Stanford Exploration Project, Report 75, November 17, 1997, pages 293–309

"Plug and Play" wave equation modules

Martin Karrenbach[1]

keywords: *modeling, algorithm, acoustic, elastic*

ABSTRACT

Prestack modeling and migration programs can share a common structure. I outline this structure in the common shot domain for scalar or vector fields of any degree of anisotropy. The generality of the code is striking, and works for any combination of space and vector dimensionality. The actual pieces of FORTRAN 90 code are shown as listings in the paper. The advantage of this modeling strategy is that there exists only one piece of code that has to be maintained. The various media types are declared in the main program as single variables (objects), and they all use the same subroutines. The mathematical formulation is apparent in the notation and the modeling code can easily handle the usual finite differences or pseudo spectral cases or any combination. Reverse time migration is accomplished by using the same set of subroutines, conjugating of the modeling operation and additionally calculating the gradient ("imaging"). There is a trade-off between performance and common structure when, for example, the Laplacian $b\nabla^2$ could be used instead of $\nabla b \nabla$ in constant velocity models.

INTRODUCTION

For theoretical details I refer to some of the recent papers by Etgen (1989), Mikhailenko (1991); or Karrenbach (1991) for anisotropic imaging and material property calculation, and Cunha (1991) for reflectivity retrieval using reverse time migration.

As a programming language FORTRAN 90 (F90) is practical and uses a reasonably compact notation. Starting from the most general expression does not necessarily mean starting with the most difficult of all cases. As it turns out, the most general case, can be written very simply. Reducing the space dimensionality or reducing "component" dimensionality is merely reflected in different loop limits, while the code itself remains the same. In a recent report Muir (1991) implemented a similar idea in F90 with great success.

I was inspired to do this by two things. First Jon Claerbout's programming method for his recent book "Processing versus Inversion". It contains a collection of

[1]**email:** not available

2D processing subroutines that are used through out the book as building blocks for larger subroutines and programs. When using those I was amazed by the versatility and easy maintainability. Secondly I have written a couple of modeling and migration programs for various types of acoustic and elastic media. Those programs are very similar in in structure and methodology. The aim of this paper is to consolidate those programs into one or more, which all use the same building blocks. Those building blocks can serve as a basis for modeling and migration programs by a linking mechanism. Also, these blocks themselves may be added to or extended with other existing modeling tools. It is my hope that other interested people contribute to this effort. I would like to end up with a minimal set of most functional modeling and migration building blocks which can be arranged with ease to create various state of the art modeling programs, from plain finite differences with exchangeable operators, to spectral methods with Chebychev expansion or any combination of those. It might sound ambitious, but in F90 this is not an impossible task.

WAVE EQUATIONS

The general modeling implementation is based on a wave equation of the form

$$a(x)\nabla b(x)\nabla^T u(x,t) - \frac{\partial}{\partial t^2} u(x,t) = f(t) \tag{1}$$

u is an arbitrary wave field (scalar or vector); f is the force applied at a source location. ∇ is a gradient and ∇^T the divergence operator applied to the wavefield components. a and b are medium property fields (density, velocity or stiffnesses). Equation (1) can be rewritten as a sequence of first order partial differential equations. Numerical calculations typically use those on a staggered grid for improved accuracy.

Elastic medium

For the anisotropic elastic case the differential operator ∇ is a 6x3 matrix of partial spatial derivatives and when operating on the displacement field results in the symmetric Lagrange strain tensor.

$$\nabla^T = \frac{1}{2}\left(\frac{\partial}{\partial x_l} + \frac{\partial}{\partial x_k}\right) \qquad \text{with} \qquad k, l = 1, 2, 3 \tag{2}$$

Giving us the following set of first order (in space) equations:

$$\epsilon_{kl} = \frac{1}{2}\left(\frac{\partial u_k}{\partial x_l} + \frac{\partial u_l}{\partial x_k}\right) \tag{3}$$

$$\sigma_{ij} = b_{ijkl}\epsilon_{kl} \tag{4}$$

$$f_i = a\frac{1}{2}\left(\frac{\partial}{\partial x_i} + \frac{\partial}{\partial x_j}\right)\sigma_{ij} + \frac{\partial}{\partial t^2} u_j \tag{5}$$

where a is the density and b represents the stiffness matrix. In general the stiffness matrix is a sparse matrix, and can be simplified for different degrees of symmetry.

