New seismic modelling facilities in Matlab

Gary F. Margrave

ABSTRACT

The seismic modelling capabilities available in the CREWES Matlab toolbox have been substantially upgraded. These capabilities expand an already strong interactive facility to bring a large variety of useful tools to the explorationist or researcher.

The v(z) raytracing facility has a fast ray-shooting algorithm that allows an efficient, iterative solution to the two-point raytracing problem. Specialty routines are available to trace P-P and P-S primary reflections for any acquisition geometry (including source gathers, receiver gathers, VSPs, and OBC). Traveltimes and ray parameters are determined but not amplitudes. A general-purpose interface is available to trace arbitrary multi-modes that have any number of mode conversions and up or down legs.

A v(x,z) raytracing capability has been developed that solves the ray tracing differential equation on a 2D grid. This method uses a 4^{th} order Runge-Kutta solver to create a very general ray-shooting algorithm. Normal incidence raytrace migration and modelling capabilities have been developed using this raytracer.

The acoustic finite-difference facility has been improved and many bugs were fixed. Based on the 2D variable-velocity scalar-wave equation, this modelling facility uses time-stepping to advance a wavefield. Both 2^{nd} and 4^{th} order Laplacians are available and Clayton-Engquist absorbing boundaries have been implemented. Any 2D source or receiver geometry can be simulated including such effects as arrays, VSPs, and topography. An exploding reflector option also has been created to allow simple modelling of stacked sections.

An interactive picking facility has been built into the basic seismic viewer. This facility allows interactive normal-incidence raytrace modelling and migration to be done interactively. Picks can be made on depth or time sections and the corresponding raypaths are drawn in depth.

INTRODUCTION

A number of new seismic modelling facilities have been created in Matlab. These include: raytracing for v(z), raytracing for v(x,z), full waveform modelling by diffraction summation, and acoustic finite difference modelling. Of these, the first and last have existed previously but are now extensively upgraded. The v(x,z) raytracer and the diffraction summation modelling are completely new.

The v(z) raytracing facility is a fast, flexible raytracer that determines traveltimes in isotropic, horizontally layered media. It can shoot fans of rays or perform two-point raytracing. (Two-point raytracing means that a ray is traced through specific

beginning and ending points. Ray shooting means that the starting point and take-off angle of the ray are prescribed but its ending point is not.) Functions are provided for automatic determination of traveltimes of primary reflections in shot-record geometry for P-P, S-S, P-S, and S-P modes. With slightly more effort, an arbitrary multimode can be traced and other geometries (such as VSP) can be modelled.

The v(x,z) raytracer can shoot rays through an arbitrarily variable velocity field (in 2D) and thereby determine traveltimes. It works by solving the differential equation of rays on a spatial grid. Rays are stepped at constant-time increments across the grid using a fourth-order Runge-Kutta solver (Press et al., 1992). Tools are provided for normal incidence modelling and migration.

Full-waveform, zero-offset modelling is provided with the *event* facility. This produces high-resolution synthetics by superimposing zero-offset diffraction responses. Diffraction hyperbolae can be superimposed along an arbitrary track in (x,z) and simple geometric correction factors are included. This is useful for studying the performance of migration algorithms and for documenting the response of simple geologic structures. This facility is not discussed in this paper but is fully described in chapter 4 of Margrave (2000).

Finally, the acoustic finite difference facility, described in Youzwishen and Margrave (1999) has been updated and improved. Numerous bugs have been repaired, performance improved, and function interfaces have been streamlined. This is a very flexible facility that can model acoustic wave propagation in heterogeneous, isotropic media. Sources and receivers can be placed in arbitrary locations so that source records, VSPs, and cross-well geometries are all easily handled. Also, an exploding reflector function is provided for quick simulation of stacked seismic sections.

THE V(Z) RAYTRACING FACILITY

Technical description

Raytracing in constant velocity and linear gradient media can be done with analytic expressions (e.g. Slotnick 1959). These analytic expressions in the previous section produce first-order realism by including the effects of ray bending when v(z) is a linear function of depth. However, more accurate results are often desired for complicated v(z) such as those that result from well log measurements. In this case the only way to proceed is through a numerical implementation of the integral equations for traveltime and depth known from elementary seismology (e.g. Shearer, 1999). For a ray identified by its ray parameter (horizontal slowness) p, the expression for traveltime from depth z_1 to depth z_2 is

$$t(p) = \int_{z1}^{z2} \frac{1}{v(z)\sqrt{1 - p^2 v^2(z)}}\, dz \qquad (1)$$

and the horizontal distance travelled is

$$x(p) = \int_{z1}^{z2} \frac{pv(z)}{\sqrt{1 - p^2 v^2(z)}} dz \,. \tag{2}$$

A computer implementation of these equations can be done by approximating v(z) by a set of N discrete layers v_k, k=1,2,…N . For a ray that travels from the top of layer 1 to the bottom of layer N (layer number increases with z), the traveltime expression becomes

$$t(p) = \sum_{k=1}^{N} \frac{\Delta z_k}{v_k \sqrt{1 - p^2 v_k^2}} \tag{3}$$

and the distance equation transforms to

$$x(p) = \sum_{k=1}^{N} \frac{p v_k \Delta z_k}{\sqrt{1 - p^2 v_k^2}} \,. \tag{4}$$

These expressions can be vectorized and implemented very efficiently in Matlab. This is done in the function *shootray* (italics will be used to denote Matlab function names). The quantity pv_k is easily shown to be $\sin\theta_k$ where θ_k is the angle the ray makes with the vertical in the k^{th} layer (Figure 1). For a single p value, let **sn** denote a column vector of $\sin\theta_k$, **cs** a similar vector of $\cos\theta_k$, and **dz** a similar vector of layer thicknesses. Then equation (4) is implemented with the single line of Matlab code as: **x=sum((sn.*dz)./cs)**. This expression uses the array multiplication and division operators (**.*** and **./**) that perform element-by-element operations on matrices. The **sum** operator does the obvious summation. The same single line of code can trace M rays simultaneously if **sn**, **cs**, and **dz** are extended to matrices with one ray per column. Then, the resulting **x** is a row vector with one entry per ray. This operation is extremely fast and, for traveltimes, equation (3) can be implemented with similar efficiency. These ideas are implemented in *shootray* and form the basic computation module for v(z) raytracing.

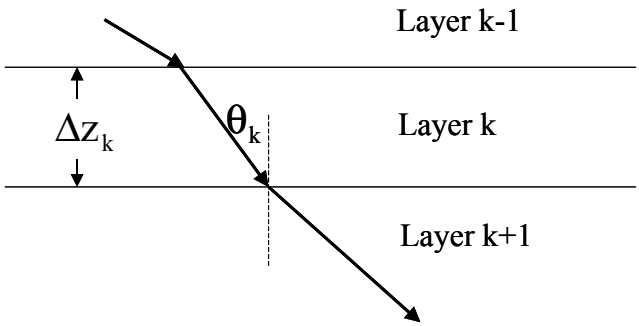

Figure 1. A ray is traced across a set of horizontal layers.

The procedure just described is known as "ray shooting" because the starting point and direction of the ray are specified but the final position is not known until the calculation is completed. Often, it is desired to trace rays between two specific points such as a source and receiver, perhaps via a reflection at a specific depth. For general

v(z), there is no known solution technique that solves this two-point raytracing problem in one step. Instead, an iterative procedure must be used. Suppose it is desired to trace a fan of rays from some fixed point, x_1, at the top of layer 1 to a point x_2 at the bottom of layer N. The *shootray* procedure can be used to shoot a fan of M rays from x_1 that is estimated to bracket the point x_2. Assuming that $x_2 > x_1$, the fan can have extremal p values of $p_1=0$ and $p_M=v_{max}^{-1}$ that, if it is indeed possible to trace a ray to x_2, will bracket x_2. Suppose it is found that rays p_k and p_{k+1} (of the fan of M rays, $k \in \{1,2,\ldots M-1\}$) bracket the point x_2, then a new fan of M rays can be shot with p_k and p_{k+1} as the extremal ray parameters. This procedure can then be repeated as often as desired until a ray is found that comes within an arbitrary "capture radius" of x_2. Thus the two-point problem of shooting a ray across of stack of layers can be solved to any desired precision. However, a solution is not guaranteed because there can be "shadow zones" where classical rays cannot penetrate.

Matlab function *traceray_pp* uses the procedure just described to trace a reflection from a source at x_1 via a reflector at an arbitrary depth z_r to a receiver at x_2. The reflector depth need not be a layer boundary. Rather than trace a ray down to the bottom of a stack of N layers and then back up, an equivalent problem using 2N layers and one-way raytracing is formulated and solved (Figure 2). If the reflector depth occurs within layer N, then that layer is reshaped to terminate at z_r. A stack of 2N layers is then formed by reversing the order of layers 1→N and placing them beneath layer N. That is, the new stack has the layer sequence numbers: 1,2, … N-1, N, N, N-1, … 2, 1. Then, the two-point procedure described in the preceding paragraph is performed to trace a ray from x_1 at the top of layer 1 to x_2 at the bottom of layer 2N. This ray will have the same traveltime and ray parameter as that which solves the original problem.

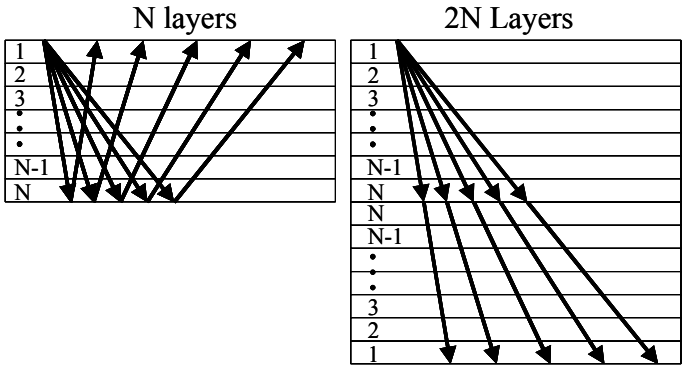

Figure 2. The two-point problem of tracing a fan of rays down to a reflector and back up to receivers through N layers (left) is equivalent to tracing the rays directly through a stack of 2N layers (right).

Function *traceray_ps* does a similar process to solve the problem of tracing a P-S reflection. The only difference is that the second (inverted) stack of N layers in the 2N layer stack is assigned S-wave velocities. Thus two velocity models must be supplied, for P and S waves, and they may have completely different layer boundaries. Once the 2N layer stack is built, solving the two-point problem across it solves the

desired P-S reflection problem. The same function can also trace an S-P reflection by simply reversing the meaning of P and S wave velocities.

Both functions *traceray_pp* and *traceray_ps* can accommodate sources and receivers at different depths by simply altering the layer stack to only include those layers actually traversed on the down and up legs.

Function *traceray* uses this process to trace a general multimode through a layered medium. The multimode is described by a raycode that is a list of ordered pairs, (z_m,i_m), where z_m is the depth at the start of a ray segment and i_m is either 1 (for a P-wave) or 2 (for an S-wave). For example, the raycode [0, 1; 1000, 2; 500, 2; 1000, 1; 0, 1] indicates a P-S-S-P mode. It is a P-wave from 0 to 1000 m, an S-wave from 1000 to 500 m, and S-wave from 500 to 1000 m, and finally a P-wave from 1000 to 0 m. (Note that the final value of i_m is meaningless.) This problem is solved with the same stratagem as before by building an equivalent stack of layers and tracing rays through it one-way. In this way, a completely arbitrary multimode can be traced through a v(z) medium.

This v(z) raytrace facility is quite general but has a major theoretical limitation. It cannot turn a ray around by refraction. However, the gridded v(x,z) facility can do so though it currently does not have two-point raytracing.

Examples

This section describes a number of examples for the v(z) raytrace facility. These are all taken directly from the m-file *raytrace_demo* that is included with the CREWES software distribution. Thus, any of them can be recreated (and more that aren't shown here) by simply running that demo. Examining the source code should be sufficient so see how any of them are done.

For the velocity model shown in Figure 3, consider the task of tracing P-P and P-S reflections from a target at zd=3000 m depth. This velocity model includes a 200m thick water layer at the top so OBC geometry is adopted. Let the source be at zsrc=50 m and the receivers on the ocean bottom at zrec=200 m. Also let the receivers be at offsets xoff=[1000, 1100 1200 … 3000] for a total of 21 receivers. Then Figure 4 shows the Matlab code required to trace the P-P reflection and produce Figure 5. The code example is a bit involved because it was decided to show all of the code required to produce the fully annotated plot. The actual raytracing is done on the third line. Of the inputs to *traceray_pp*, the first two are the P-wave velocity and layer depths, the next four specify the recording geometry, and the next six numeric values are respectively: the capture radius (10 meters), a flag indicating that the initial ray fan is to be determined automatically, the number of iterations to attempt before non-convergence is assumed (10), a flag indicating that the final times are to be improved by linear interpolation between the captured ray and the next closest ray, a flag requesting that information about failed rays be printed on the screen, and a flag requesting a plot of the raypaths in the current figure window. (To see a complete description of these parameters, run Matlab and type "help traceray_pp" at the command line.) The returned variables from *traceray_pp* are a vector of traveltimes

(t) and a vector of ray parameters (p). The traveltimes are plotted versus offset on line 12 of this example.

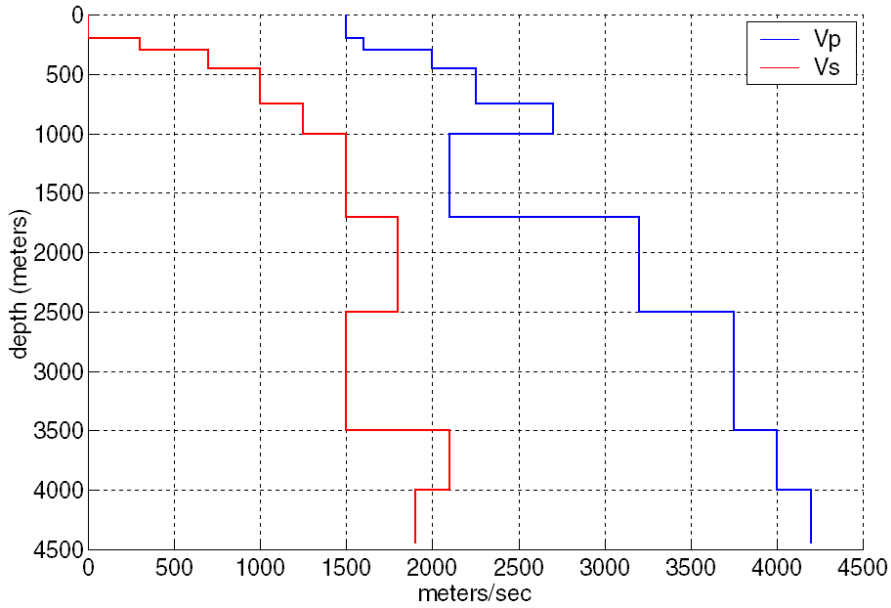

Figure 3. A simple layered medium is shown as characterized by is P-wave velocity curve (right) and its S-wave velocity curve (left). Note the 200 m water layer at the top.

```
1.  figure;subplot(2,1,1);flipy
2.  %Trace P-P rays and plot in upper subplot
3.  [t,p]=traceray_pp(vp,zp,zsrc,zrec,zd,xoff,10,-1,10,1,1,2);
4.  %put source and receiver markers
5.  line(xoff,zrec*ones(size(xoff)),'color','b','linestyle','none','marker','v')
6.  line(0,zsrc,'color','r','linestyle','none','marker','*')
7.  %annotate plot
8.  title('OBC simulation, P-P mode, water depth 200 meters')
9.  xlabel('meters');ylabel('meters');grid
10. %plot traveltime versus offset in lower subplot
11. subplot(2,1,2);flipy;
12. plot(xoff,t);grid;xlabel('meters');ylabel('seconds')
13. xlim([0 3000])
```

Figure 4. This sequence of Matlab code uses the velocity model of Figure 3 to trace P-P rays and create figure 5.

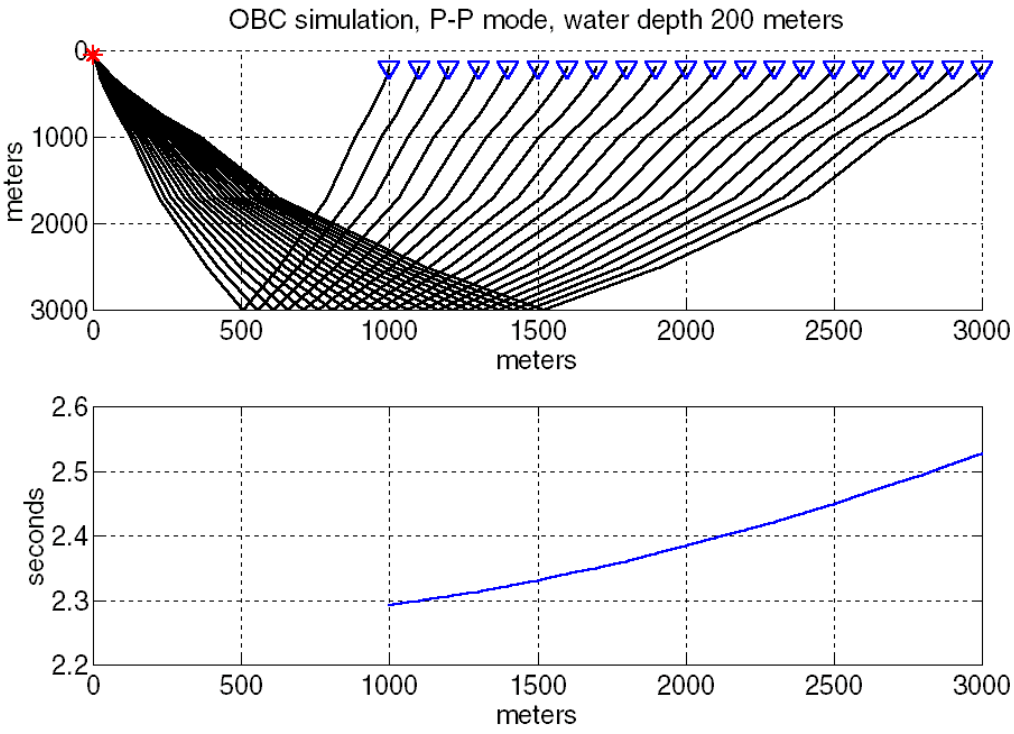

Figure 5. Running the code of Figure 4 creates this figure using the velocity model of Figure 3. In the top frame are the actual raypaths as plotted by *traceray_pp* in line 3. In the bottom frame, the traveltimes are plotted versus offset.

