Stanford Exploration Project, Report 93, November 11, 1997, pages 305–316

Stanford Exploration Project, Report 93, November 11, 1997, pages 305–316

Short Note

A generic NMO program

Sergey Fomel, Sean Crawley, and Robert Clapp[1]

keywords: *NMO, interpolation, velocity*

INTRODUCTION

Jon Claerbout's books *Processing versus Inversion* (1992) and *Three-dimensional Filtering* (1994) list normal moveout (NMO) among the basic linear operators. Indeed, the NMO transformation plays a kernel role in many applications of geophysical data processing, from simple CMP stacking to prestack migration and velocity analysis. The importance of this role increases with the development of 3-D processing tools, since the so-called *Kirchhoff* method, based essentially on the NMO transformation, proved to be successful for irregular 3-D acquisition geometries.

In the most general terms, one can describe the NMO transform as a mapping, controlled by the traveltime transformation function and the assigned amplitude function. In the most common application, the traveltime function is a hyperbola. In other application, it may take the form of a linear function (slant stack), ellipsoid (DMO), etc. It can even be the result of an expensive ray tracing computation, as in the case of depth migration.

Because NMO is an operator central to many widely-varied processing and inversion schemes, and has a large space of implementation details which may affect its performance in an application-dependent manner, we seek to design a moveout operator (and more generally, a family of operators) which is easily adaptable to a particular research application. We use Fortran 90's modularity to separate such implementation details as input-orientation vs. output orientation and nearest neighbor interpolation vs. linear interpolation, from the definition of the normal moveout mapping. Coding in object-oriented style brings us reusable code: many different applications can be generated from a single source. It means that improving one part of the implementation (traveltime computation, amplitude computation, etc.) doesn't require rewriting the major library routines, and that one can easily compare different implementations with each other. Object-orientation also facilitates a cooperative code sharing among researchers. In this paper, we explain and illustrate these

[1]**email:** sergey@sep.stanford.edu, sean@sep.stanford.edu, bob@sep.stanford.edu

ideas with Fortran 90 programs.

We start with defining the relationship between the NMO transformation and the interpolation problem. We show how this relationship is affected by the choice of the "pull" or "push" implementation, and carry out generic NMO as a trace-by-trace transformation. Finally, we assemble a program for operations on irregular geometries defined by the Seplib90 data structure. The program is illustrated with a simple example.

NMO DEFINED

In a continuous world, we could define NMO precisely as a mapping transformation. If $M(t)$ is a continuous zero-offset trace, and $D(t)$ is a trace at some non-zero offset x, then conventional NMO with a constant velocity v is defined by the relationship

$$M(t) = D(\sqrt{t^2 + \frac{x^2}{v^2}}) \; . \tag{1}$$

If the continuous dot products in the model and data space have the usual L_2 form

$$(M_1(t), M_2(t)) \;=\; \int M_1(t)\, M_2(t)\, dt \; , \tag{2}$$

$$(D_1(t), D_2(t)) \;=\; \int D_1(t)\, D_2(t)\, dt \; , \tag{3}$$

the adjoint of operator (1) is simply

$$D(t) = \frac{t}{\sqrt{t^2 - \frac{x^2}{v^2}}}\, M(\sqrt{t^2 - \frac{x^2}{v^2}}) \; , \tag{4}$$

where the scaling factor appears as the result of the time coordinate stretching (Crawley, 1995). A generic inverse NMO operator can be defined accordingly as

$$D(t) = W(t)\, M\left(\tau^{-1}(t)\right) \; , \tag{5}$$

where the function τ defines the time coordinate transformation, and W describes the amplitude scaling. The continuous adjoint of (5) has the form

$$M(t) = \left|\frac{d\tau(t)}{dt}\right| W(\tau(t))\, D\left(\tau(t)\right) \; . \tag{6}$$

The adjoint operators (5) and (6) have quite different appearance in the discrete word of digital signal processing. After the stretch $\tau(t)$ or $\tau^{-1}(t)$ the signal transforms from a regular grid to an irregular distribution on the time axis. After it is interpolated back to the regular grid, discretization errors enter the computation, and we discover that operators (5) and (6) are no longer adjoint to the machine precision. This fact splits the implementation of the NMO transform into two branches: stretching and squeezing, or, as Claerbout (1995) calls them, " push" and "pull". In some applications, only one of them is available. This happens, for example, when the transform function $\tau(t)$ doesn't have an easily computed inverse. In other applications, both approaches are possible, which opens interesting possibilities for their comparison.