Acoustic Medium

In the special case of an acoustic medium the stiffness matrix and the stress tensor reduces to a scalar value (impedance and pressure). Identifying u in (5) with the pressure gradient, we can write again the familiar set of first order equations.

$$\rho \frac{\partial}{\partial t} u_i = -\frac{\partial}{\partial x_i} P \tag{6}$$

$$\frac{\partial}{\partial t} P = K \frac{\partial}{\partial x_k} u_k \tag{7}$$

This corresponds to the general form (1), where here K is the bulk modulus and ρ is the density.

It is obvious from the previous two paragraphs that the equation structure is identical for different type of media. One can either treat the acoustic case as a completely self contained problem or as an elastic problem where the material properties happen to be acoustic. Differences in spatial dimensionality or component dimensionality are reflected just in the bounds of the subscripts. From that we can conclude that if we have a given modeling structure we can use it for all our modeling purposes. It is just a matter of keeping track of the indices and to make sure those indices are efficiently mapped in each case. Thus all the building blocks can reuse the same pieces of code if they are written in a general form (i.e. using the full anisotropic formulation). Clearly for small spatial variations in density we could use a more compact Laplacian in case of the cascaded first order derivative operators.

OBJECT TYPES

Variables used in the entire program are located in an include file which acts as an object specification for different variables. I list here only some of the possible types I adopted. Since this information is in an include file, the port to different machines is easy and only a few machine dependent lines have to be changed.

- `MODEL(uuu)` specifies the variable uuu to contain model material parameters; it has typically 3 space dimensions
- `MODELLAY(uuu)` specifies a layout for the variable uuu. This is a machine dependent specification.
- `WFIELD(uu)` designates the variable uu as wave field, eg. displacement or pressure.
- `TENSOR(uu)` designates the variable uu as a tensor, eg. stress or strain.
- `STIFF(uu)` designates the variable uu as an elastic stiffness tensor.

- `BOUND(uu)` designates the variable uu as a boundary grid type.

Each if those types is actually a macro that is expanded during compilation. And if a change to the types is made, automatically all source will be getting the correct type. To see how to use these specifications, see the programs included in this paper.

The modeling program right now categorizes the medium in different types: "ACOUSTIC, ISOTROPIC, CUBIC, TRANSISO, ORTHO, MONO and TRICLINIC". The keywords are recognized as parameters and some simple parameter checking is performed. But more importantly, these keywords set an internal representation of the stiffness matrix. This representation is a mask function as can be seen in the code `getstress()` on page ??. This representation is a simple image of the symmetry properties of the medium in the notation most commonly used. As an example, here is the acoustic symmetry mask:

```
stiffmask(1,1)=1;
        stiffmask(2,2)=1;
                   stiffmask(3,3)=1;
```

and here the transverse isotropic symmetry mask:

```
stiffmask(1,1)=1; stiffmask(1,2)=1; stiffmask(1,3)=1;
stiffmask(2,1)=1; stiffmask(2,2)=1; stiffmask(2,3)=1;
stiffmask(3,1)=1; stiffmask(3,2)=1; stiffmask(3,3)=1;
                                              stiffmask(4,4)=1;
                                                         stiffmask(5,5)=1;
                                                             stiffmask(6,6)=1;
```

In general my coding scheme implements notions of C++ or look very similar to code written in C++. This suggests that C++ would be the preferred language. Nichols et al (1992) and Dunbar (1992) describe in detail the design and implementation of linear operators in C++.