The code snippet of Figure 6 demonstrates the use of *traceray_ps* to model the P-S reflection corresponding to the P-P reflection just discussed. As before, the rays are both traced and plotted in the upper half of Figure 7 by line 3. The syntax to invoke *traceray_ps* is almost identical to *traceray_pp* except that there are two additional input parameters in the third and fourth position to describe the S-wave velocity structure.

```
1.  figure;subplot(2,1,1);flipy
2.  %Trace P-S rays and plot in upper subplot
3.  [t,p]=traceray_ps(vp,zp,vs,zs,zsrc,zrec,zd,xoff,10,-1,10,1,1,2);
4.  %put source and receive markers
5.  line(xoff,zrec*ones(size(xoff)),'color','b','linestyle','none','marker','v')
6.  line(0,zsrc,'color','r','linestyle','none','marker','*')
7.  %annotate plot
8.  title('OBC simulation, P-S mode, water depth 200 meters')
9.  grid;xlabel('meters');ylabel('meters');
10. subplot(2,1,2);flipy;
11. plot(xoff,t);grid;xlabel('meters');ylabel('seconds');
12. xlim([0 3000])
```

Figure 6. This sequence of Matlab code uses the velocity model of Figure 3 to trace P-S rays and create figure 7.

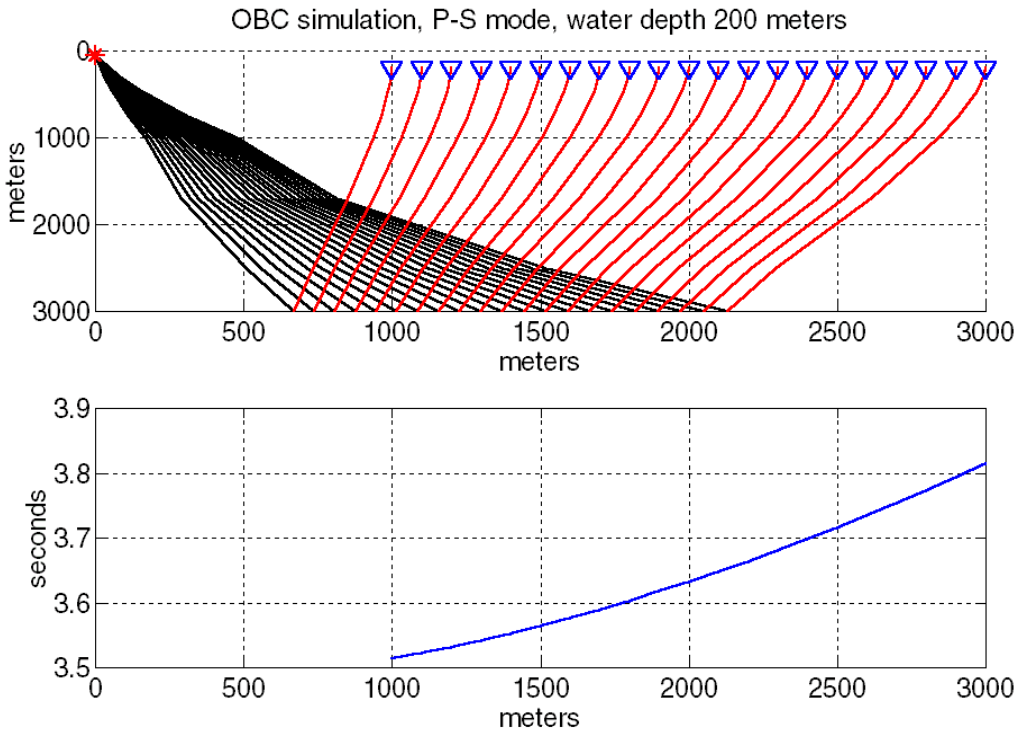

Figure 7. Running the code of Figure 6 creates this figure using the velocity model of Figure 3. In the top frame are the actual raypaths as plotted by *traceray_ps* in line 3. In the bottom frame, the traveltimes are plotted versus offset.

Now, consider the task of computing the P-S conversion point as a function of depth. For this purpose, *traceray_ps* can be run repeatedly in a simple loop. A loop is necessary because the function is written to handle a single reflection at a time with an array of receivers. Let the reflector depth be given as a vector, zd=[250, 300, 350, …3000] m, and fix the receiver offset at a single scalar value, xoff=1500 m. Then the code snippet of Figure 8 can be used to produce Figure 9. Lines 2-5 form a loop over reflector depth and for each depth a P-S ray is traced to offset 1500 m. (For this example the times and ray parameters are not saved after each iteration though this could easily be changed.) On line 4, *traceray_ps* is invoked in much the same manner as before except that some of the numeric input arguments have been changed. The meaning of the final six arguments is (from left to right): the capture radius (10 m), a flag indicating that the initial ray parameter fan will be the final fan used on the previous call, the number of iterations (30), the linear optimization flag, the flag to display information about failed rays, and the flag requesting that the raypaths be drawn. On the first iteration this final flag is set to 1 to request a new figure and is thereafter 2 indicating that drawing should continue in this figure.

```
1.  %loop over depth and show conversion point
2.  for kk=1:length(zd);
3.      if(kk==1)dflag=1;else;dflag=2;end
4.      [t,p]=traceray_ps(vp,zp,vs,zs,zsrc,zrec,zd(kk),xoff,10,-2,30,1,1,dflag);
5.  end
6.  %draw source and receiver symbols
7.  line(xoff,zrec,'color','b','linestyle','none','marker','v')
8.  line(0,zsrc,'color','r','linestyle','none','marker','*');
9.  %annotate plot
10. title('OBC simulation, P-S mode, fixed offset CCP determination');grid;
```

Figure 8. For the velocity model of Figure 3 and a source-receiver offset of 1500 m, this code uses a looping structure to determine the P-S conversion point as a function of depth. It creates Figure 9.

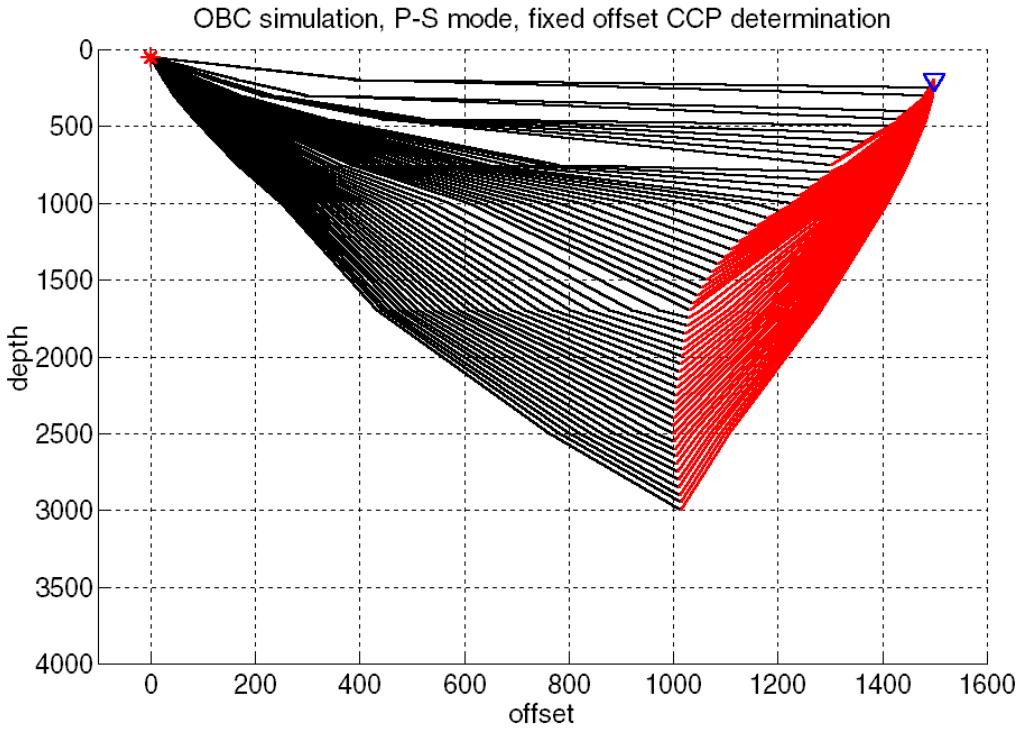

Figure 9. For the velocity model of Figure 3 and a source-receiver offset of 1500 m, the P-S conversion point is indicated as a function of depth. This was created by the code in Figure 8.

Next, consider the task of calculating P-P and P-S reflections for an offset VSP. Let the source be at the surface and offset 1500 m from the well. The receivers are in the well from 500 to 2500 m at 100 m intervals. Let the P-wave velocity structure be given by $v_p(z)=1800 + .6z$ with $v_p/v_s=2$. The code to model a P-P arrival from a reflection at zd=3000 m is shown in Figure 10 and the resulting plot is in Figure 11. Line 2 creates the velocity model for both P and S and line 5 preallocates a vector to hold the traveltimes. Lines 7-9 loop over receiver depth and call *traceray_pp* for each receiver. As before, *traceray_pp* draws the raypaths in the upper part of Figure 11 while the traveltimes are plotted in the lower part of the same Figure by the command on line 18.

The P-S reflection for the VSP geometry is drawn by the code in Figure 12. The strategy is identical to the P-P case with the only difference being that *traceray_ps* is called in the loop instead of *traceray_pp*.

Most recording geometries can be accommodated by writing a suitable loop. For example, a VSP with a non-vertical borehole simply requires that the both receiver coordinates (x,z) be changed with each iteration. Crosswell experiments could be done by changing both source and receiver depths at each iteration.

```
1.  %build the velocity model
2.  zp=0:10:4000;vp=1800+.6*zp;vs=.5*vp;zs=zp;
3.  %P-P offset VSP
4.  figure;subplot(2,1,1);flipy
5.  t=zeros(size(zrec)); %preallocate t
6.  %loop over receiver depth
7.  for kk=1:length(zrec);
8.      [t(kk),p]=traceray_pp(vp,zp,zsrc,zrec(kk),zd,xoff,10,-2,30,1,1,2);
9.  end
10. %draw source and receiver symbols
11. line(xoff,zrec,'color','b','linestyle','none','marker','v')
12. line(0,zsrc,'color','r','linestyle','none','marker','*')
13. %annotation
14. title([' VSP Vertical gradient simulation, P-P mode '])
15. grid;xlabel('meters');ylabel('meters');
16. %plot traveltime versus depth
17. subplot(2,1,2);
18. plot(t,zrec);xlabel('seconds');ylabel('depth (meters)')
19. grid;flipy;ylim([0 3000])
```

Figure 10. This code models a P-P reflection for offset VSP geometry. The velocity model is $v_p(z)=1800 + .6z$ with $v_p/v_s=2$. The result is Figure 11.

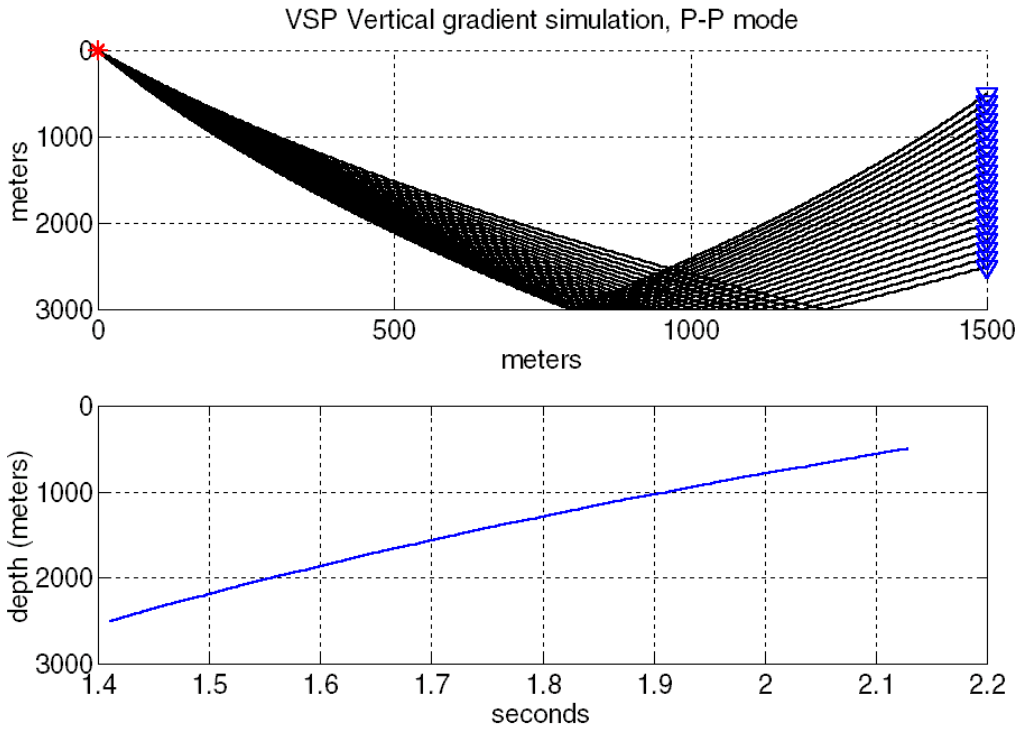

Figure 11. A P-P reflection for an offset VSP recording is shown. The raypaths (top) and traveltimes (bottom) were computed by the code in Figure 10.

```
1.  %P-S offset VSP
2.  figure;subplot(2,1,1);flipy;
3.  t=zeros(size(zrec));%preallocate t
4.  for kk=1:length(zrec);
5.        [t(kk),p]=traceray_ps(vp,zp,vs,zs,zsrc,zrec(kk),zd,xoff,10,-2,30,1,1,2);
6.  end
7.  %draw source and reciver symbols
8.  line(xoff,zrec,'color','b','linestyle','none','marker','v')
9.  line(0,zsrc,'color','r','linestyle','none','marker','*')
10. %annotate plot
11. title([' VSP Vertical gradient simulation, P-S mode '])
12. grid;xlabel('meters');ylabel('meters');
13. %plot traveltime versus depth
14. subplot(2,1,2);
15. plot(t,zrec);xlabel('seconds');ylabel('depth (meters)');
16. grid;flipy;ylim([0 3000])
```

Figure 12. This code models a P-S reflection for an offset VSP. It assumes the code of Figure 10 was previously run to create the velocity model. The result is Figure 13.

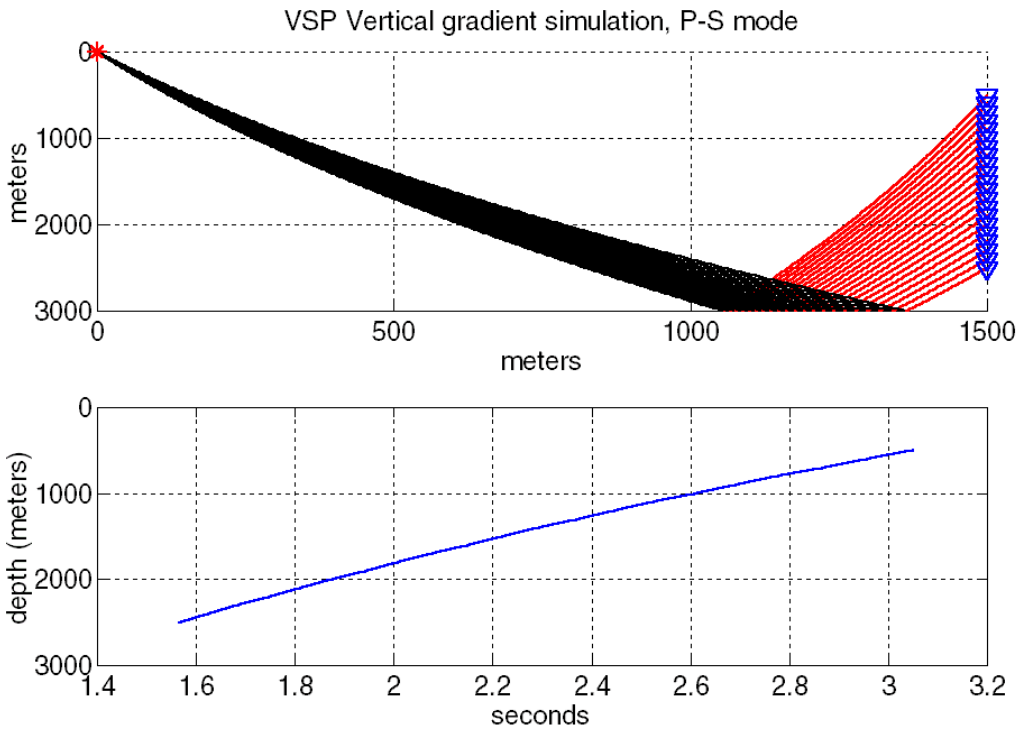

Figure 13. A P-S reflection for an offset VSP is shown. This was created by the code in Figure 12. Compare with Figure 11.