PULL AND PUSH NMO AND INTERPOLATION

NMO stretches a (data) trace at some offset to estimate a (model) trace at zero offset. This can be written as the matrix multiplication $m = N'd$. We have some choice in writing the matrix N'. For instance, we may choose N' to have a one in each column or a one in each row. In our subroutine, this translates to either choosing to loop over the values in the input d and push each to a sample in the output, variously termed an 'input-oriented' or 'push' operator; or choosing to loop over the values in the output m and for each pull in the appropriate input sample, an 'output-oriented' or 'pull' operator (Levin, 1994; Claerbout, 1992b). The matrixes represented by N' might look like Equation 1 for a push operator, or Equation 2 for a pull (after Levin (1994)).

$$m = N'd = \begin{bmatrix} . & . & 1 & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & 1 & . & . \\ . & . & . & . & . & . \\ . & . & . & . & 1 & . \\ . & . & . & . & . & 1 \end{bmatrix} \begin{bmatrix} d1 \\ d2 \\ d3 \\ d4 \\ d5 \\ d6 \end{bmatrix} \tag{7}$$

$$m = N'd = \begin{bmatrix} . & . & . & . & . & . \\ . & . & . & . & . & . \\ 1 & 1 & . & . & . & . \\ . & . & 1 & . & . & . \\ . & . & . & . & 1 & . \\ . & . & . & . & . & 1 \end{bmatrix} \begin{bmatrix} d1 \\ d2 \\ d3 \\ d4 \\ d5 \\ d6 \end{bmatrix} \tag{8}$$

Neither of the above is perfect, in that each is a hyperbola and not a waveform. However, the lower might be called better because it does not leave holes in the output (Claerbout, 1995a).

To do inversion, we need operator pairs. Depending on our inversion method, we may or may not need an operator pair which is exactly adjoint. The adjoint of a push operator will be a pull operator, and vice versa, though push push or pull pull pairs can be approximately adjoint. Also, though we can choose a solver methodology which allows us to use an approximate adjoint (Fomel, 1996), pull operators may still be impractical. Coding NMO forward modeling as a pull operator is difficult because where velocity is not constant, traveltime may be multivalued (Claerbout, 1995a), or at least nontrivial to compute. This is a problem if we wish to upgrade stacking by inversion, but not for inverse velocity transform. Thus, as is the case for many geophysical operators, optimal implementation really is a function of application. The next section shows how different implementations may be easily swapped.

IMPLEMENTATION

The following Fortran 90 module implements an object-oriented design of the generic NMO program.

```
module nnnmo
  use interp

  logical,                          private :: pul
  integer,                          private :: nt
  real,                             private :: ot, dt
  real, dimension (:), allocatable, private :: t, z, w

contains

  subroutine nnnmo_init (pull, t0, z0, dz, n)
    logical, intent (in) :: pull
    real,    intent (in) :: t0, z0, dz
    integer, intent (in) :: n

    integer              :: i

    pul = pull ; ot = t0 ; dt = dz; nt = n
    if (.not. allocated (z)) allocate (z (nt), t (nt), w (nt))
    if (pul) then                                   ! create a regular trace
       z = (/ (z0 + dt * (i - 1), i = 1, nt) /) ; ot = t0
    else
       t = (/ (t0 + dt * (i - 1), i = 1, nt) /) ; ot = z0
    end if
  end subroutine nnnmo_init

  subroutine nnnmo_step (type, time, ampl, x)
    integer,             intent (in) :: type
    real, dimension (:), intent (in) :: x
    interface
       function time (pull, x, z, t) result (stat)
         integer             :: stat
         logical             :: pull
         real, dimension (:) :: x, z, t
       end function time
       function ampl (pull, x, z, t, weight) result (stat)
         integer             :: stat
         logical             :: pull
         real, dimension (:) :: x, z, t, weight
       end function ampl
    end interface

    integer                         :: stat1,stat2

    stat1 = time (pul, x, z, t)                    ! time transform
    stat2 = ampl (pul, x, z, t, w)                 ! amplitude

    if (pul) then                                  ! interpolation setup
```

```
      call interp_init (type, t, ot, dt, nt, w)
   else
      call interp_init (type, z, ot, dt, nt, w)
   end if
 end subroutine nnnmo_step

  function nnnmo_op (adj, add, mod, dat) result (stat)
    integer             :: stat
    logical, intent (in) :: adj, add
    real, dimension (:)  :: mod, dat

    integer             :: stat0

    if (pul) then                                  ! interpolate to a trace
       stat0 = interp_op (.not. adj, add, dat, mod)
    else
       stat0 = interp_op (      adj, add, mod, dat)
    end if

    stat = 0
  end function nnnmo_op

  subroutine nnnmo_close ()
    call interp_close ()
    deallocate (z, t, w)
  end subroutine nnnmo_close

end module nnnmo
```