MODELING STRUCTURE

Following is an outline of the modeling algorithm for a general wave equation (1). Compare this to subroutine `wave()` on page ??.

loop over shots {

calculate accelerations {

calculate strain

stress = b(x) × strain

apply stress boundary conditions
(add in stress sources)
constrain stress (eg. free surface)
calculate divergence of stress
accel = a(x) $\times$ div of stress
}
add in acceleration sources
time update the wave field
record seismograms
}

Strain calculation

The calculation of the strain tensor (pressure gradient) is concisely written in the subroutine `del()` on page ??. The notation follows the mathematical formula in equation (1). The only aberration from the mathematical notation becomes obvious where I use abbreviated indices for practical reasons. It is transparent which of the calculated components are evaluated in a staggered fashion. If `conj==0` then subroutine `del()` on page ?? will evaluate ∇^T in (1), meaning we perform an outer product. Displacement uu and strain ss are conjugates of each other. The actual derivative calculation is carried out in subroutine `conv1dc()` on page ??, which is a finite difference approximation to a partial derivative. Instead of calling a convolution we could have called a Fourier derivative, then we would perform pseudo-spectral modeling. Or we could alternate between them depending on which space axis is being calculated.

```
# apply symmetric Lagrangian derivative ( with 8 point convolution)
#
subroutine del (conj,uu,t,ss,bnd)
implicit none
#include "commons"
integer         conj,   t
WFIELD(uu)
WFIELDLAY(uu)
TENSOR(ss)
TENSORLAY(ss)
BOUND(bnd)
BOUNDLAY(bnd)
integer ivec,iten,direction
real scale

if(conj==0) {ss = 0.;        }
else        {uu(t,:,:,:,:)=0.}

do ivec=1,ncomp {
```

```
do iten=1,ntcomp{    direction=trudim(deriv(tcomp(iten),icomp(ivec)))
 if (direction>0){   scale = scaling(tcomp(iten),icomp(ivec))
  if (conj==0)  { stagger = stagc0(tcomp(iten),icomp(ivec)) }
  else          { stagger = stagc1(tcomp(iten),icomp(ivec)) }
    call conv1dc(conj,uu,t,ivec,ss,iten,direction,ddx(direction)*scale,bnd)
  }
}}

return
end
```

Stress calculation

The stress calculation is an uneventful subroutine `getstress()` on page ??. It performs the double inner product of the stiffness matrix with the strain tensor and yields the stress tensor. Note again the nearly mathematical formulation as in (5). Since the stiffness matrix is generally sparse we only use nonzero elements in the calculation, the scale applied to certain components is due to the ∇ calculation according to formula (2).

```
# getstress:    Hooke's Law
# get stresses from strain
#%
subroutine getstress(strain,stiff,stress,bnd)
implicit none
#include "commons"
TENSOR(strain)
TENSORLAY(strain)
TENSOR(stress)
TENSORLAY(stress)
STIFF(stiff)
STIFFLAY(stiff)
#TENSOR(stiff)
#TENSORLAY(stiff)
BOUND(bnd)
BOUNDLAY(bnd)
integer i,j,k,l

stress=0.

do i=1,ntcomp {
do j=1,ntcomp {               k = ccin(tcomp(i),tcomp(j))
   if (k!=0)  {
      stress(i,:,:,:) = stiff(k,:,:,:) * stiffmask(tcomp(i),tcomp(j))*
                        strain(j,:,:,:) + stress(i,:,:,:)
      }
}}

return
end
```

Acceleration calculation

Calculating the stress divergence yields the acceleration (second time derivative of u). The divergence operation is just the conjugate of ∇^T. For that I am using again subroutine `del()` on page ?? with `conj=1`. This calculates the acceleration components from the stress tensor.

Convolution operator

```
# conv1dc
# convolution 1 dimensional circular boundary condition
# u and w are conjugate to each other
# axis: axis over which to convolve
# stagger:  0 normal conv
#           1 shift by one grid point
#%
subroutine conv1dc(conj,u,t,k,w,sidx,  axis, scale,bnd)
implicit none
#include "commons"
integer             conj,  t,k,  sidx
integer                                  axis
real                                           scale
WFIELD(u)
WFIELDLAY(u)
TENSOR(w)
TENSORLAY(w)
BOUND(bnd)
BOUNDLAY(bnd)
integer ishift

if (conj==0) {                           #  the gradient
do ishift = nlo-stagger, nhi-stagger {
   w(sidx,:,:,:) = w(sidx,:,:,:) +
        coef(ishift+stagger)*cshift(u(t,k,:,:,:),dim=axis,shift=ishift) * scale
   }
}
else{
do ishift = nlo-stagger, nhi-stagger {  #  the divergence
   u(t,k,:,:,:) = u(t,k,:,:,:) +
       coef(ishift+stagger)*cshift(w(sidx,:,:,:),dim=axis,shift=ishift) * scale
   }
}
return
end
```

The finite difference operator is implemented as a 8 point convolution. The boundary conditions are circular, but a two line change will allow absorbing one-way boundary conditions. Even a spatially changing convolution operator (variable grid spacing) is mere one line change.