As a final example, consider the task of tracing a complex multimode. That is, let the ray bounce up and down between various depths and change back and forth between P and S. This is possible with the function *traceray_pp*, the most general of the three "trace_ray" functions. All use the algorithm described previously that makes an equivalent stack of layers and traces the ray through it in a single direction. The functions *traceray_pp* and *traceray_ps* are optimized to trace primary (single bounce) reflections from a single source to an array of receivers. Function *traceray* is far more general and can trace a ray that undergoes any number of bounces and mode conversions. The ray is described through a ray code as described previously. For an M-bounce multiple, the raycode is a matrix with M+2 rows and 2 columns. The extra two rows are for the start and end depths. The first column is a list of depths that the ray is to visit and the second is a list of flags indicating P or S mode. The list of depths corresponds either to points of reflection or mode-conversion or both. These depth values need have no connection to layers in the velocity models. Figure 14 shows the raycode (lines 2-3) for a complicated multi-bounce P-wave that starts at depth zero and ends at depth 300 m. The raycode is simply typed in as a list of number pairs separated by a semi-colon (Matlab's row separator). On line 7, *traceray_pp* is invoked with a list of receiver offsets from 1000 to 3000 m. The creation of the P and S-wave velocity models is not shown but is the same linear-gradient medium as for the VSP example of Figure 10. Most of the parameters in *trace_ray* are analogous to those in *traceray_ps* and have already been discussed. However, there are no specifications of source and receiver depth because those are part of the raycode.

The result of running the code snippet of Figure 14 is shown in Figure 15. This is perhaps an unlikely candidate for a physically significant multiple because the raypath is not symmetric. Nevertheless, it is shown to demonstrate the flexibility of the tool. Like all of the "traceray" programs, *traceray* returns traveltimes and ray parameters. The traveltimes are plotted beneath the raypaths in Figure 15.

Figures 16 and 17 document the creation of a similar multimode but with a number of changes of mode. These changes are a P-S conversion at 1300 m, an S-P conversion at 3000 m, and another P-S conversion at 1500 m (on the up leg). The result, shown in Figure 17, is similar to Figure 15 except that the S-wave raypaths are generally steeper than their P-wave counterparts. Also, the total traveltime is significantly greater.

```
1.  %define the ray code for a pure P multiple
2.  raycode=[0 1;1500 1;1300 1;2000 1;1800 1;3000 1;2000 1;2300 1;1000 1;...
3.                                  1500 1; 300 1];
4.  figure;subplot(2,1,1);flipy
5.  %trace the rays
6.  xoff=1000:100:3000;
7.  [t,p]=traceray(vp,zp,vs,zs,raycode,xoff,10,-1,10,1,1,2);
8.  %Source and receiver symbols
9.  line(xoff,raycode(end,1)*ones(size(xoff)),'color','b','linestyle','none','marker','v')
10. line(0,raycode(1,1),'color','r','linestyle','none','marker','*')
11. %annotate
12. title('A P-P-P-P-P-P-P-P-P-P mode in vertical gradient media');grid
13. %Plot traveltimes
14. subplot(2,1,2);flipy
15. plot(xoff,t);grid;xlabel('offset');ylabel('time')
16. xlim([0 3000])
```

Figure 14. This code example illustrates the use pf *traceray_pp* to create a complicated multiple that remains a P-wave at every bounce. The raycode (lines 2-3) defines the M-bounce multiple by a set of M+2 depths (first column) and M+2 flags (second column). The flags are all 1 indicating a P-wave. The result is Figure 15.

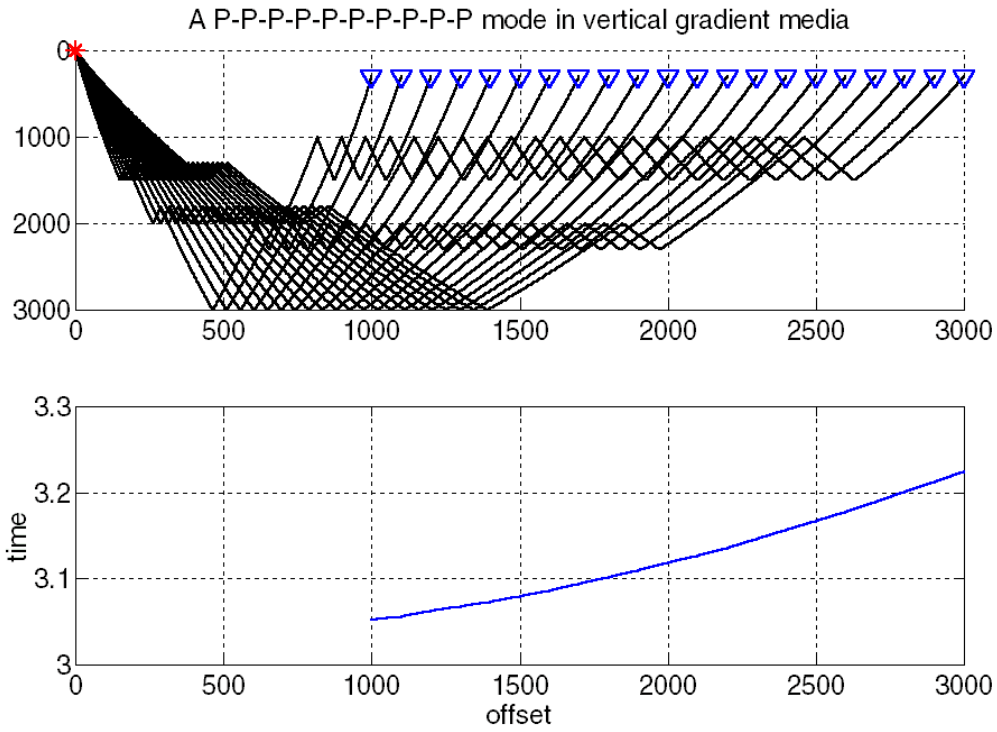

Figure 15. The raypath of a complicated multiple is shown (top) that remains a P-wave at every bounce. The traveltimes are displayed beneath.

```
%define the ray code for a P-S multimode
raycode=[0 1;1500 2;1300 2;2000 2;1800 2;3000 1;2000 1;2300 1;1000 1;...
                                1500 2; 300 1];
figure;subplot(2,1,1);flipy
%trace the rays
xoff=1000:100:3000;
[t,p]=traceray(vp,zp,vs,zs,raycode,xoff,10,-1,10,1,1,2);
%source and receiver symbols
line(xoff,raycode(end,1)*ones(size(xoff)),'color','b','linestyle','none','marker','v')
line(0,raycode(1,1),'color','r','linestyle','none','marker','*')
%annotate
title('A P-S-S-S-S-P-P-P-P-S mode in vertical gradient media');grid
%Plot traveltimes
subplot(2,1,2);flipy
plot(xoff,t);grid;xlabel('offset');ylabel('time')
xlim([0 3000])
```

Figure 16. This code creates a multimode similar to that of Figure 15 except that the raycode requests a P-S conversion at 1300 m, an S-P conversion at 3000 m, and another P-S conversion at 1500 m (on the up leg). The result is Figure 17.

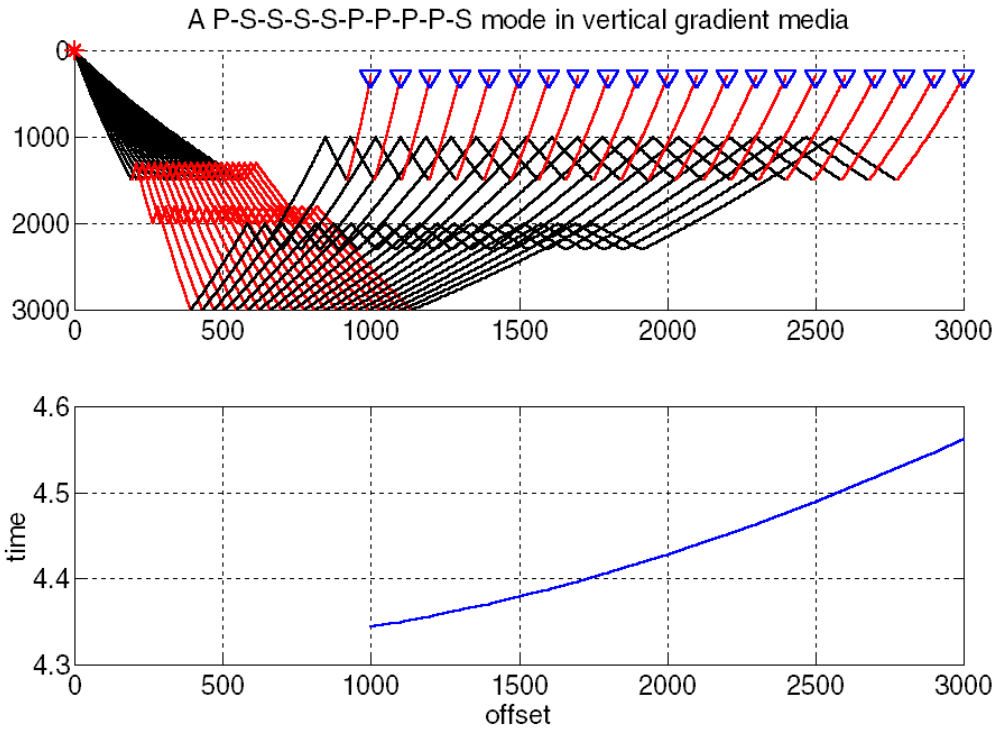

Figure 17. The raypath for a complicated P-S multimode is shown (top) as produced by the code snippet in Figure 16. The corresponding traveltimes are below. Compare with Figure 15.

THE V(X,Z) RAYTRACER

Technical Description

It is often desired to trace rays through media with arbitrary velocity variations. There are at least two popular strategies for doing this. One is to represent the medium as polygonal regions of constant velocity and use analytic methods to raytrace across each polygon. This approach has the virtue of requiring minimal computer resources to represent the velocity model and this is especially important in 3D. The other approach is to use gridded velocity models and trace rays by solving the differential equation, *the ray equation*, for rays. This is the approach taken here and has the virtue of being simple to code but it requires a lot of computer memory to represent the velocity model. Furthermore, the effort required to trace rays is dependent only on the size of the velocity grid not on its complexity. Thus it takes just as long to trace rays through constant-velocity media as through random-velocity media.

It is well known (e.g. Aki and Richards, 1980, or Shearer, 1999) that raypaths can be calculated as the solution to two coupled, first-order, ordinary differential equations

$$\frac{d\vec{x}}{dt} = v^2(\vec{x})\vec{p} \tag{5}$$

and

$$\frac{d\vec{p}}{dt} = -\frac{\vec{\nabla} v(\vec{x})}{v(\vec{x})} = -\vec{\nabla} \ln(v(\vec{x})). \tag{6}$$

In these expressions, $\vec{x}$ is the position vector, $\vec{p}$ is the slowness vector, $v(\vec{x})$ is the velocity field (model), and t is time. These can be combined into a single differential equation by defining the abstract "ray vector", $\vec{r} = [\vec{x}, \vec{p}]$, that obeys the differential equation

$$\frac{d\vec{r}}{dt} = \vec{a} \tag{7}$$

where $\vec{a} = [v^2 \vec{p}, -\vec{\nabla} \ln v]$. Given initial values for $\vec{r}$ and the velocity field, equation (7) can be solved by any of a variety of solution techniques for ordinary differential equations. Perhaps the most common approach is the *fourth-order Runge-Kutta* scheme (Press et al., 1992) and that will be used here. Matlab has a built-in method for this purpose but it was decided to construct another following the discussion in Press et al. because various constraints could be more readily incorporated.

The Matlab implementation is in 2D and requires a velocity matrix giving v(x,z) on a grid with $\Delta x = \Delta z \equiv \Delta g$ and uses the convention that (x=0,z=0) is in the upper left corner (row 1, column 1). Prior to raytracing, the function *rayvelmod* is invoked with arguments being the velocity matrix and the grid spacing Δg. This function creates a number of global variables that will be used repeatedly in the raytracing. These include matrices of $v^2(x,z)$, $\partial(\ln v)/\partial x$, and $\partial(\ln v)/\partial z$ that are needed in equation (7). This pre-computation speeds the raytracing and is especially beneficial if a great many rays are to be traced; however, the cost is three times the memory overhead of the simple velocity matrix. Function *rayvelmod* need only be called once at the beginning of the raytracing unless the velocity model is changed.

The Runge-Kutta solver is contained in function *shootrayvxz*. The input to this function is very simple. It requires a vector of time steps to be taken and the initial values for $\vec{r}$. The time-step vector should contain regular increments from 0 to some maximum at an interval, Δt, of a few milliseconds. Intuitively, a ray should not travel far enough in time Δt to encounter dramatic velocity variations. The initial values of $\vec{r}$ determine the starting point and the take-off angle.

An alternative to *shootrayvxz* that is useful in tracing normal-incidence rays is *shootraytosurf*. This function works in the same way as *shootrayvxz* except that it terminates at z=0 rather than some maximum time.

Examples

These examples are taken from the demonstration program *rayvxz_demo*. Thus, the reader can easily recreate and extend them.

Figure 18 shows the code required to create the velocity model shown in Figure 19. This model has a background velocity of 2700 m/s with several polygonal regions

where the velocity is 3800 m/s. Overtop of everything there has been superimposed a random fluctuation of +/- 500 m/s. In Figure 18, lines 1-5 establish the basic geometry for a grid that will be 100 x 100 with a grid spacing of 10 m. Initially, the grid is enlarged on all sides by the dimensions of a boxcar smoother (nsmooth, line 3) and will later be reduced to 100 x 100 after smoothing. On line 8, the velocity matrix is built and filled with the background velocity of vlow=2700 m/s. Lines 9-10 define the (x,z) coordinates of the vertices of a polygonal region and line 11 invokes *afd_vmodel* to fill this region with the velocity vhigh=3800 m/s. Function *afd_vmodel* was built as part of the finite difference modelling toolbox (afd means acoustic finite difference) for just this purpose. Calling *afd_vmodel* repeatedly allows complicated velocity models to be constructed from the superposition of many polygonal regions. When two polygons overlap, the last one to be constructed determines the velocity in the overlap region. Lines 12-15 define and fill two more polygons. (What appears to be one large polygon in the lower half of Figure 19 is actually the union of two separate polygons.) Finally, line 16 defines a uniformly distributed set of random numbers, ranging from –vdel/2 to +vdel/2, that are added to the velocity matrix on line 17.

```
1.  % Define geometry
2.  nx=100;nz=nx;
3.  dg=10;nsmooth=10;
4.  xb=(0:nx+nsmooth-2)*dg;zb=(0:nz+nsmooth-2)*dg;
5.  x=(0:nx-1)*dg;z=(0:nz-1)*dg;
6.  %Build the velocity model
7.  vlow=2700;vhigh=3800;vdel=1000;
8.  v=vlow*ones(nx+nsmooth-1,nz+nsmooth-1);
9.  xpoly=[max(xb)/2 2*max(xb)/3 1.5*max(xb)/2.5 max(xb)/pi];
10. zpoly=[max(zb)/3 max(zb)/2 2.1*max(zb)/2.5 .4*max(zb)];
11. v=afd_vmodel(dg,v,vhigh,xpoly+max(xb)/4,zpoly);
12. v=afd_vmodel(dg,v,vhigh,zpoly-max(xb)/4,xpoly-max(zb)/4);
13. xpoly=[min(xb) .9*max(xb)  .6*max(xb) min(xb)];
14. zpoly=[.5*max(zb) .6*max(zb) max(zb) .7*max(zb)];
15. v=afd_vmodel(dg,v,vhigh,xpoly,zpoly);
16. vrand=vdel*(rand(nx+nsmooth-1,nz+nsmooth-1)-.5);
17. v=v+vrand;
```

Figure 18. This code example creates the velocity matrix shown in Figure 19. Lines 1-5 establish the basic geometry. Line 8 fills the velocity matrix with vlow. Lines 11,12, and 15 fill three different polygonal regions with the velocity vhigh. Lines 16-17 superimpose a random fluctuation uniformly distributed between +/- vdel/2.

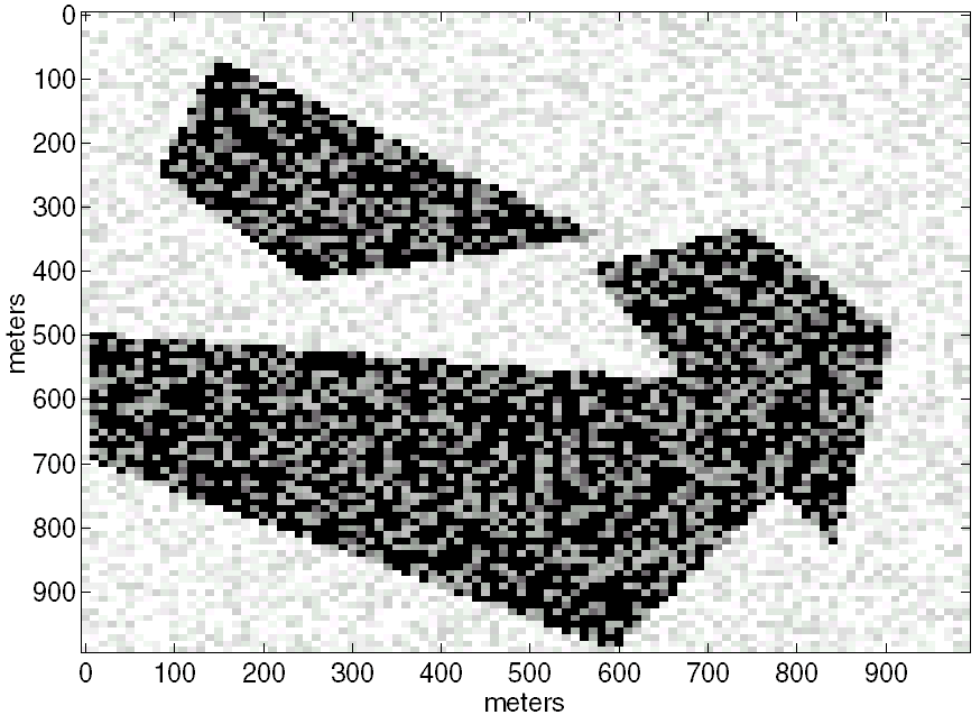

Figure 19. The unsmoothed velocity matrix used in the subsequent raytracings. This model has a background velocity of 2700 m/s with several polygonal regions where the velocity is 3800 m/s. Overtop of everything there has been superimposed a random fluctuation of +/- 500 m/s. The code in Figure 18 created this model.