The module contains the initialization routine `nnnmo_init`, which receives the time grid information and saves it to module variables. It also translates the regular grid into a one-dimensional array. The next subroutine, `nnnmo_step`, computes the NMO time stretch and amplitude scaling. Its arguments are a logical variable, which defines pull or push operation, a real array `x`, which may contain the trace header information, and two functions to compute the time and amplitude transformation. The user of the `nnnmo` module is free to specify these functions according to any particular application provided that they comply to the generic interface. Typical examples of commonly used functions are collected in module `nmo`.

```
module nmo_mod
contains

  function linear_time  (pull, x, z, t) result (stat)
    integer            :: stat
    logical            :: pull
    real, dimension (:) :: x, z, t
```

```
  if (pull) then
     t = z + sum (x)
  else
     z = t - sum (x)
  end if

  stat = 0
end function linear_time

function hyper_time (pull, x, z, t) result (stat)
  integer             :: stat
  logical             :: pull
  real, dimension (:) :: x, z, t

  if (pull) then
     t = sqrt (z*z + dot_product (x,x))
  else
     z = t*t - dot_product (x,x)
     where (z > 0.) z = sqrt (z)
  end if

  stat = 0
end function hyper_time

function vofz_hyper_time (pull, x, z, t) result (stat)
  integer             :: stat
  logical             :: pull
  real, dimension (:) :: x, z, t

  if (pull) then
     t = sqrt (z*z + x*x)
  else
     z = t*t - x*x
     where (z > 0.) z = sqrt (z)
  end if

  stat = 0
end function vofz_hyper_time

function unit_ampl (pull, x, z, t, w) result (stat)
  integer             :: stat
  logical             :: pull
  real, dimension (:) :: x, z, t, w

  w = 1. ; stat = 0
end function unit_ampl

function cos2D_ampl (pull, x, z, t, w) result (stat)
  integer             :: stat
  logical             :: pull
  real, dimension (:) :: x, z, t, w

  w = 0.
```

```
    if (pull) then
       where (t > 0.) w = (z / t) / sqrt (t)
    else
       where (t > 0.) w =     1.  / sqrt (t)
     end if

    stat = 0
  end function cos2D_ampl

end module nmo_mod
```

After initialization calls to `nnnmo_init` and `nnnmo_step`, the actual operator `nnnmo_op` simply calls an interpolation operator to complete the job. When the `pull` parameter is set to `.true.`, the forward operator is "push", and the adjoint is "pull". In this case, the time interpolation is carried out in the data space. When `pull` is `.false.`, the time interpolation is done on the model trace. The interpolation is performed by module `interp`, which calls either `bin1` or `lint1` (Fomel and Claerbout, 1996) depending on the user-specified parameter.

```
module interp

  use bin1
  use lint1

  implicit none

  integer, private :: tp

contains

  subroutine interp_init (type, coord, o1, d1, n1, weight)
    real, dimension (:), intent (in) :: coord, weight
    real,                intent (in) :: o1, d1
    integer,             intent (in) :: type, n1
    optional                         :: weight

    tp = type
    select case (tp)
    case (1)
       call  bin1_init  (coord, o1, d1, n1, weight)
    case default
       call  lint1_init (coord, o1, d1, n1, weight)
    end select
  end subroutine interp_init

  function interp_op (adj, add, mod, ord) result (stat)
    integer              :: stat
    logical, intent (in) :: adj, add
```

```
    real, dimension (:)  :: mod, ord

    integer              :: stat0

    select case (tp)
    case (1)
       stat0 = bin1_op  (adj, add, mod, ord)
    case default
       stat0 = lint1_op (adj, add, mod, ord)
    end select

    stat=0
  end function interp_op

  subroutine interp_close ()
    select case (tp)
    case (1)
       call bin1_close  ()
    case default
       call lint1_close ()
    end select
  end subroutine interp_close

end module interp
```

Antialiasing

If an NMO-type operator is applied on regularly sampled but spatially aliased data, we can enhance it with an efficient antialiasing technique, based on low-pass time filtering (Claerbout, 1992a; Bevc and Claerbout, 1992; Lumley et al., 1994).