The wave operator

```
# apply the simple wave equation operator
# repeatedly  useable for more complicated time updates
#
subroutine wave(u,strain,stiff,stress,idim,ndim,icomp,ncomp,coef,nlo,nhi_
                                                        ,abbin,ccin,truin,bnd)
implicit none
integer                                                    nlo,nhi, bnd(:,:,:)
real             u(:,:,:,:),   strain(:,:,:,:), stress(:,:,,:)
real                                                   coef(nlo:nhi),ddx(3)
#mf$layout       u(:serial,:news,:news,:news)
#mf$layout       stress(:serial,:news,:news,:news)
#mf$layout       strain(:serial,:news,:news,:news)
#mf$layout       bnd(:news,:news,:news)
integer           idim(3),ndim,icomp(3),ncomp,  abbin(3,3)

call del(0,u(2,:,:,:,:),idim,ndim,icomp,ncomp,strain,abbin,coef,nlo,nhi,ddx,bnd)

call getstress(strain,stiff,stress,idim,ndim,icomp,ncomp,abbin,ccin,truin,bnd)

where(free)     #free surface constraint
  stress(abbin(1,3)) = 0.
end where

call del(1,u(2,:,:,:,:),idim,ndim,icomp,ncomp,strain,abbin,coef,nlo,nhi,ddx,bnd)

#add in sources

return
end
```

Combining the previous 3 steps we can compose the wave operator $\Box$ which is the left most term in (1) and is given in subroutine `wave()` on page ??. This routine can be used if we calculate the time update in a more complicated fashion, like Chebychev recursion.

Time Update

For the time update we have the basic choice to what order in time we carry out our update. The straight forward approach uses a Taylor expansion in the neighbourhood of the current time time step to calculate the next time step. The simplest update is a 2nd order update, which is shown in used in the subroutine `update2()` on page ??.

```
# update2
# second order time update
```

```
#%
subroutine update2( dt2, u , bnd )
#include "commons"
real                dt2
WFIELD(u)
WFIELDLAY(u)
BOUND(bnd)
BOUNDLAY(bnd)

do ic = 1, ncomp {
  where(bnd==INTERIOR .or. bnd==FREE)
  u(3,ic,:,:,:)=2.*u(2,ic,:,:,:)-u(1,ic,:,:,:)+dt2*u(3,ic,:,:,:)
  u(1,ic,:,:,:)=u(2,ic,:,:,:)
  u(2,ic,:,:,:)=u(3,ic,:,:,:)
  endwhere
}

return
end
```

If we want to calculate to higher orders of time we can use the general version of the time update of order n. That is used in subroutine `update()` on page ??, which incorporates `update2()` on page ?? as the first step in the `1..norder` loop.

```
# update
# time update using Taylor expansion of order n
# that allows multiple source with different signature
#
# default will be one step of this routine per timestep
%

subroutine update(norder,u,nts,bnd,ncomp,dt2)
integer          norder,  nts,    ncomp
real                   u(:,:,:,:,:),   dt2
integer                bnd(:,:,:)
integer dtn,n,ic,nfac

nfac=1
dtn = 1.
do n=1,norder {                         #the actual order is 2,4,6,8  even
  nfac = nfac*2*n
  dtn = dtn*dt2
  sourceu = sourcef(:,:,:,n)            # time differentiated sourcef
  call wave(u,3,..,sourceu,...)         # apply -L^2+F
  do ic=1,ncomp {
       u(3,ic,:,:,:) = dt2*u(3,ic,:,:,:)*2./nfac
  }
}
do ic=1,ncomp {
   where(bnd==0 | bnd==1 )
   u(3,ic,:,:,:) = 2.*u(2,ic,:,:,:)-u(1,ic,:,:,:)+u(3,ic,:,:,:)
   u(1,ic,:,:,:) = u(2,ic,:,:,:)
   u(2,ic,:,:,:) = u(3,ic,:,:,:)
```

```
   endwhere
}

return
end
```

The Chebychev time update repeatedly calculates $\Box u$. The key is that a recursion can be used to update the time step to high accuracy and thus the time step interval becomes less of an issue. It is very convenient, that it incorporates simpler schemes as special cases, just as loop limits.