Figure 20 is a code snippet that illustrates the use of *shootrayvxz* with the velocity model of Figure 19. Prior to running this snippet, the velocity model has been smoothed by convolving it (2D) with a square boxcar of a certain size. (This smoothing operation is not shown.) Line 1 installs the velocity model in the appropriate global variables by calling *rayvelmod*. Lines 2-3 plot the smoothed velocity model in preparation to drawing rays on top of it. Line 5 establishes the tstep parameter that is required by *shootrayvxz*. The step size is set to .004 seconds and the maximum time is calculated as that required to travel vertically across the model at the velocity vlow (2700 m/s). (Experimentation with these parameters is the best way to gain confidence with them.) Lines 7-10 do some preliminary calculations that are needed to determine the initial values of the ray vector for each ray. Line 7 defines the take-off angles for a fan of rays to be from -70° to +70° at increments of 2.5° for a total of 57 rays. Line 8 establishes the point of origin for the ray fan as midway across the velocity model at the very top. Lines 9 and 10 determine the velocity at the point (x_0,z_0).

The actual raytracing happens in the loop on lines 12-16 of Figure 20. The loop iterates once for each ray and on line 13 the initial value of the ray vector is calculated. Recalling the definition of $\vec{r} = [\vec{x}, \vec{p}]$, the initial ray vector, r0, consists of four numbers, the initial coordinates and the initial slownesses. Since each ray has a different takeoff angle, r0 changes each time through the loop but tstep does not. On line 14, *shootrayvxz* traces each ray and returns the variables t and r. The first

variable, t, is just a vector of times along the ray. It is identical to tstep if the ray completed within the model but if the ray encountered the edge, t will be a shortened version of tstep. More useful is r, which is an N by 4 matrix where N is the length of t. The k^{th} row of r contains the ray vector for time t(k). Thus, the x coordinates of the raypath are found in column 1 and the z coordinates in column 2. Lines 15 plots the raypath on top of the active figure, which is the plot produced on line 1 of the smoothed velocity model. The raypath is actually plotted as a line in 3D where the third coordinate is set to a vector of ones. This ensures that the raypath is "in front" of the velocity model and will always render on top even if the figure is zoomed.

```
1.  rayvelmod(v,dx)%install velocity model
2.  plotimage(v-mean(v(:)),x,z)
3.  xlabel('meters');ylabel('meters')

4.  %estimate tmax,dt,tstep
5.  tmax=max(z)/vlow;dt=.004;tstep=0:dt:tmax;

6.  %specify a fan of rays
7.  angles=[-70:2.5:70]*pi/180;
8.  x0=round(nx/2)*dg;z0=0;
9.  indx=near(x,x0);indz=near(z,z0);
10. v0=v(indz,indx);

11. %trace the rays
12. for k=1:length(angles)
13.       r0=[x0 z0 sin(angles(k))/v0 cos(angles(k))/v0];
14.       [t,r]=shootrayvxz(tstep,r0);
15.       line(r(:,1),r(:,2),ones(size(t)),'color','r');
16. end
```

Figure 20. This code plots the velocity model and then traces rays through it. The rays are drawn on top of the model. Results from four different tests with different levels of smoothing are shown in Figures 21-24. The smoothing is not shown in the code but was done by a 2D convolution with a square boxcar.

Figures 21-24 show the results of four different raytracing experiments with different smoothers. In Figure 21, the model is used "as is" with no smoothing and the resulting raypaths are quite chaotic. This cannot be expected to be a physically plausible result because ray theory has been used in a context where it is not valid. That is, since ray theory is a high frequency approximation, it assumes that the velocity field is smooth over the scale of the wavelength of interest. This is clearly not the case with Figure 21. In the next three Figures, the wavefield is smoothed with progressively longer smoothers of 30 m, 50m, and 100m. The raypaths quickly stabilize to a more plausible result. (Precisely how much smoothing should be done for a practical problem such as estimating traveltimes for depth migration is a topic of current research.)

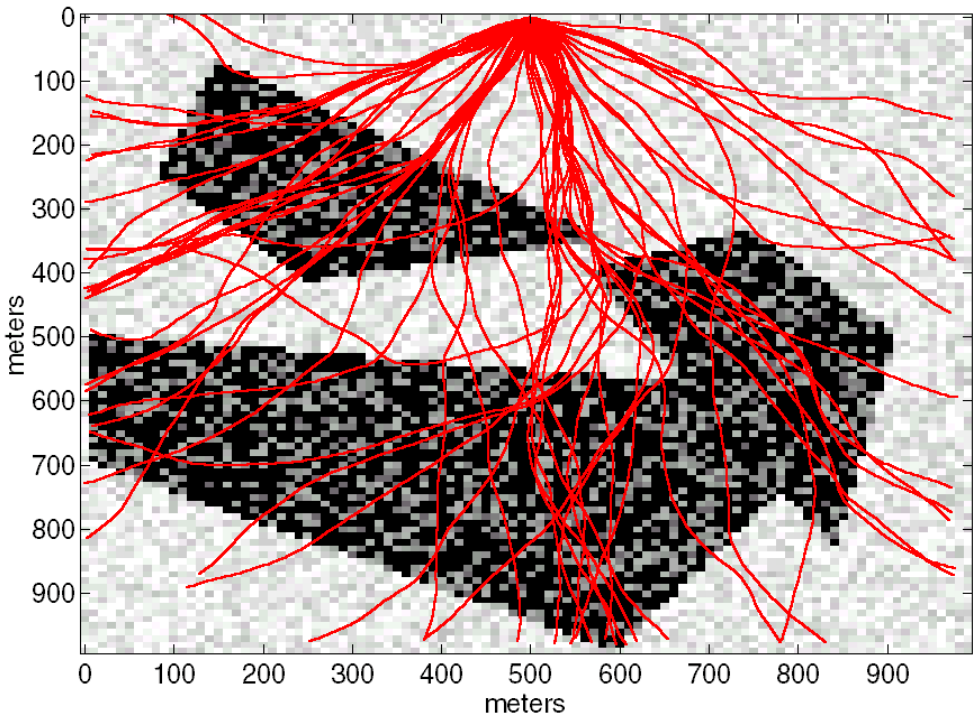

Figure 21. The result of raytracing through the velocity model of Figure 19 with no smoothing.

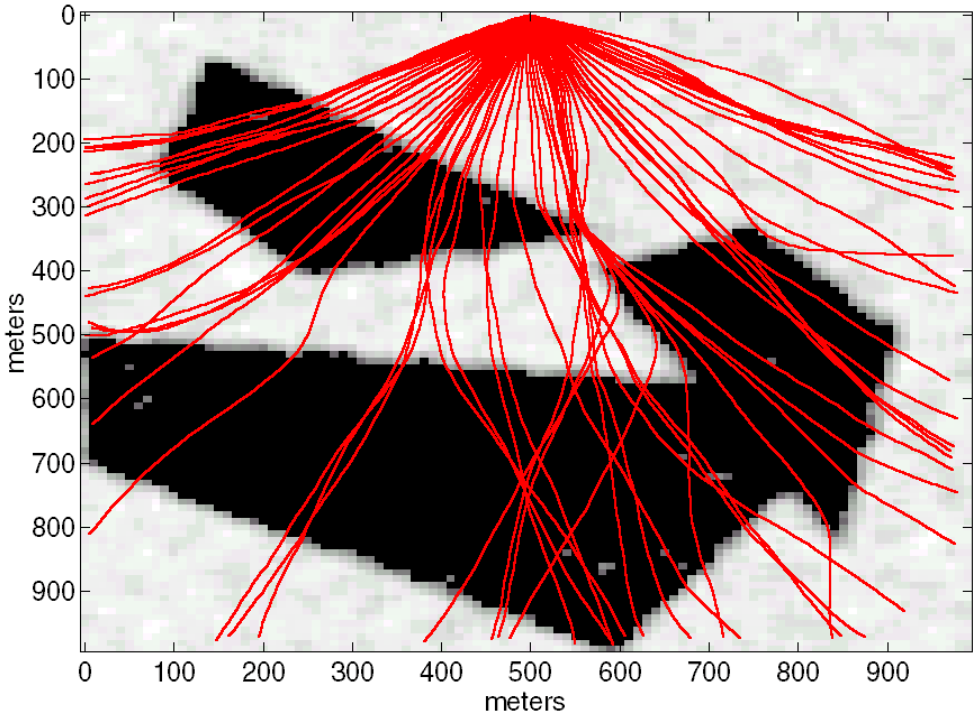

Figure 22. The result of raytracing through the velocity model of Figure 19 after applying a 30 m smoother (3 pts).

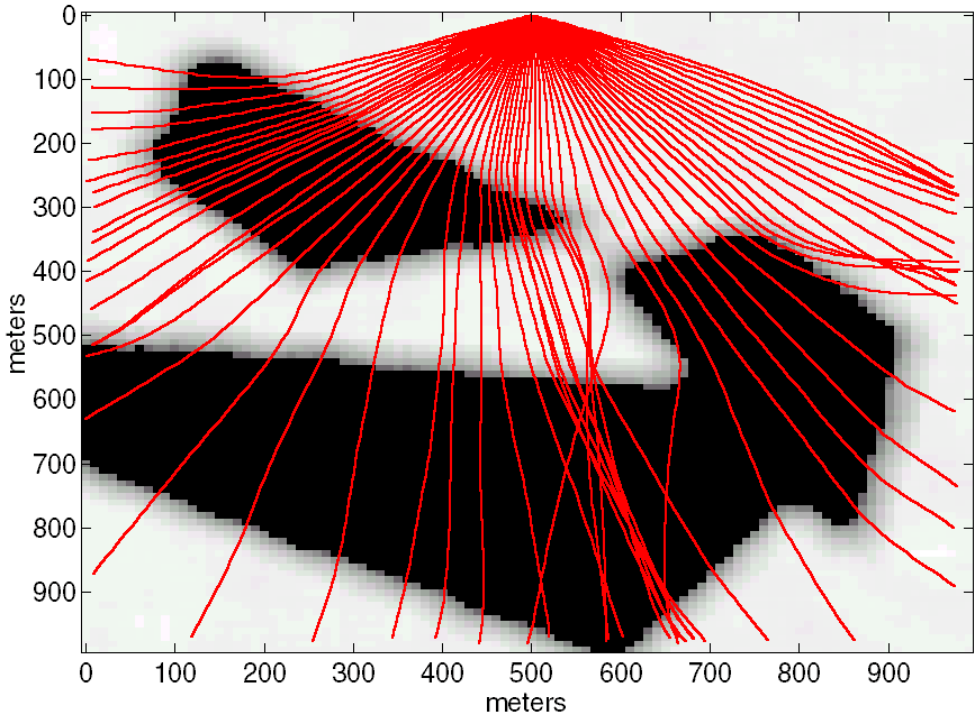

Figure 23. This is the result of raytracing through the velocity model of Figure 19 after applying a 50 m smoother.

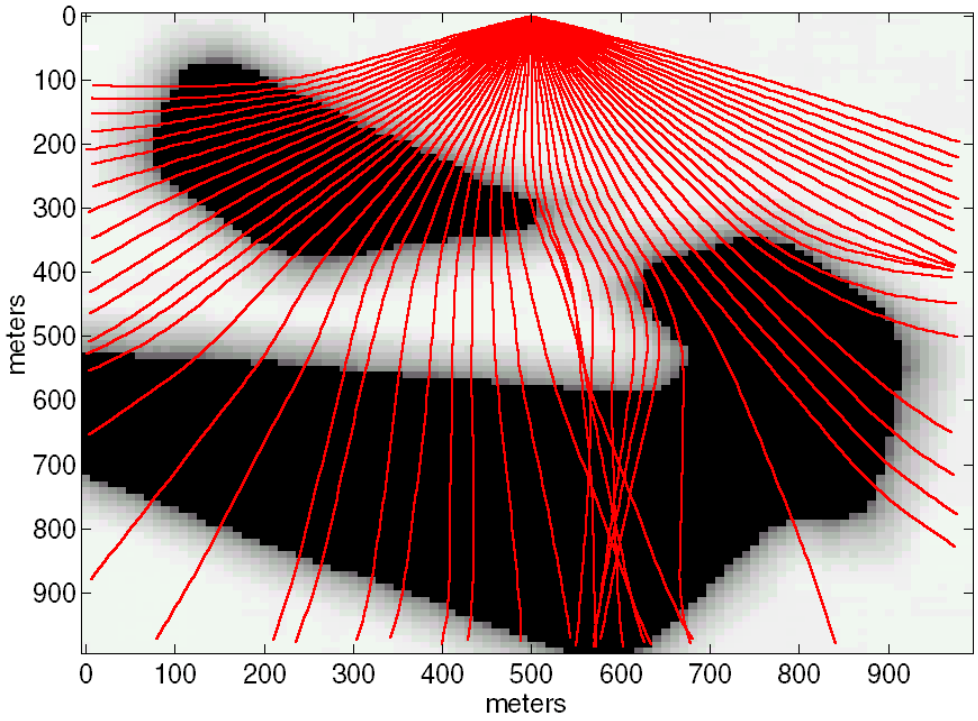

Figure 24. This is the result of raytracing through the velocity model of Figure 19 after applying a 100 m smoother.

THE ACOUSTIC FINITE-DIFFERENCE MODELLING FACILITY

This facility was described in Youzwishen and Margrave (1999) but has been extensively improved. Many bugs were found and corrected and the efficiency of a number of codes was improved. Furthermore, the calling interfaces of the programs was streamlined and made consistent. Therefore a brief description will be given here in the hope that potential users will be encouraged to investigate. (Also, perhaps those who found the previous release too frustrating will try again.)

Technical description

The acoustic finite difference modelling tools are found in the "finitedif" toolbox. The facility is based on the variable-velocity scalar wave equation and is suitable for modelling P-waves in complex media. However, any effect that relies on the generation of shear waves (such as P-wave AVO) will not be properly modelled. The facility is most suitable for studying the effects of imaging (P-waves) in heterogeneous media, lateral resolution, multiple generation, and so on.

The theoretical development of the modelling facility begins with the variable-velocity scalar wave equation in two spatial dimensions

$$\nabla^2\Psi(x,z,t)=\frac{1}{v^2(x,z)}\frac{\partial^2}{\partial t^2}\Psi(x,z,t). \tag{8}$$

In this expression, $\Psi(x,z,t)$ is the pressure wavefield, $v(x,z)$ is the heterogeneous velocity field, and the 2D Laplacian is

$$\nabla^2=\frac{\partial^2}{\partial x^2}+\frac{\partial^2}{\partial z^2}. \tag{9}$$

If the second time derivative in equation (8) is replaced by its second-order finite-difference approximation, the result is

$$\nabla^2\Psi(x,z,t)=\frac{1}{\Delta t^2 v^2(x,z)}\left[\Psi(x,z,t+\Delta t)-2\Psi(x,z,t)+\Psi(x,z,t-\Delta t)\right]. \tag{10}$$

This expression can be solved for the wavefield at t+Δt to give the basic modelling equation

$$\Psi(x,z,t+\Delta t)=\left[2+\Delta t^2 v^2(x,z)\nabla^2\right]\Psi(x,z,t)-\Psi(x,z,t-\Delta t). \tag{11}$$

This is an expression for time stepping the wavefield. It shows that estimation of the wavefield at t+Δt requires knowledge of the two earlier wavefields at t and t-Δt. Each of these wavefields is called a *snapshot* and, in a computer simulation, they are all two-dimensional matrices.

Equation (11) shows that $\Psi(x,z,t+\Delta t)$ is estimated from the superposition of three terms: $2\Psi(x,z,t)$, $\Delta t^2 v^2(x,z)\nabla^2\Psi(x,z,t)$, and $-\Psi(x,z,t-\Delta t)$. The first and

third terms are simply scaled versions of previous wavefields and are easily obtained. However, the second requires the computation of the Laplacian on the current wavefield, which is an expensive operation. Matlab supplies the function *del2* that computes $\nabla^2/4$ of a matrix using centered second-order finite-difference operators that are modified near the boundary. Experimentation showed that *del2* was not optimal for use with absorbing boundary conditions so two alternate Laplacians, *del2_5pt* and *del2_9pt*, were created. Both of these functions pad the entire boundary of the input matrix with extra rows and columns of zeros and this works better with absorbing boundaries. Function *del2_5pt* implements ∇^2 using second-order finite-difference operators while *del2_9pt* uses fourth-order. Thus *del_5pt* computes the approximation

$$\nabla^2\Psi \approx \frac{\Psi(x+\Delta x)-2\Psi(x)+\Psi(x-\Delta x)}{\Delta x^2}+\frac{\Psi(z+\Delta z)-2\Psi(z)+\Psi(z-\Delta z)}{\Delta z^2} \quad (12)$$

while *del2_9pt* calculates

$$\nabla^2\Psi \approx \frac{-\Psi(x+2\Delta x)+16\Psi(x+\Delta x)-30\Psi(x)+16\Psi(x-\Delta x)-\Psi(x-2\Delta x)}{12\Delta x^2}$$
$$+\frac{-\Psi(z+2\Delta z)+16\Psi(z+\Delta z)-30\Psi(z)+16\Psi(z-\Delta z)-\Psi(z-2\Delta z)}{12\Delta z^2}. \quad (13)$$

The fundamental time-stepping function in the finite difference toolbox is *afd_snap*. This function requires two input wavefields representing $\Psi(x,z,t)$ and $\Psi(x,z,t-\Delta t)$ and computes $\Psi(x,z,t+\Delta t)$ according to equation (11). This function is intended to be used in a computation loop that time-increments a wavefield any number of steps. For the first step, two initial wavefields must be constructed to start the simulation and sources may be prescribed by placing appropriate impulses in these two wavefields. Receivers are simulated by extracting samples at each receiver location from each $\Psi(x,z,t)$ as it is computed. These extracted samples may be accumulated in vectors representing the recorded traces. Thus, sources and receivers may be placed anywhere in (x,z) and so any acquisition geometry (VSPs, topography, OBC, etc.) can be simulated.