Without going into implementation details, we can mention that the interface for initializing anti-aliased NMO is slightly different from that of `nnnmo`, but the operator interface is very similar. At the initialization step, the anti-aliased NMO needs a spatial sampling interval and a function to compute the slope of the time transform function. The operator function for antialiased NMO calls a "triangular" interpolation routine, which invokes multiple chaining of interpolation operators.

NMO IN SEPLIB 90

SEPF90(Clapp and Crawley, 1996a,b) provides an object-oriented Fortran90 interface to SEplib90(Biondi et al., 1996), SEP's irregular data extension to SEPlib. With SEPF90, irregular seismic data can be read, processed, and written with a minimum of cluttering overhead, and with an increased proportion of reusable code.

The power of an object-oriented definition of both NMO and data is best illustrated through example. The program `simple.f90` creates a synthetic CMP gather from irregular trace geometries, a $v(z)$ velocity model, and a zero offset trace.

simple.f90

```
program test
  use sep_f90_lib_mod     ! sepf90 library (for simple I/O)
  use nnnmo               ! generic shift and scale programs
  use nmo_mod             ! module containing shift and scale definitions

  type(sep_90),pointer          :: sepdata,velocity  ! sep90 derived types
  real,pointer,dimension(:)     :: offset            ! trace offsets
  real,pointer,dimension(:,:)   :: in_data,out_data  ! cmp values
  real,pointer,dimension(:)     :: vel_vals          ! velocity values
  real,allocatable,dimension(:) :: t                 ! x/vel
  integer                       :: i2,stat,n1,interp
  real                          :: o1,d1             ! time axis parameters

  call initpar()                                     !
  call getch("interp","d","inter")                   ! get interpolation type
  nullify(in_data,out_data,offset,vel_vals)          ! nullify pointer arrays
  call init_vars(sepdata,velocity)                   ! set up SEP data
  call sep_reed("velocity",velocity)                 ! read in the velocity
  call get_trace_vector(velocity,vel_vals)           ! put velocity in an array
  call sep_reed("in",sepdata)                        ! read in cmp gather
  call get_trace_slice(sepdata,in_data)              ! copy values to an array
  o1=sepdata%o(1);d1=sepdata%d(1);n1=sepdata%n(1)    !
  allocate(out_data(size(in_data,1),size(in_data,2)))! create output space
  call nnnmo_init (.true., o1, o1, d1, n1)           ! initiate nmo operator
  allocate(t(n1))
  call get_key_values(sepdata,"aoffset",offset)      ! get offset values
  do i2=1,size(in_data,dim=2)                        ! loop over traces
     t=offset(i2)/vel_vals                           ! init  t

     call nnnmo_step(interp,vofz_hyper_time,cos2D_ampl,t)
     stat=nnnmo_op(.false.,.false.,in_data(:,i2),out_data (:,i2))

  end do
  call nnnmo_close ()                                ! close nmo operator
  call put_trace_slice(sepdata,out_data)             ! trace into septype
  call sep_rite("out",sepdata)                       ! write out dataset
end program test
```

By performing this ‘spray’ operation (L) followed by its adjoint, NMO (L'), we should recover something close to the input. Figure 1 shows the input trace, the modeled CMP gather, and the result of applying LL'.

Their are two primary advantages to the object-oriented style. First, it is virtually data independent, no portion of the code makes assumptions on data geometry. Second, because the implementation is highly modular, experimenting with the effect of constant vs. variable velocity, differing interpolation and anti-aliasing schemes, or push vs. pull operator implementation is simple. For example, suppose we wish to