REVERSE TIME MIGRATION

For reverse time migration one step of the algorithm loop on page 296 is carried out to simulate a shot. Then that synthetic wave field is backward propagated, simultaneously with the real recorded data, which are fed in time-reverse at the geophone locations. At each time step the cross correlation between the strains of the two wavefields is summed for each spatial position.

Playing with different media and space dimensions

Here are is a button for a reader, who reads this article on a computer screen. He can test the modeling code that uses the above subroutines. I have only a short modeling program , which runs and displays wave fronts on your screen as they are calculated. Press the button for a movie.

EXAMPLES

The outlined modeling program lends itself easily to comparison between different types of modeling , for example acoustic versus elastic modeling. Since the same pieces of code are used one can be sure that differences in the modeling results are most likely produced medium properties itself and not by artifacts introduced by the difference in modeling programs.

Marmousi

As an example I used the Marmousi model with its original velocity and density specification. The acoustic modeling was carried out with it (Figures 2 and 3). For the elastic modeling I created s-wave velocities assuming a constant Possion's ratio. The elastic modeling results are shown in Figures 4 and 5. In both cases pressure sources were used and horizontal and vertical displacement components were recorded on the surface.

Figure 6and 7 compare the divergence of both types of modeling. The divergence is computed from the wave field components identically. It clearly shows that as expected pure acoustic modeling cannot account for mode converted waves. A realistic modeling of the earth should include mode converted waves (p, sv and sh). Since mode conversions are absent in the pure acoustic modeling, the amplitudes are different of even the "primary" waves. No energy is leaking into other modes, but stays within the p wave.

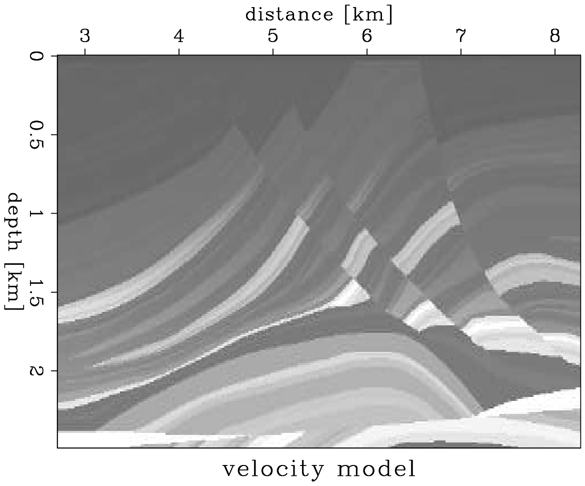

Figure 1: Marmousi subsurface velocity model. martin3-fig1 [NR]

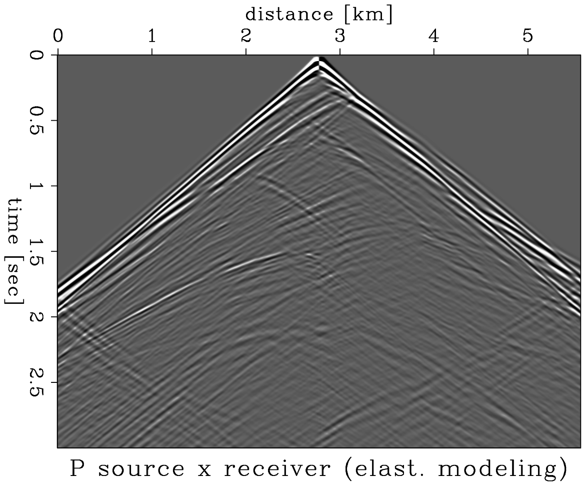

Figure 2: Elastic modeling with a pressure source recorded into a horizontal receiver. martin3-fig6 [NR]

A borehole source

As a second modeling example I used a vertical component source placed in a bore hole. The medium is homogeneous except the borehole, which is fluid filled and three extended point scatterers. In the first case I used an impulsive source, in the second case I used a continuous source, as would have been produced by a drill bit. There is a free surface at the top of the medium and resonances occur in the borehole. Clicking on the figure buttons will bring up movies showing wave propagation in that medium.