The use of equation (11) in a time-stepping simulation is known to be unstable in certain circumstances. Instability means that the amplitudes of the wavefield grow without bound as it is stepped through time. The key to this behavior is the amplitude of the $\Delta t^2 v^2(x,z)\nabla^2\Psi(x,z,t)$ term in equation (11). Using the five-point Laplacian of equation (12) (with $\Delta z=\Delta x$) in equation (11) leads to

$$\Psi(x,z,t+\Delta t) = 2\Psi(x,z,t)-\Psi(x,z,t-\Delta t)+\cdots$$
$$\frac{\Delta t^2 v^2(x,z)}{\Delta x^2}\left[\delta_{xx}\Psi(x,z,t)+\delta_{zz}\Psi(x,z,t)\right] \quad (14)$$

where the second central difference operator is

$$\delta_{xx}\Psi(x,z,t) = \Psi(x+\Delta x,z,t) - 2\Psi(x,z,t) + \Psi(x-\Delta x,z,t) \tag{15}$$

and a similar form for δ_{zz}. In equation (14), all of the Ψ terms can be considered to have similar magnitude. Thus, the factor $\Delta t^2 v^2 \Delta x^{-2}$ is a possible amplification factor if it becomes too large. Lines et al. (1999) show that the condition for stability is

$$\frac{v\Delta t}{\Delta x} \le \frac{2}{\sqrt{a}} \tag{16}$$

where the constant "a" is the sum of the absolute values of the weights for the various wavefield terms in the finite-difference approximation for ∇^2. For the Laplacian of equation (12), a=8, while for equation (13), a=128/12=32/3. Also, since v is a function of (x,z) it suffices to use the maximum velocity in the model. Thus the stability conditions are

$$\frac{v_{max}\Delta t}{\Delta x} \le \begin{cases} \frac{1}{\sqrt{2}} & \text{second order Laplacian} \\ \sqrt{\frac{3}{8}} & \text{fourth order Laplacian} \end{cases} . \tag{17}$$

These stability considerations mean that the time and space sample rates should not be chosen independently. Generally, finite-difference operators need many more samples than the Nyquist criterion of two per wavelength. Technically, this is because the operators cause an artificial dispersion called *grid dispersion*. Grid dispersion preferentially affects the shorter wavelengths so oversampling reduces the dispersion. A good rule of thumb is that, for good fidelity, about five samples per wavelength for the 4th order Laplacian and ten samples per wavelength for the 2nd order. Typically, in the creation of a model, a desired temporal frequency range is known. Then, the minimum wavelength is given by $\lambda_{min} = v_{min} / f_{max}$ and the spatial sample rate can be chosen to achieve a desired number of samples-per-wavelength. Finally the temporal sample rate is chosen to achieve stability. (Manning and Margrave (2000) describe a technique that can overcome these stability and dispersion problems.)

Typically, the user will not invoke *afd_snap* directly. Instead, *afd_shotrec* is provided to create a source record and *afd_explode* will create exploding reflector models. Function *afd_shotrec* requires inputs giving: the temporal and spatial sample sizes, the maximum record time, the velocity matrix, the receiver positions, the wavelet, the desired Laplacian (five point or nine point), and the two initial snapshots of the wavefield (snap1 and snap2). The snapshots should be initialized to matrices of zeros the same size as the velocity model. Then the source configuration is described by placing appropriate impulses in these two snapshots. A simple strategy is to leave snap1 as all zeros and simply place impulses in snap2 to form an appropriate source array.

Examples

Figure 25 shows the code required to create the velocity model of Figure 26. This is a simple layered earth that will serve to demonstrate basic finite difference behavior. The model is defined on a 128 by 128 grid with a cell size (dx) of 10 m. Initially the velocity matrix is filled with values of v3 (3200 m/s) corresponding to the third layer (line 5). Then the first and second layers are superimposed using *afd_vmodel*. As described in the discussion of Figure 18, *afd_vmodel* fills in a polygonal region in a velocity matrix with a constant value. Lines 8 and 9 define the four corners of a rectangle that encompasses layer 1 and line 10 calls *afd_vmodel* to install the layer in the velocity matrix. Lines 10 and 11 repeat this process for the second layer. It is an unfortunate behavior of *afd_vmodel* that only the interior and not the boundary of the polygon is filled with the new velocity. For this reason, the polygon is defined to be one-half of a grid cell larger than it might otherwise be.

```
1.  %make a velocity model
2.  nx=128;dx=10;nz=128; %basic geometry
3.  x=(0:nx-1)*dx;z=(0:nz-1)*dx;
4.  v1=2000;v2=2800;v3=3200;%velocities
5.  vmodel=v3*ones(nx,nz);% fill matirx with v3
6.  z1=(nz/8)*dx;z2=(nz/2)*dx;dx2=dx/2;
7.  xpoly=[-dx2 max(x)+dx2 max(x)+dx2 -dx2];
8.  zpoly=[-dx2 -dx2 z1+dx2 z1+dx2];
9.  vmodel=afd_vmodel(dx,vmodel,v1,xpoly,zpoly);%install layer 1
10. zpoly=[z1+dx2 z1+dx2 z2+dx2 z2+dx2];
11. vmodel=afd_vmodel(dx,vmodel,v2,xpoly,zpoly);%install layer 2
```

Figure 25. This code example uses *afd_vmodel* to create the velocity model of Figure 26.

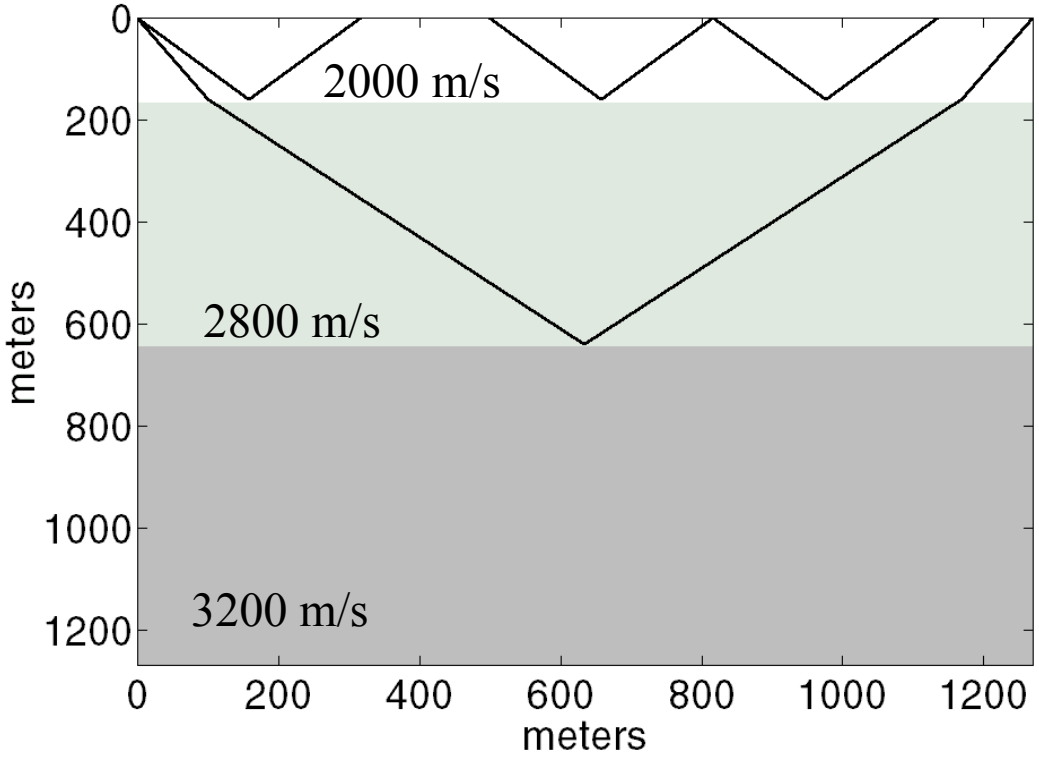

Figure 26. A layered velocity model used to model the shot records in Figures 28-31. The raypaths show correspond to the three events that are raytraced on the Figures. The code in Figure 25 created this model.

Figure 27 shows code that was run following that of Figure 25 to calculate two finite difference shot records. Line 1 defines the time step to be used in the simulation (Δt of equation (11)) while line 2 defines the time-sample rate and maximum time of the final seismogram. In many finite-difference codes, these two time-intervals (dtstep and dt) are not distinguished and the seismograms are sampled at the same rate as the time stepping. This is a very inefficient use of memory because a finite-difference simulation must be very strongly oversampled in time (relative to the Nyquist criterion) to compensate for the poor spectral performance of the finite-difference derivative. A reasonable rule-of-thumb is that frequencies are only good out to (at most) 25% of the Nyquist frequency defined by dtstep. Thus dt is set to four times dtstep. (Internal to *afd_shotrec*, the seismogram is calculated at a sample rate of dtstep and then resampled, with a zero-phase anti-alias filter, to the rate dt.)

Lines 5-7 define the first two snapshots of the simulation and therefore determine the source strength and location. Line 5 sets $\Psi(x, z, t = -\Delta t)$ to all zeros while lines 6-7 set $\Psi(x, z, t = 0)$ to all zeros except for a unit impulse at ($x=x_{max}/2$, $z=0$). Lines 9-10 invoke *afd_shotrec* to create synthetic with a second-order Laplacian and lines 12-13 create a similar synthetic but with a fourth-order Laplacian. (The last input argument in *afd_shotrec* is a flag denoting the desired Laplacian approximation.) There are actually two seismograms returned from *afd_shotrec*. The first is the filtered seismogram and the second is unfiltered. The filter specification, in Ormsby

parameters, is the third-to-last argument and is [5 10 30 40] in this case. That is, the filter passband begins at 5 Hz., reaches full pass at 10 Hz., continues at full pass until 30 Hz., and ends at 40 Hz. The second-last parameter specifies that the filter is to be zero-phase. (Though Ormsby parameters are used to specify the filter, it is actually implemented with Gaussian slopes in the frequency domain and is far more effective than a typical Ormsby filter.)

```
1.  dtstep=.001;%time step
2.  dt=.004;tmax=1;%time sample rate and max time
3.  xrec=x;%receiver locations
4.  zrec=zeros(size(xrec));%receivers at zero depth
5.  snap1=zeros(size(vmodel));
6.  snap2=snap1;
7.  snap2(1,length(x)/2)=1;%place the source
8.  %second order laplacian
9.  [seismogram2,seis2,t]=afd_shotrec(dx,dtstep,dt,tmax, ...
10.                         vmodel,snap1,snap2,xrec,zrec,[5 10 30 40],0,1);
11. %fourth order laplacian
12. [seismogram4,seis4,t]=afd_shotrec(dx,dtstep,dt,tmax, ...
13.                         vmodel,snap1,snap2,xrec,zrec,[5 10 30 40],0,2);
```

Figure 27. A coding example that creates the shot records of Figures 28 and 29. The source is established as an impulse at $x=x_{max}/2$ and $z=0$ on line 7. The data shown in Figure 28 is created by lines 9-10 while that of Figure 29 is done by lines 12-13. The last argument in *afd_shotrec* is a flag denoting which Laplacian to use. Figures 30 and 31 were created with slight modifications to this code.

Figures 28 and 29 show the 2nd order and 4th order seismograms created with the code of Figure 27. Posted on top of the right half of each picture are three curves giving the raytraced traveltimes corresponding to the three raypaths in Figure 26. From top down, these are: the reflection off of the first layer, the first multiple in the first layer, and the reflection off of the second layer. It is apparent that the corresponding events in the seismograms lag behind these traveltimes though the 4th order solution is better than the 2nd order. This is a manifestation of the phenomenon known as grid dispersion. This refers to the fact that the actual speed of wave propagation on a finite grid is a function of wavelength. The shorter the wavelength, the slower the propagation is. The effect is a function of the spectral performance of the spatial derivatives and so is lessened with the 4th order Laplacian.

Also apparent in these figures are a pair of quasi-linear events that appear to originate from the boundaries of the model. These are edge-effect artifacts. The AFD facility incorporates absorbing boundaries based on the method of Clayton and Engquist (1977) but these are not perfectly effective. Examination of Figures 28-31 shows that the absorbing boundaries are apparently less effective for the 4th order solution than for the 2nd. This is likely a program bug and is under investigation.

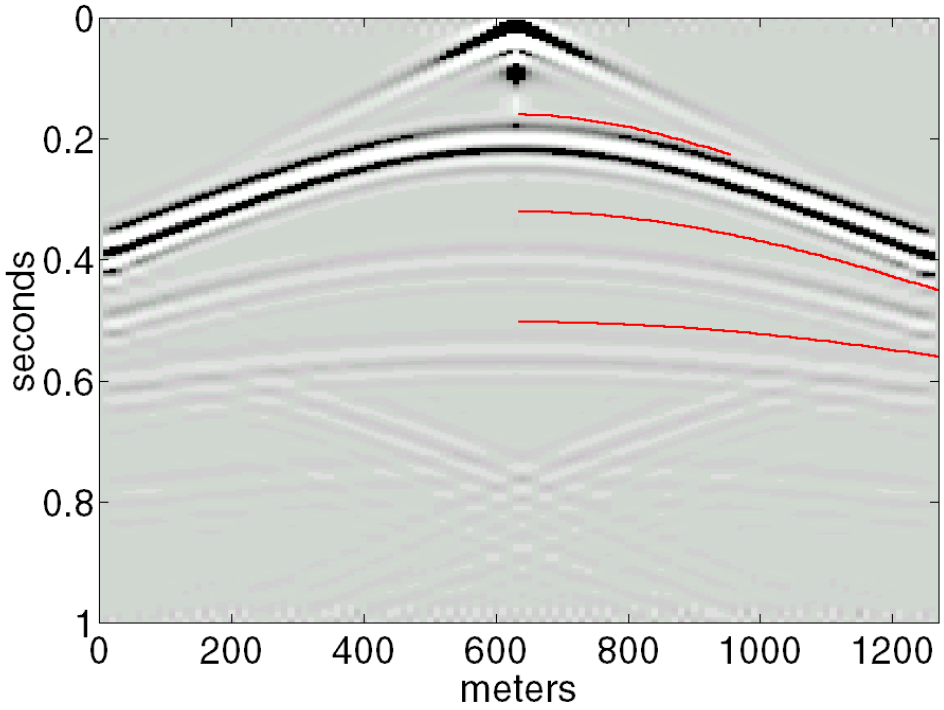

Figure 28. A shot record created by the code of Figure 27 using a second order Laplacian approximation. The grid size was 10 m and the time-step size was .001 seconds. The three lines on the right side of the record are raytraced traveltimes for the three rays of Figure 27.

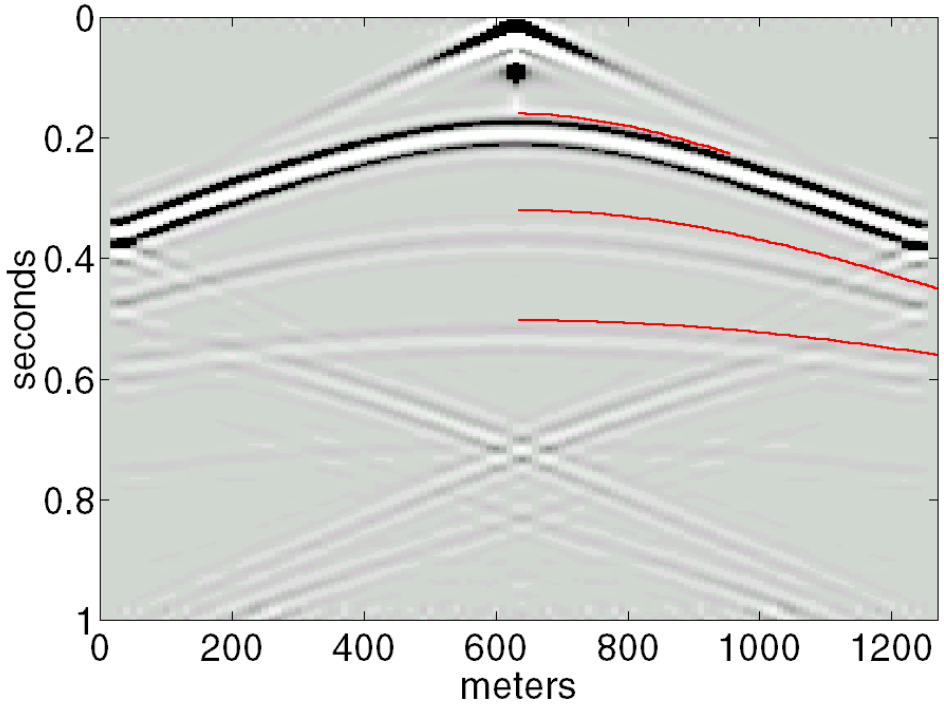

Figure 29. A shot record created by the code of Figure 27 using a fourth order Laplacian approximation. The grid size was 10 m and the time-step size was .001 seconds. The three lines on the right side of the record are raytraced traveltimes for the three rays of Figure 27.