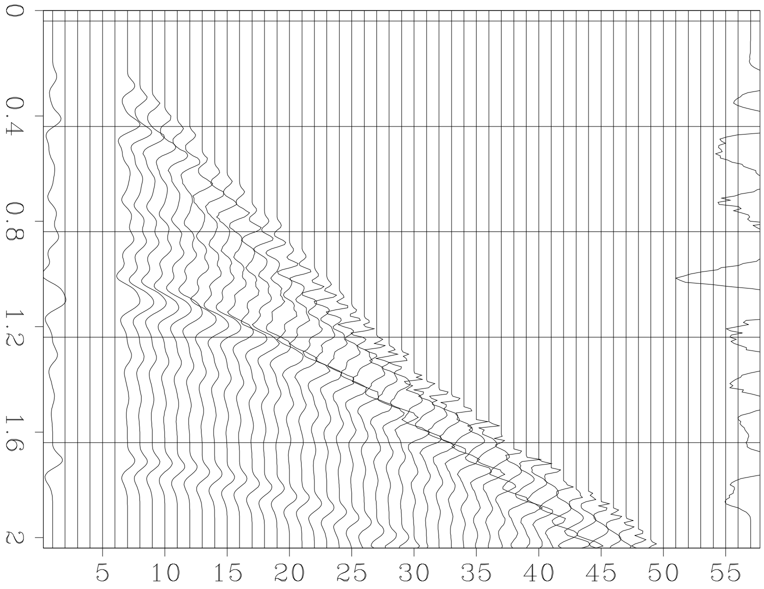

Figure 1: Trace on the left represents the input t(0) model; middle portion is the sprayed, irregularly spaced, CMP gather; right represents the result of applying L followed by L'. sergey1-nmo.simple [CR]

see the effect on recovering the original $t(0)$ trace due to variable vs. constant velocity for both linear and nearest-neighbor interpolation. Only the input velocity trace and the interpolation parameter (command-line arguments in the case of `simple.f90`) need to be altered. Figure 2 shows the resulting CMP gathers while Figure 3 shows the resulting $t(0)$ trace using the different velocity characterizations and interpolation schemes. As expected linear interpolation outperforms nearest-neighbor interpolation in recovering the amplitudes without generating high-frequency noise; also the adjoint is a slightly better approximation when we have constant velocity compared to a $v(z)$ medium.

Rewriting the NMO "spray and stack" program for different kinds of transformation (DMO, post-stack and prestack migration, datuming, etc.) becomes a fairly simple exercise.

CONCLUSIONS

Normal moveout mapping is an example of a geophysical operator which is central to many and diverse research projects, and which may have enhanced or diminished suitability for a given task depending upon various implementation details. A convenient code for such an operator should be easily reusable, in easily configurable form; antialiasing, push vs. pull, constant vs. stacking vs. interval velocity, etc. should be selectable by the researcher with trivial code modification. We implement such a moveout operator in this paper.

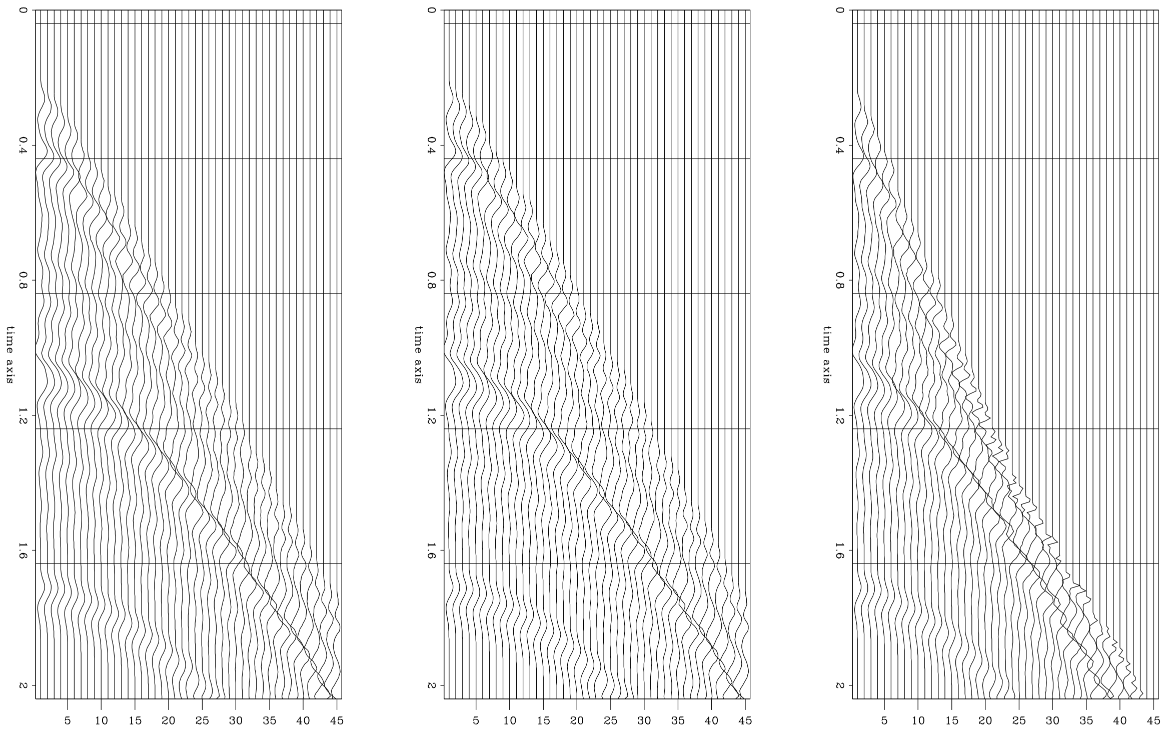

Figure 2: Left panel is modeling using constant velocity and linear interpolation, center panel nearest-neighbor and v(z), right panel linear interpolation and v(z). sergey1-nmo-mix [CR]