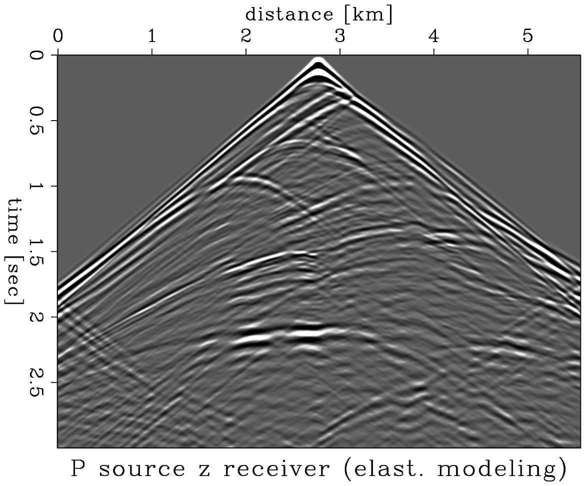

Figure 3: Elastic modeling with a pressure source recorded into a vertical receiver. martin3-fig7 [NR]

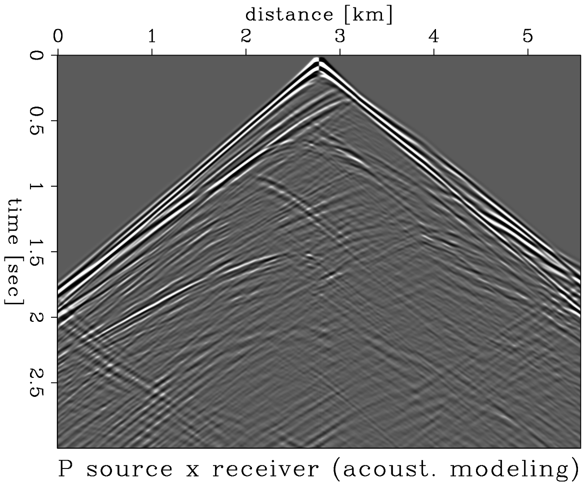

Figure 4: Acoustic modeling with a pressure source recorded into a horizontal receiver. martin3-fig8 [NR]

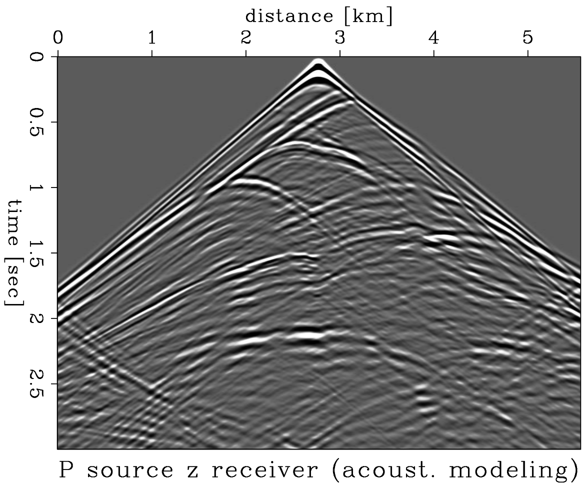

Figure 5: Acoustic modeling with a pressure source recorded into a vertical receiver. martin3-fig9 [NR]

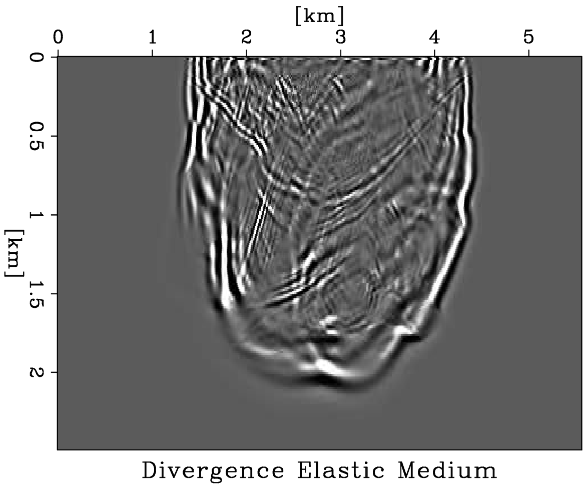

Figure 6: Divergence computed from elastic modeling schemes. There are mode converted waves. Compare it to the acoustic modeling. martin3-fig3 [NR]