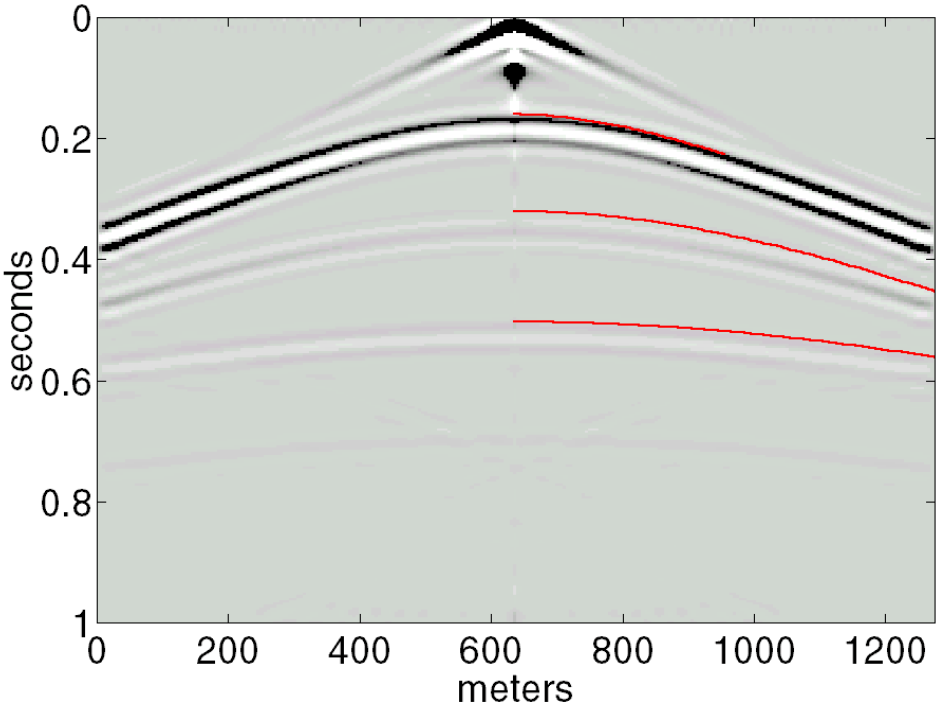

Figure 30. Similar to Figure 28 in all respects except that the grid size was 5 m.

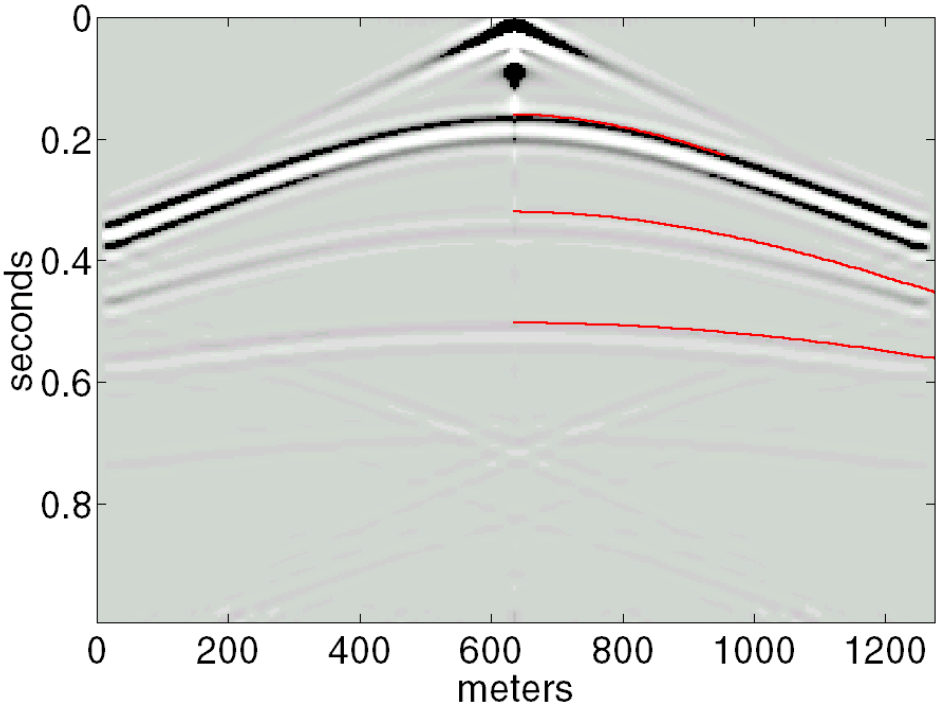

Figure 31. Similar to Figure 29 except that the grid size was 5 m and the time-step size was .0009 sec.

Figures 30 and 31 are more accurate seismograms than Figures 28 and 29. They were created with slight modifications to the code of Figures 25 and 27. In Figure 30, the 2nd order Laplacian was again used but the spatial grid size was reduced to 5 m.

(To obtain the same offsets, the grid dimensions were doubled to 256 by 256.) This solution shows very little grid dispersion and almost no boundary artifacts. With a minimum velocity of 2000 m/s and a maximum frequency of 40 Hz., the shortest wavelengths in this simulation are about 50 m or almost ten times the grid size. Figure 32 compares near-offset traces from all four seismograms. The trend towards less grid dispersion with a more accurate simulation is apparent as a gradual movement of events to earlier times. More subtlely, grid dispersion appears also to affect the event phase.

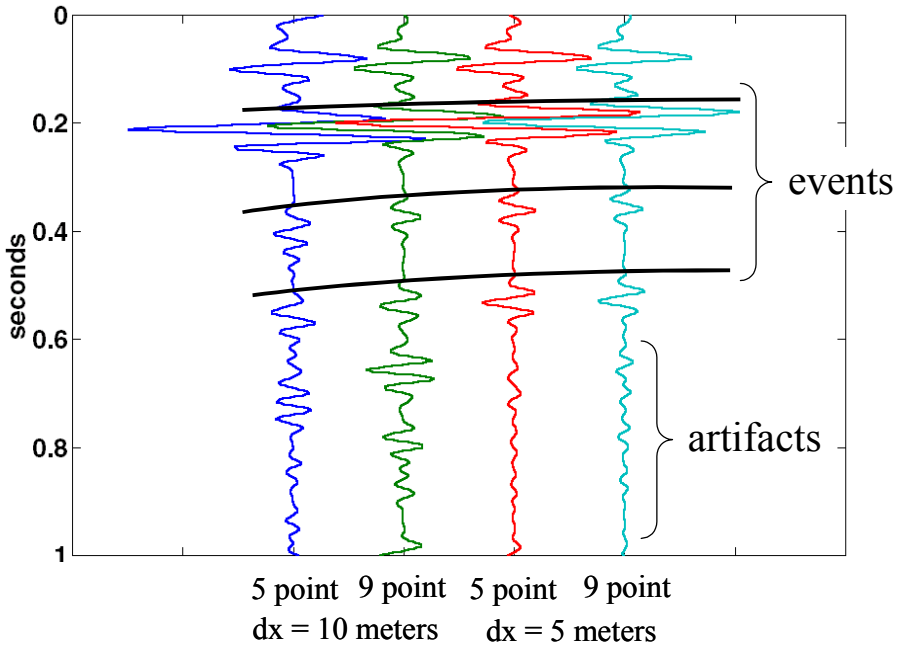

Figure 32. Here is a comparison of a near-offset trace from each of Figures 28-31. The physical accuracy of the simulation increases from left to right. Note the progression of events towards earlier times that is also from left to right. This is caused by grid dispersion. The "artifacts" are mostly boundary reflections that are not fully suppressed by the "absorbing" boundaries.

In addition to modelling shot records, the AFD package can also simulate stacked sections through the exploding reflector model. The function *afd_explode* provides this function. Unlike *afd_shotrec*, it does not require the specification of two initial snapshots. Instead, these are generated from the velocity model itself. Matlab's numerical gradient function is used to calculate $\Psi(x,z,t=0)$ directly from the velocity model while $\Psi(x,z,t=-\Delta t)$ is set to zero. Specifically

$$\Psi(x,z,t=0)=\left|\vec{\nabla}\ln v(x,z)\right| \tag{18}$$

which effectively sets the reflectivity to be $\delta v / v$ taken in the direction normal to the local "reflector".

Figure 33 is a code example that builds a velocity model representing a small channel beneath a layered medium. Function *afd_vmodel* is used to build a four-layered model in similar fashion to the previous example. Then, on lines 14-17, a small channel is defined as a rectangle 20 m wide and 50 m deep and installed at the top of the fourth layer using *afd_vmodel*. On line 18, *plotimage* is used to display the velocity model with the result being Figure 34. Function *plotimage* is designed to display seismic data that is always nearly zero mean. A velocity matrix, consisting of all positive numbers, is decidedly not zero mean and will generally display as solid black. Hence, the mean value of the velocity matrix is subtracted as it is passed to *plotimage* for display.

```
1.  dx=10;xmax=2500;zmax=1000;%grid size, max line length, max depth
2.  x=0:dx:xmax;z=0:dx:zmax; % x and z coordinate vector
3.  vhigh=4000;vlow=2000;vrange=vhigh-vlow; % high and low velocities
4.  vel=vlow*ones(length(z),length(x));%initialize velocity matrix
5.  z1=100;z2=200;v1=vlow+vrange/5;%first layer
6.  xpoly=[-dx xmax+dx xmax+dx -dx];zpoly=[z1 z1 z2 z2];
7.  vel=afd_vmodel(dx,vel,v1,xpoly,zpoly);%install first layer
8.  z3=271;v2=vlow+2*vrange/5;zpoly=[z2 z2 z3 z3];%second layer
9.  vel=afd_vmodel(dx,vel,v2,xpoly,zpoly);%install second layer
10. z4=398;v3=vlow+pi*vrange/5;zpoly=[z3 z3 z4 z4];%third layer
11. vel=afd_vmodel(dx,vel,v3,xpoly,zpoly);%install third layer
12. zpoly=[z4 z4 zmax+dx zmax+dx];%last layer
13. vel=afd_vmodel(dx,vel,vhigh,xpoly,zpoly);%install last layer
14. width=20;thk=50;vch=vlow+vrange/6;%channel
15. xpoly=[xmax/2-width/2 xmax/2+width/2 xmax/2+width/2 xmax/2-width/2];
16. zpoly=[z4 z4 z4+thk z4+thk];
17. vel=afd_vmodel(dx,vel,vch,xpoly,zpoly);%install channel
18. plotimage(vel-.5*(vhigh+vlow),z,x);%plot the velocity model
```

Figure 33. This code creates a velocity model representing a small channel beneath a layered medium. There are three horizontal layers above the channel plus a fourth that the channel is embedded in. The layers are installed in the velocity matrix using *afd_vmodel*. The channel is defined as a small rectangle, 20 m wide and 50 m deep (line 14) and installed at the top of the fourth layer with *afd_vmodel* (line 17). The *plotimage* command (line 18) creates Figure 34.

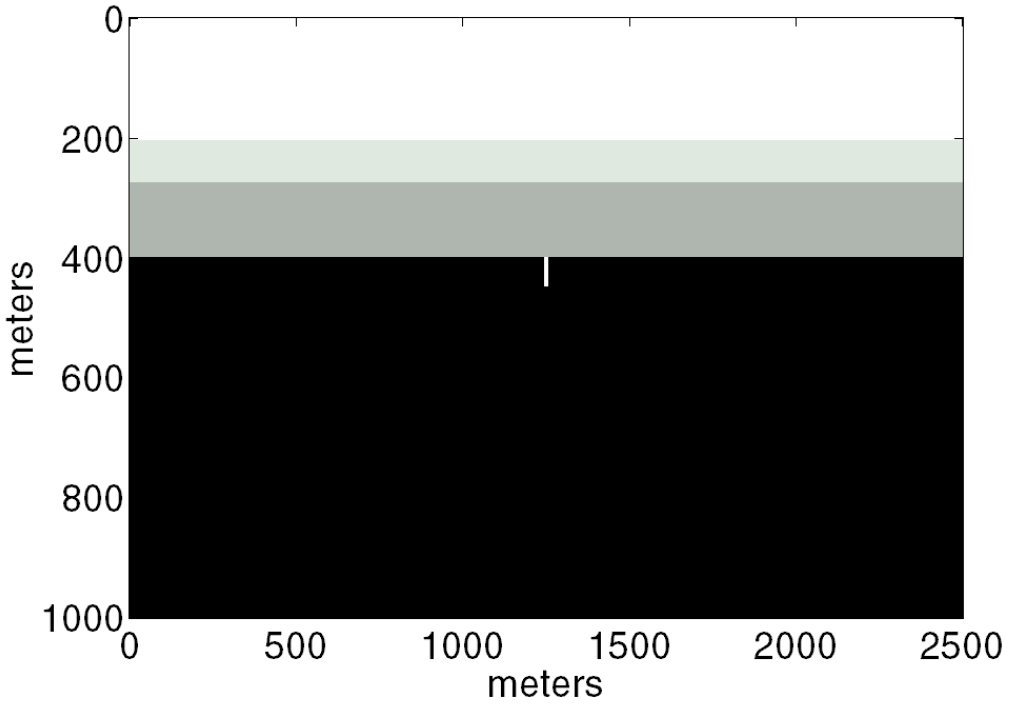

Figure 34. A velocity model showing a buried channel. The darker the shading, the higher the velocity. This was created by the code in Figure 33.

Figure 35 shows a code sample that is to be run after that of Figure 33 to create an exploding reflector seismogram ("explodogram"). In similar fashion to *afd_shotrec*, both a time-step size and an output temporal sample rate are prescribed as .001 and .004 seconds respectively. On lines 5-6 the explodogram is created with a 2nd order Laplacian and a [10 15 40 50] bandpass filter is applied. After plotting the filtered explodogram with *plotimage*, lines 10-11 compute the vertical traveltime to the top and bottom of the channel and lines 13-14 plot picks at the appropriate points on top of the seismic data display. The picks are plotted as horizontal that are only 20m wide so they are difficult to discern. Look for two short horizontal marks, one above the other, at slightly greater than .3 seconds.

```
1.  %do a finite-difference exploding reflector model
2.  dt=.004; %temporal sample rate
3.  dtstep=.001; %modelling step size
4.  tmax=2*zmax/vlow; %maximum time
5.  [seisfilt,seis,t]=afd_explode(dx,dtstep,-dt,tmax, ...
6.                  vel,x,zeros(size(x)),[10 15 40 50],0,1);
7.  %plot the seismogram
8.  plotimage(seisfilt,t,x)
9.  %compute times to top and bottom of channel
10. tchtop=2*(z1/vlow + (z2-z1)/v1 + (z3-z2)/v2 + (z4-z3)/v3);
11. tchbot=tchtop+2*(thk/vch);
12. %annotate times
13. h1=drawpick(xmax/2,tchtop,0,width);
14. h2=drawpick(xmax/2,tchbot,0,width);
```

Figure 35. A sample code that creates an exploding reflector model from the channel section in Figure 34. The model is created by *afd_explode* on lines 5-6 using a second order Laplacian with 10 m spatial sampling. (The latter is set in the code of Figure 33). The data is plotted on line 8 and lines 10-14 compute and annotate the times to the top and bottom of the channel.

Computational artifacts, especially grid dispersion effects, dominate the explodogram of Figure 36. A user unfamiliar with these might assume that the response is correct and somehow dominated by reverberations. Such an assumption is dangerous when working with synthetic seismogram codes. The wise choice is to rerun the computation with the parameters incremented towards greater realism. If a significant change is observed, this procedure should be continued until an iteration with no change occurs. Accordingly, the code of Figure 35 was rerun using a 4^{th} order Laplacian approximation. (This amounts to changing the last parameter in *afd_explode* from 1 to 2.) The result, in Figure 37, is dramatically different (and better) than that in Figure 36. Continuing in this fashion, Figure 38 again used the 4^{th} order Laplacian but with a grid spacing of 5 m while Figure 39 was created with the 2^{nd} order Laplacian and a grid size of 2.5 m. These final two results are quite similar so it seems reasonable (though not certain) to assume that either is an accurate simulation. Though the 4^{th} order Laplacian always takes longer to compute than the 2^{nd} order for the same grid size, the 4^{th} order at 5m completes in roughly 1/3 the time of the 2^{nd} order at 2.5 m.