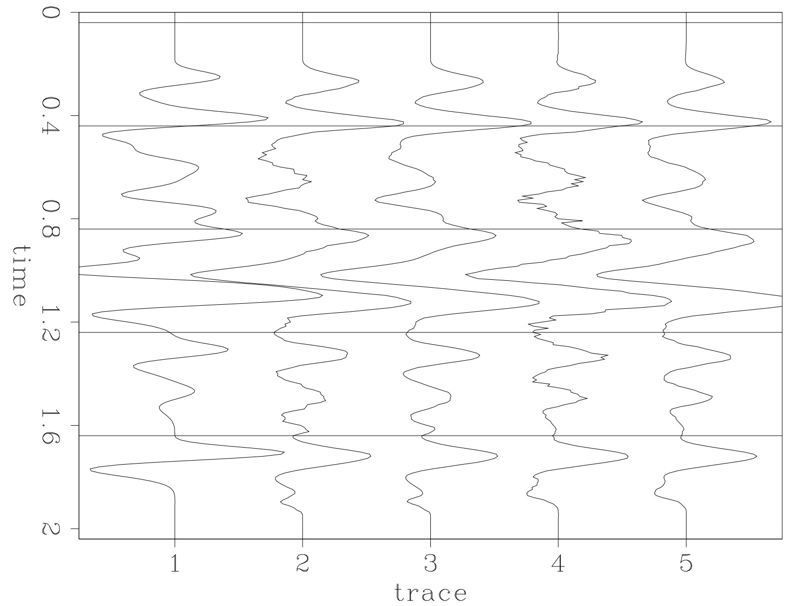

Figure 3: Traces from left to right, scaled t(0) input; result of constant velocity (CV), nearest-neighbor interpolation (NN); (CV) and linear interpolation (L); v(z) and NN; and v(z) and linear. sergey1-ampl [CR]

Over half of the reproducible figures in *Basic Earth Imaging* (Claerbout, 1995b) belong to the extended definition of an NMO-type operator. We were able to reproduce almost all of them with a single NMO subroutine. Introducing the SEPF90 data structure brings this subroutine to the world of irregular 3-D geometries. We hope to see useful applications of it in 3-D processing (and inversion) projects.

ACKNOWLEDGMENTS

We are thankful to Jon Claerbout for his interest in this project and severe critic of the earlier versions of this paper. Of course, we are solely responsible for the remaining imperfections.

REFERENCES

Bevc, D., and Claerbout, J., 1992, Fast anti-aliased Kirchhoff migration and modeling: SEP–**75**, 91–96.

Biondi, B., Clapp, R., and Crawley, S., 1996, Seplib90: Seplib for 3-D prestack data: SEP–**92**, 343–364.

Claerbout, J. F., 1992a, Anti aliasing: SEP–**73**, 371–390.

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

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

Claerbout, J., 1995a, Ellipsoids versus hyperboloids: SEP–**89**, 201–206.

Claerbout, J. F., 1995b, Basic Earth Imaging: Stanford Exploration Project.

Clapp, R. G., and Crawley, S., 1996a, Fortran90: Introduction and use in 3-D geophysical problems: SEP–**92**, 365–384.

Clapp, R. G., and Crawley, S., 1996b, SEPF90: SEP–**93**, 293–304.

Crawley, S., 1995, Approximate vs. exact adjoints in inversion: SEP–**89**, 207–216.

Fomel, S., and Claerbout, J., 1996, Simple linear operators in Fortran 90: SEP–**93**, 317–328.

Fomel, S., 1996, Least-square inversion with inexact adjoints. Method of conjugate directions: A tutorial: SEP–**92**, 253–265.

Levin, S. A., 1994, Stolt without artifacts? — dropping the Jacobian: SEP–**80**, 513–532.

Lumley, D. E., Claerbout, J. F., and Bevc, D., 1994, Anti-aliased Kirchhoff 3-D migration: SEP–**80**, 447–490.