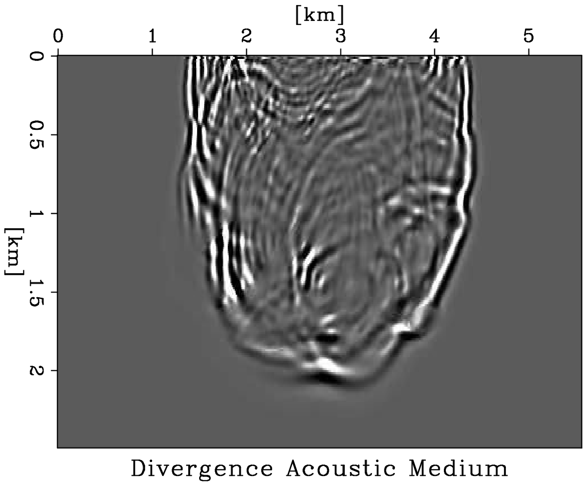

Figure 7: Divergence computed from acoustic modeling schemes. The acoustic modeling does not show mode converted waves.Compare it to the elastic modeling. martin3-fig4 [NR]

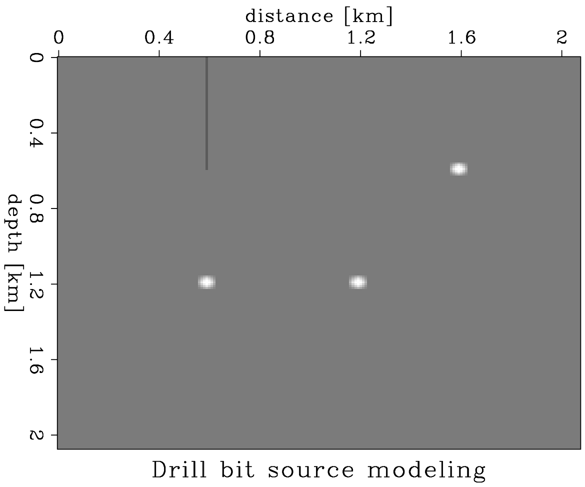

Figure 8: Medium model with a fluid filled borehole and three extended point scatterers. martin3-scatter [NR]

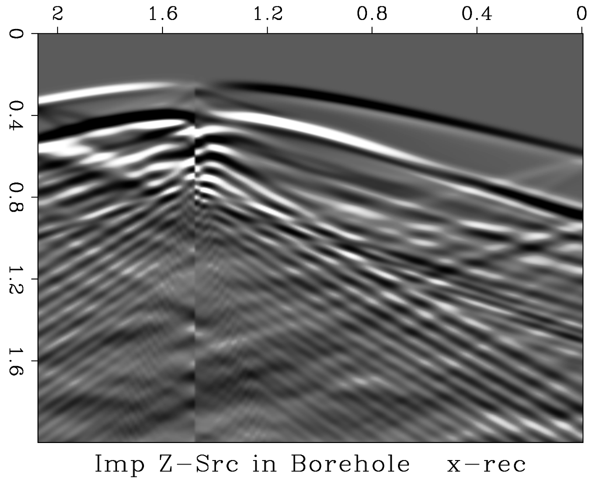

Figure 9: Horizontal component seismogram recorded at the surface using an impulsive source. martin3-xseis.imp [NR]

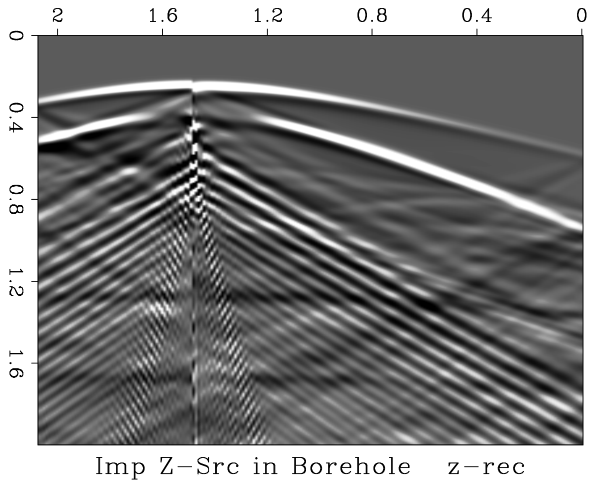

Figure 10: Vertical component seismogram recorded at the surface using an impulsive source. martin3-zseis.imp [NR]