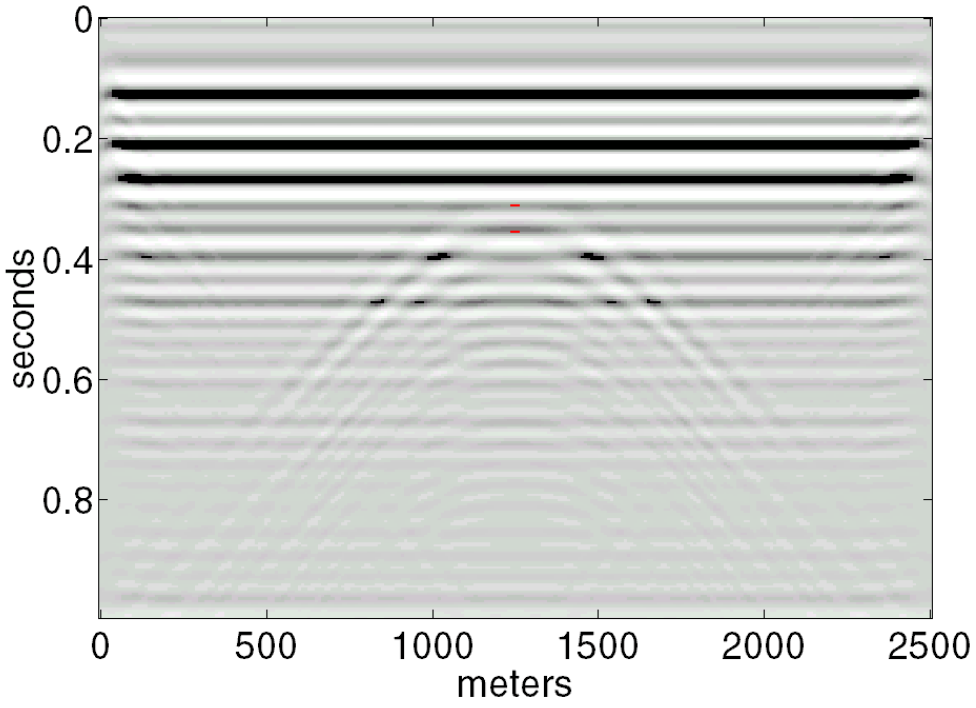

Figure 36. The explodogram (exploding reflector seismogram) created by the code of Figure 35 for the channel model of Figure 34. The Laplacian was 2nd order, the grid spacing was 10 m and the time step was .001 s. Though there is some hint of the channel response, the image is dominated by grid dispersion effects.

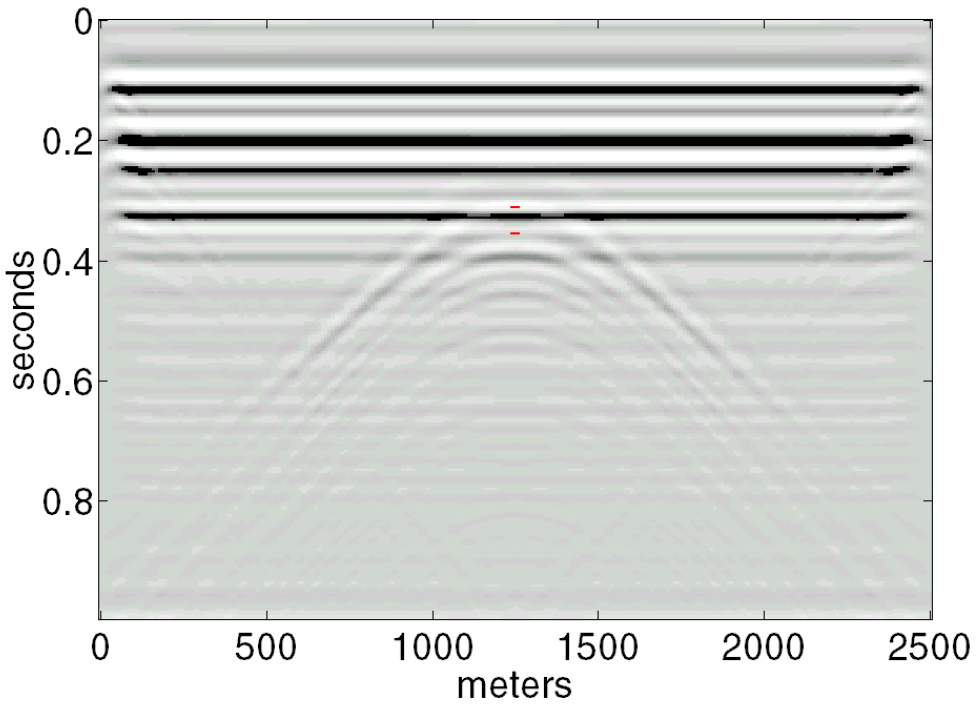

Figure 37. Similar to Figure 36 except that the Laplacian was 4th. A dramatic reduction in grid dispersion effects is evident.

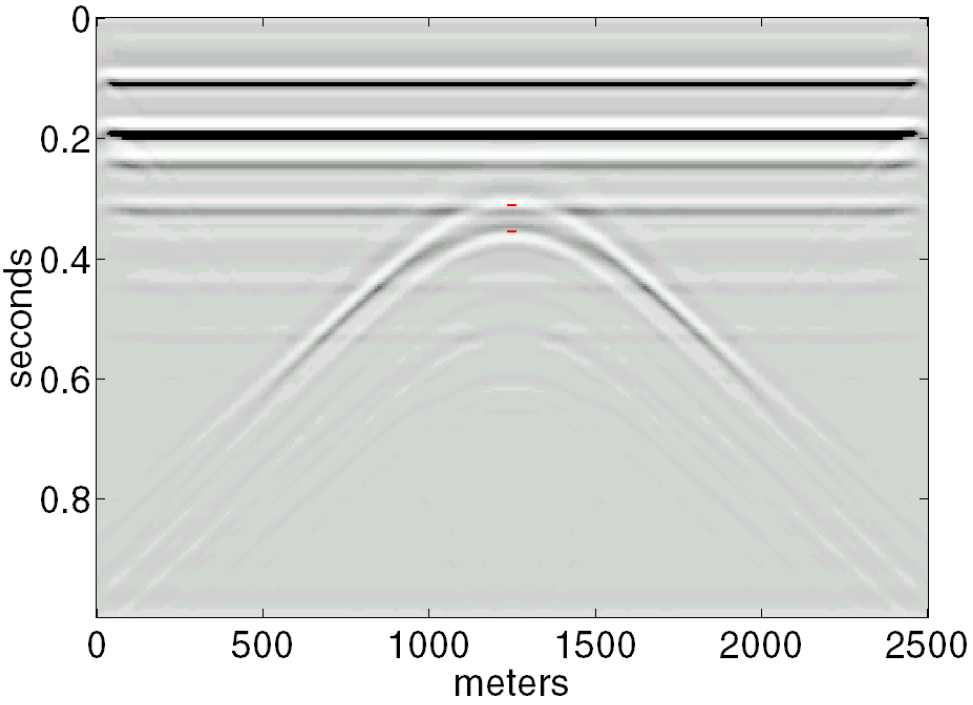

Figure 38. Similar to Figure 36 except that the Laplacian was 4th order and the grid spacing was 5 m.

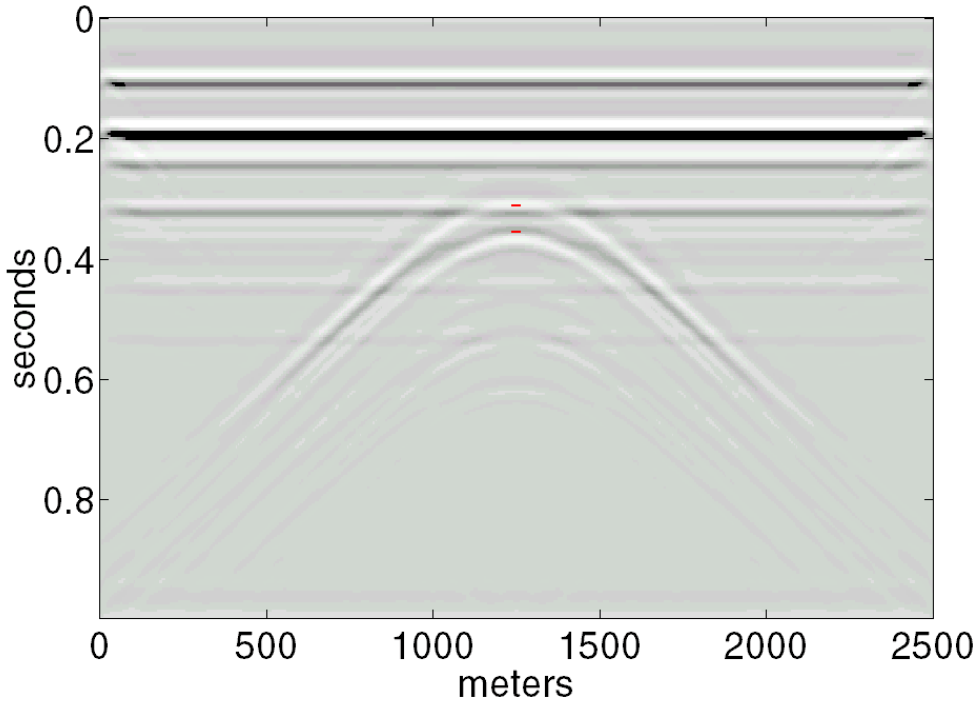

Figure 39. Similar to Figure 36 except that the grid spacing was 2.5 m, the time-step size was .0008 s, while the Laplacian was still 2nd order.

Figure 40 shows the code required to build the velocity model of Figure 41. This model has an anticline beneath a high-velocity wedge and is an abstracted but

essential model of an exploration target beneath a thrust sheet. Figure 42 shows the reflectivity of Figure 41 created by *afd_reflect* that implements equation (18). This is the exploding reflector wavefield at time t=0. Figure 43 shows the code required to create an explodogram while Figure 44 shows such a result using a 2^{nd} order Laplacian and a 10 m grid. The response of the anticline is clearly dispersive so better results are shown in Figures 45 and 46. Figure 46 shows the anticline as a nearly impulsive wavefront with a multiple training behind it.

```
1.  dx=5;xmax=2500;zmax=1000; % grid size, max line length, max depth
2.  xpinch=1500; zwedge=zmax/2;% wedge parameters
3.  x=0:dx:xmax; z=0:dx:zmax;% x&z coordinate vector
4.  vhigh=4000;vlow=2000; % high and low velocities
5.  vel=vlow*ones(length(z),length(x));%initialize velocity matrix
6.  dx2=dx/2;xpoly=[-dx2 xpinch -dx2];zpoly=[-1 -1 zwedge];% wedge
7.  vel=afd_vmodel(dx,vel,vhigh,xpoly,zpoly);% install the wedge
8.  x0=xpinch/2;z0=zwedge+100; % x and z of the crest of the anticline
9.  a=.0005; % a parameter that determines the steepness of the flanks
10. za=a*(x-x0).^2+z0; % model the anticline as a parabola
11. % build a polygon that models the anticline
12. ind=near(za,zmax+dx);xpoly=[x(1:ind) 0 ];zpoly=[za(1:ind) za(ind)];
13. vel=afd_vmodel(dx,vel,vhigh,xpoly,zpoly);%install the anticline
14. xpoly=[0 xmax xmax 0];zpoly=[.9*zmax .9*zmax zmax+dx zmax+dx];
15. vel=afd_vmodel(dx,vel,vhigh,xpoly,zpoly);% install bottom layer
```

Figure 40. This code creates the velocity model of Figure 41. The anticline is modelled as a parabolic shape.

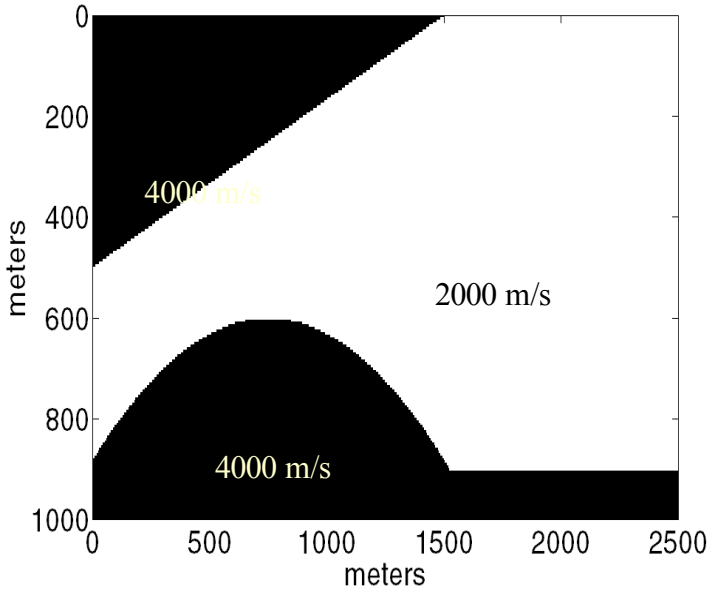

Figure 41. The velocity model created by the code example of Figure 40.

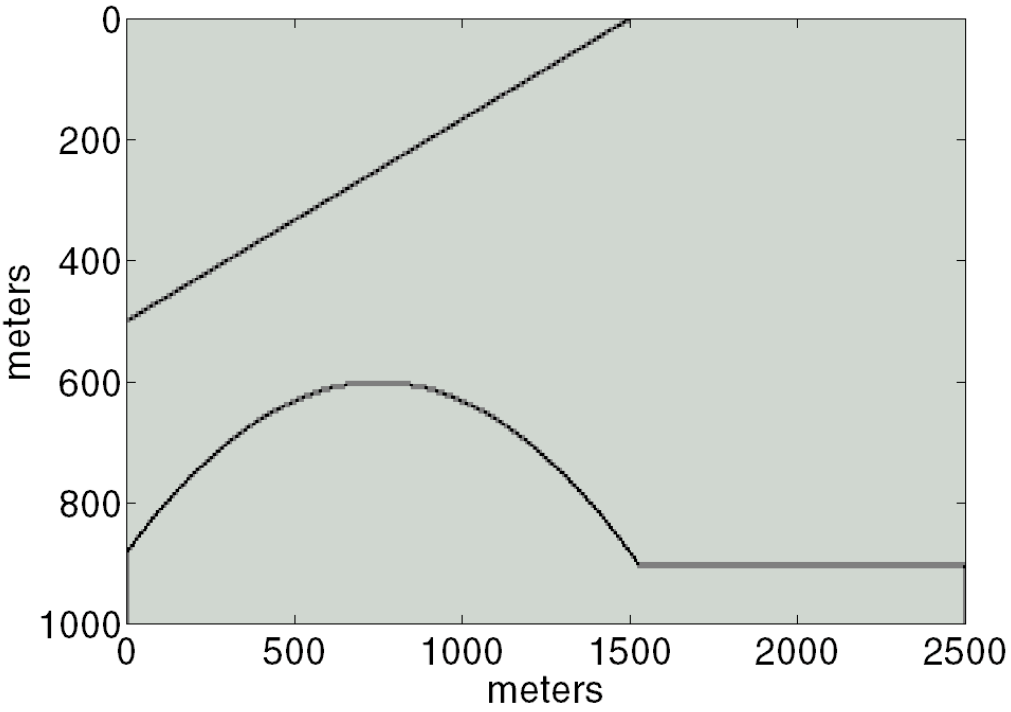

Figure 42. The reflectivity computed from the model of Figure 41 using the definition of equation 18. This is the exploding reflector wavefield at time t=0.

```
1.  %do a finite-difference model
2.  dt=.004; %temporal sample rate
3.  dtstep=.001;
4.  tmax=2*zmax/vlow; %maximum time
5.  [seisfilt,seis,t]=afd_explode(dx,dtstep,dt,tmax, ...
6.                  vel,x,zeros(size(x)),[5 10 40 50],0,2);
```

Figure 43. This code example creates an explodogram from the velocity model of Figure 41. Sample results are in Figures 44-46.

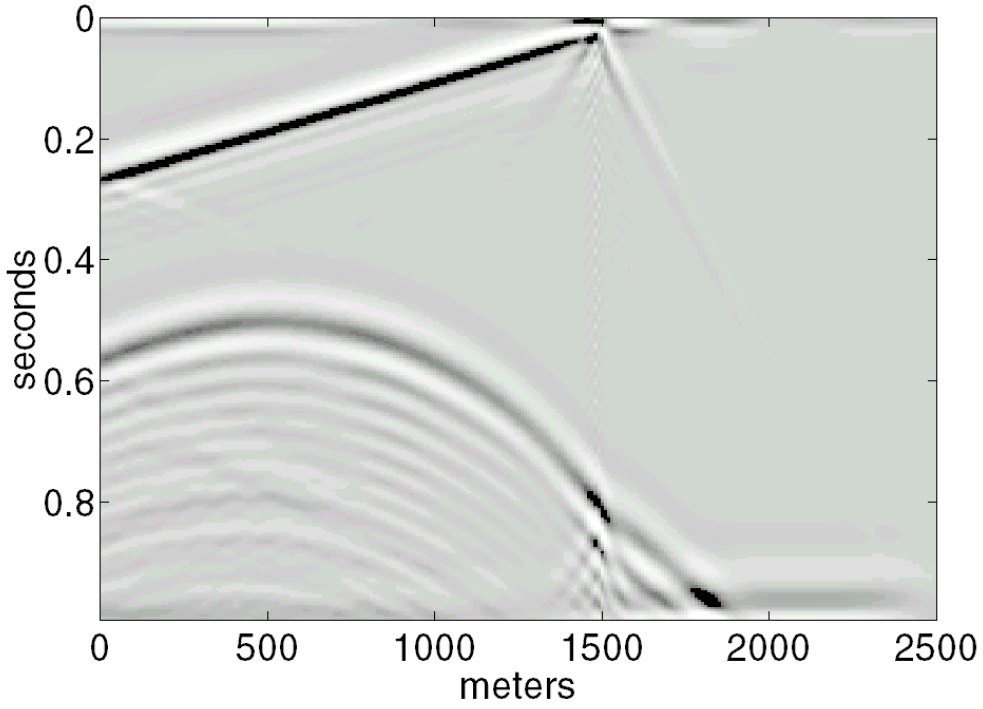

Figure 44. An explodogram created from Figure 41 using the code of Figure 43. The Laplacian was 2nd order, the grid spacing 10 m, and the time-step .001 seconds.

