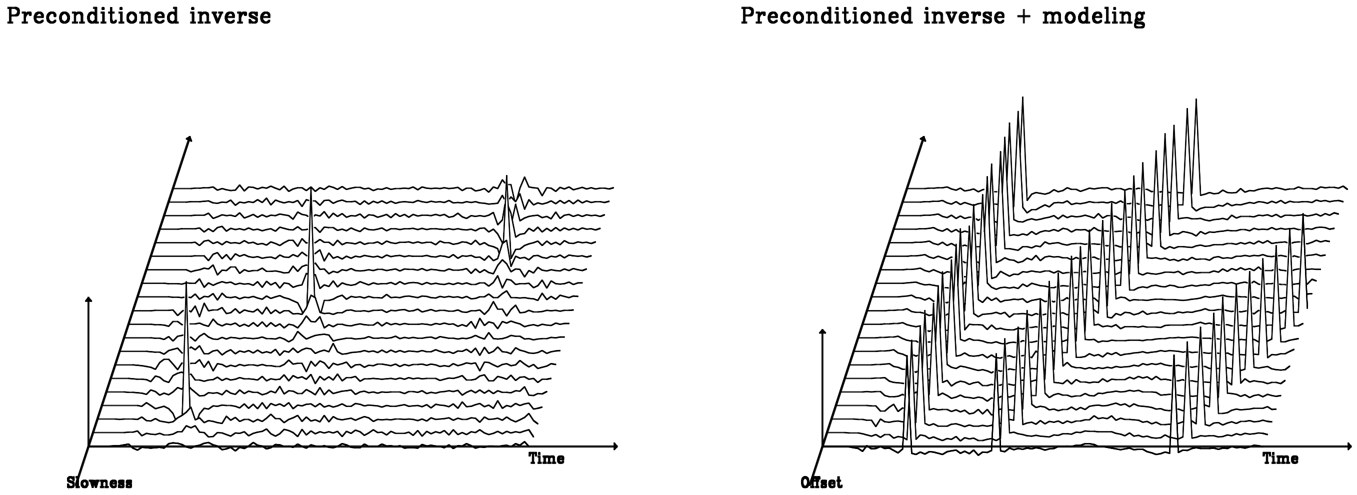
The rho filter can be regared as an estimate of $(A'A)^{-1}$ for the slant stack operator. As discussed in the chapter on solvers (chapter ??). We can use the square root of this operator as a preconditioner. The square root of the rho filter is a operator that scales each frequency by the square root of its magnitude. I use an operator of the class `fopContrunc` to implement this operator. The filter is an approximate time domain representation of the half-rho filter.

Figure ?? shows the result of solving the preconditioned inverse operator, it is a little worse than the non preconditioned result! The iterative solver reported the following statistics.

```
iteration: 0   residual norm: 3.70267   norm ratio: 0.518477
iteration: 1   residual norm: 2.78889   norm ratio: 0.390522
iteration: 2   residual norm: 2.44061   norm ratio: 0.341753
iteration: 4   residual norm: 2.04424   norm ratio: 0.286251
iteration: 6   residual norm: 1.79228   norm ratio: 0.25097
iteration: 8   residual norm: 1.6462    norm ratio: 0.230514
iteration: 10  residual norm: 1.53393   norm ratio: 0.214793
```

We can see that the use of the preconditioner did indeed make the operator more unitary. The first iteration has a residual norm that is less than that for the original operator. However, the preconditioner has not accelerated convergence of the algorithm, instead it has *slowed* it. The reason for this appears to be that most of the artifacts from the simple adjoint are due to the truncation and not the filtering effect. The rho filter has impeded the correction of these effects. If we looked at the eigenvalues of the transformation we would probably find that we had moved the largest eigenvalue closer to unity but we had not improved the clustering of the eigenvalues.

What can we conclude from this? It tells us that a preconditioner designed from the continuous problem is not necessarily a good precoditioner for the discrete problem. We need to look at the discrete problem more thoroughly to design a good preconditioner.



Preconditioned inverse

Preconditioned inverse + modeling

Figure 17: Left frame = inverse processing with half-rho preconditioner, right frame = inverse processing with half-rho preconditioner + modeling. stacking-invrho.1 [NR]