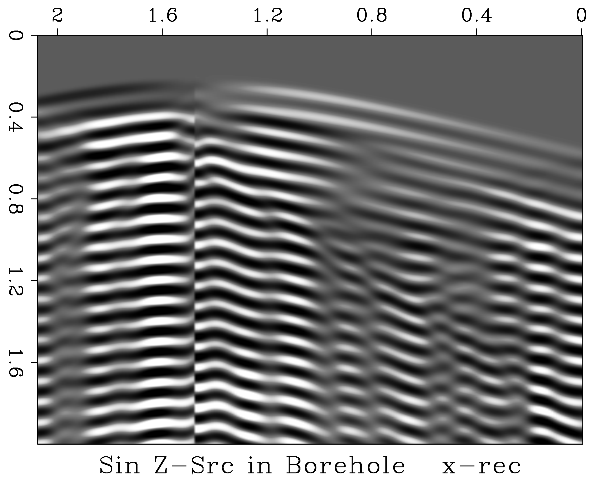

Figure 11: Horizontal component seismogram recorded at the surface using a sinusoidal source. martin3-xseis.sin [NR]

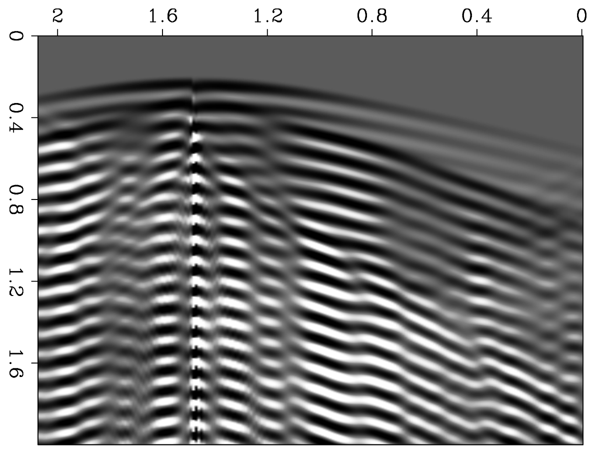

Figure 12: Vertical component seismogram recorded at the surface using a sinusoidal source. martin3-zseis.sin [NR]

EXPANDABILITY

The wave equation modeling modules I described here are easy to maintain and easily run on other computers than what we have at SEP, as long as a F90 compiler or a F90-F77 conversion tool is available. They are open ended and can be put together in many different ways according to changing needs. Other coworkers can reuse those pieces or add their own into the library. A common base is given by the object types mentioned earlier.

CONCLUSIONS

I have consolidated my modules for acoustic nd elastic 2D and 3D modeling and migration algorithms. Their numerical behaviour closely resembles the mathematical formulas. These building blocks can be put together in many different variations. Other people can easily contribute and expand the existing set. Only one source code has to be maintained. Of course the computational speed is slower than for special purpose programs, but those modules are easy to understand and code is not redundant. I see this as a big improvement compared with my existing code and will help me to show algorithms in manageable, easy to understand pieces.

REFERENCES

Cunha, C., 1991, Angle-dependent reflectivity from elastic reverse time migration: SEP–**72**, 111–122.

Dunbar, G., 1992, An implementation of operators and spaces in C++: SEP–**75**, 473–496.

Etgen, J., 1989, Accurate wave equation modeling: SEP–**60**, 131–148.

Karrenbach, M., 1991, Prestack reverse time migration in anisotropic media: SEP–**70**, 113–122.

Mikhailenko, B., 1991, personal communication:.

Muir, F., 1991, Experience with Connection Machine Fortran: SEP–**70**, 53–58.

Nichols, D., Dunbar, G., and Claerbout, J. F., 1992, The C++ language in physical science: SEP–**75**, 455–472.