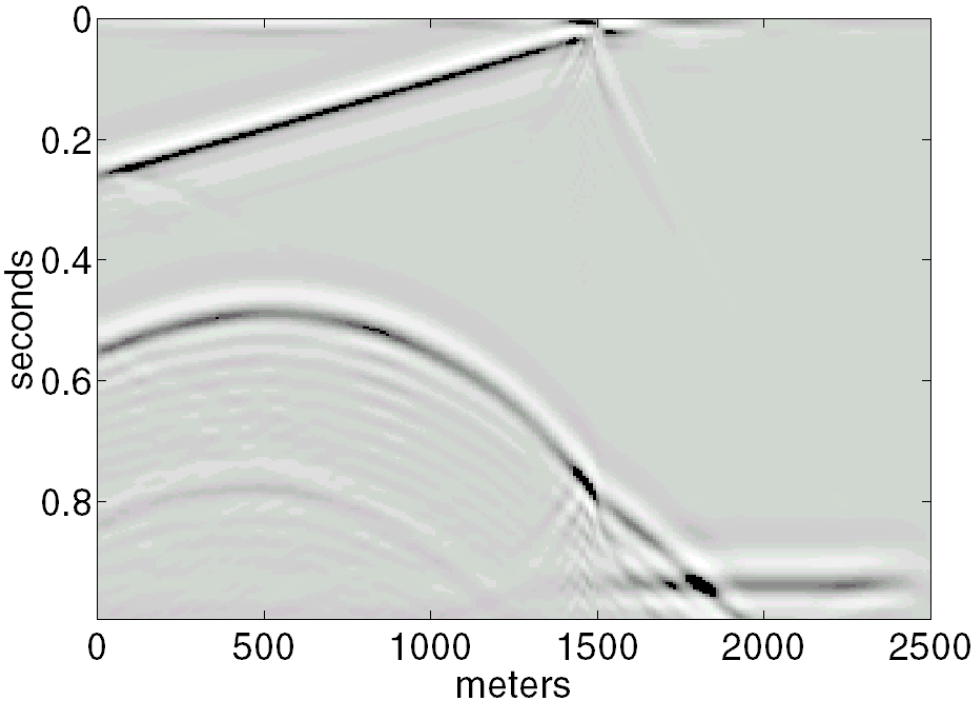

Figure 45. Similar to Figure 44 except that the 4th order Laplacian was used.

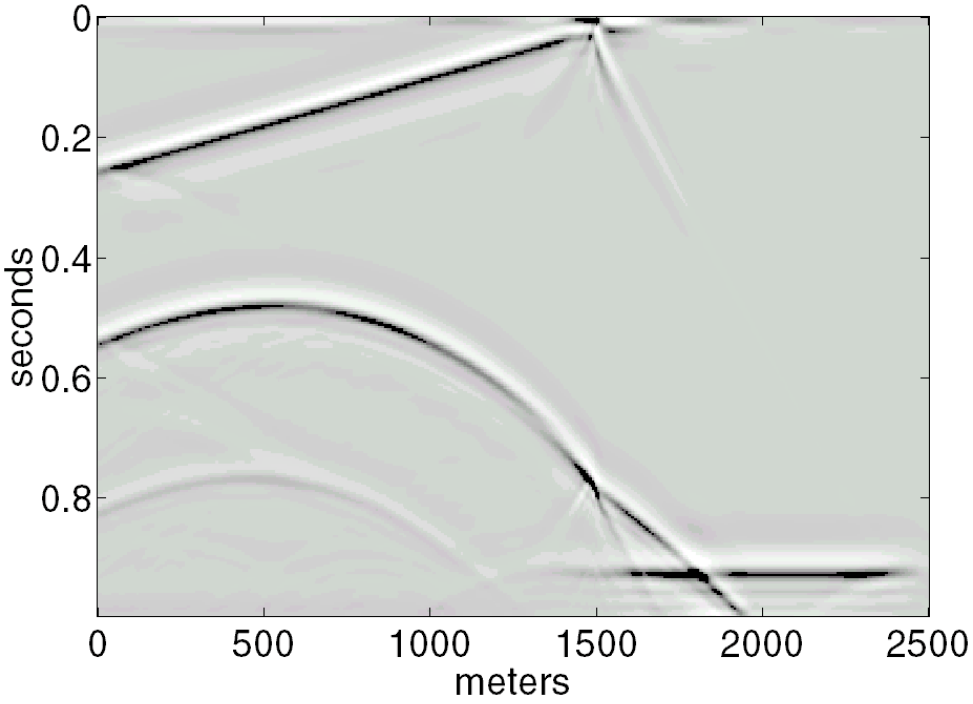

Figure 46. Similar to 44 except that the 4th order Laplacian and a 5 m grid were used.

NORMAL INCIDENCE MODELLING AND MIGRATION

Also included in the raytracing toolbox is a set of functions designed for normal-incidence raytracing. Used in the forward sense, this allows modelling of normal-incidence reflections and in the inverse sense it becomes normal-incidence raytrace migration. The v(x,z) raytrace facility is used for the raytracing since the normal-incidence approach does not require two-point raytracing.

As an example of these tools, consider making a series of "picks" on an event of Figure 46 and then migrating these picks. Let a pick be defined as a triplet of values $(x_0,t_0,dt/dx)$ defined by drawing a small straight-line segment on Figure 46. The slope of the line segment defines the horizontal slowness, dt/dx, and this defines the emergence angle, θ_0, of the normal-incidence ray through the relation

$$p_n = \frac{\sin\theta_0}{v_0} = \frac{1}{2}\frac{dt}{dx} \tag{19}$$

where p_n is the ray parameter of the normal-incidence ray. The center of the line segment defines the point (x_0,t_0). Thus, given a pick and a velocity model, the normal -incidence raypath is determined and can be traced until the measured traveltime, t_0, is twice the traveltime along the raypath. This task is accomplished by the function *normraymig* that is part of the raytrace toolbox. However, using *normraymig* for a large set of picks can be tedious so a higher-level facility is also provided.

Function *plotimage* has a rudimentary picking facility incorporated into it. Initially, the *plotimage* window contains a series of user interface controls along its

bottom edge. In the lower left corner is a popup menu that has three settings: Zoom, Pick(N) and Pick(O). The zoom setting means that mouse actions are interpreted as zooming commands. A click and drag defines a zoom-box and causes a zoom while a single click causes an unzoom. With the other two settings, mouse actions are interpreted as picking signals. A click and drag defines a pick while a single click deletes the most recent pick. The picks are stored in the global variable PICKS as a list of the end points of the line segment defining the pick. Any function that also declares this global variable can then access the picks. The Pick(N) selection signifies that a New pickset is to be defined and the global PICKS is therefore cleared. Alternatively the Pick(O) selection means that an Old pickset is to be augmented so the global PICKS is not cleared. Though many *plotimage* windows can be active simultaneously, they all share the same PICKS global buffer. Therefore, it is incumbent upon the user to keep track of the picks from separate windows if desired. Prior to initiating picking in a second window, the PICKS buffer containing picks from the first window can be copied into another variable.

The function *eventraymig* is designed to work with the *plotimage* picking mechanism and to migrate whatever picks it finds there. Prior to running *eventraymig* the appropriate velocity model must be installed as a global variable by running *rayvelmod* as described previously in the section on v(x,z) raytracing (see Figure 20). Figure 47 shows the data of Figure 46 after a series of picks have been made on the events from the anticline and the basement reflector. The simple command *eventraymod(figno)* (where *figno* denotes the figure number of a depth section that the normal raypaths are to be plotted in) will then migrate these picks. Such a result is shown in Figure 48 where the raypaths have been plotted on top of the reflectivity section of Figure 42. In Figure 48, each raypath emerges from z=0 at an x coordinate defined by one of the picks in Figure 47. It then continues down into the velocity model, obeying Snell's law at each velocity contrast, until the traveltime of the pick equals twice that of the normal raypath. The reflector is then inferred to be at right angles to the raypath and this is denoted in Figure 48 by a perpendicular drawn at the end of the ray. (Note that there is a vertical exaggeration in the scale of this plot so that angles are distorted.)

A similar facility exists for normal raytrace modelling. That is, the picks can be made on the depth section and their positions on the explodogram determined and annotated. In Figure 49, the anticline has been picked on the reflectivity section. The raypaths were drawn by function *eventraymod* that is essentially the inverse on *eventraymig*. (However, *eventraymod* requires two figure numbers as input that specify both the time and depth figures.) If *rayvelmod* has not already been run, it must be executed before the rays can be traced. (Function *rayvelmod* need only be run once in any given Matlab session provided that the velocity model has not changed.) One the raypaths have been determined on the depth section, the emergence points and angles of the rays are known so that $(x_0,t_0,dt/dx)$ picks can be posted on the time domain data. This has been done in Figure 50.

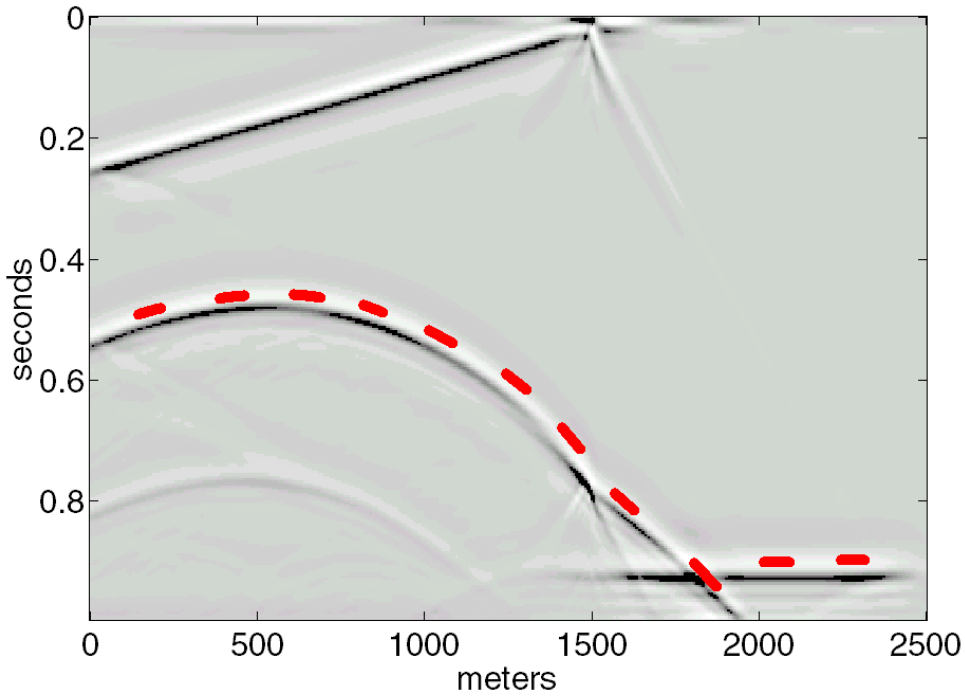

Figure 47. A repeat of Figure 46 but with a series of picks shown posted on to of the event defining the anticline and the bottom reflector.

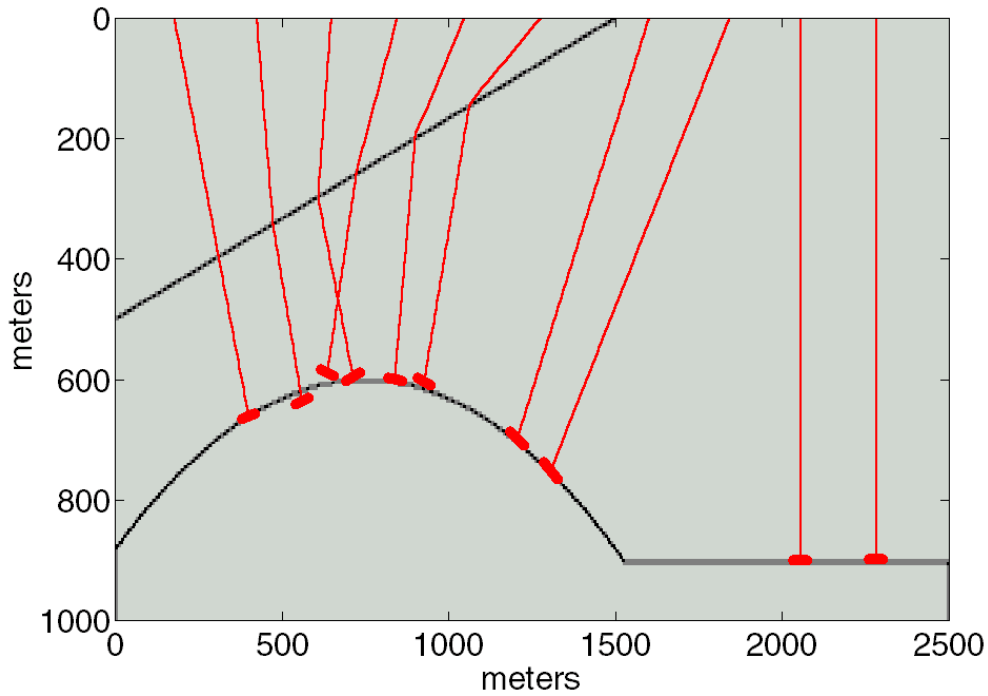

Figure 48. The picks in Figure 47 have been migrated by *eventraymig* and plotted on to of the reflectivity section of Figure 42. Some picks have migrated to positions on the reflectors while others have not. Errors in picking and strong velocity contrasts cause significant errors in migrated position.

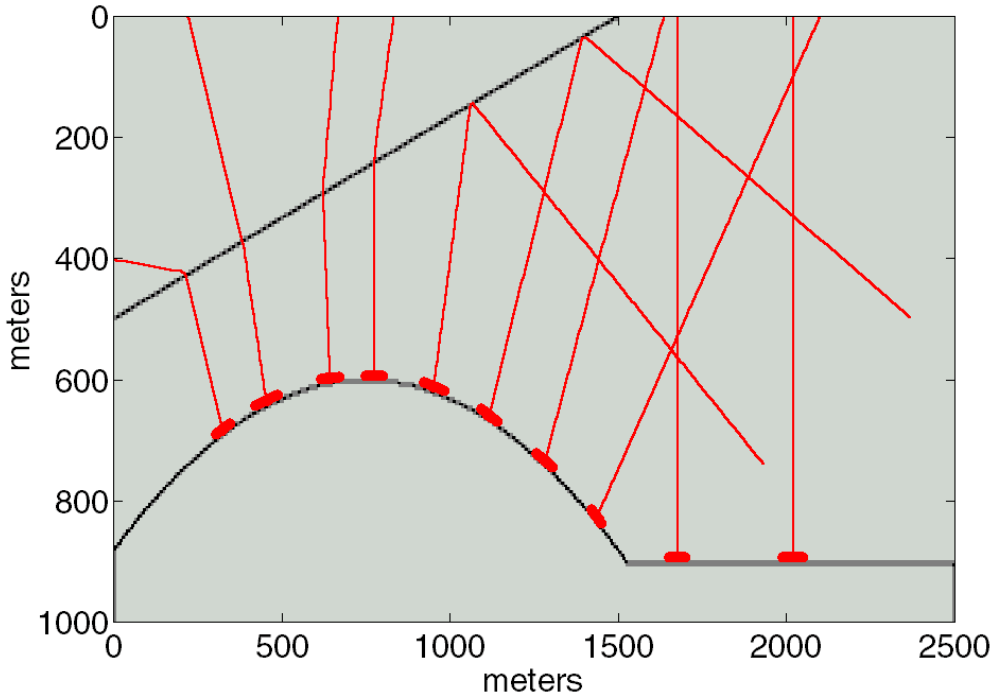

Figure 49. Picks have been made on the depth section of Figure 42 and then normal rays were computed and plotted by *eventraymod*. The results of the calculation are plotted in Figure 50.

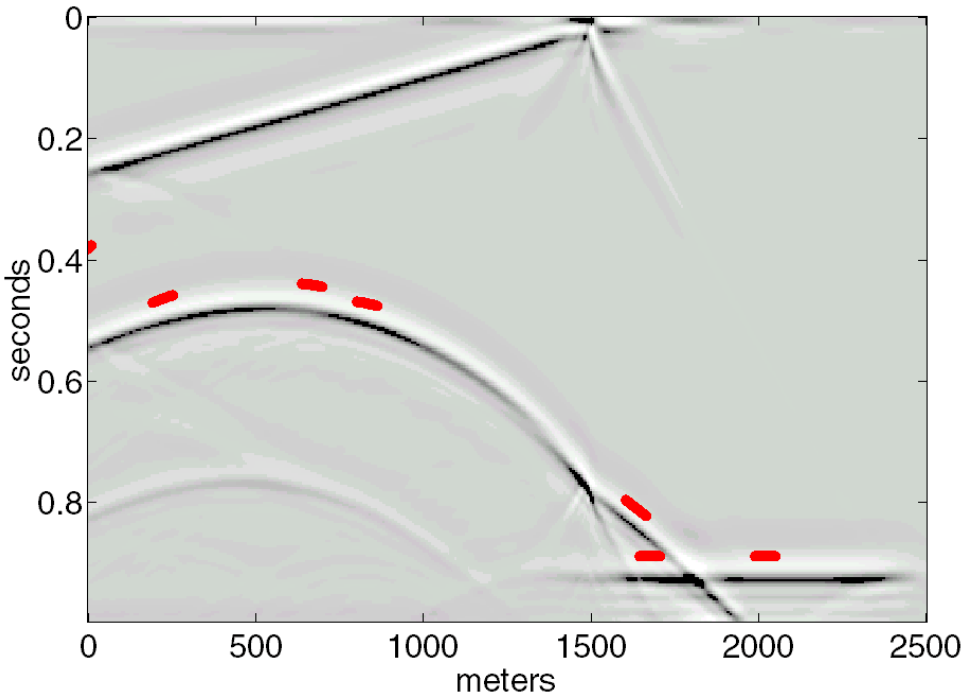

Figure 50. The normal rays of Figure 49 define picks (x_0,t_0,dt/dx) that are posted on the seismic data of Figure 46.

CONCLUSIONS

The seismic modelling capabilities of the CREWES Matlab software have been considerably expanded.

The v(z) raytracing facility provides a very general system for the determination of traveltimes, offsets, and angles in a horizontally layered earth. Both the ray shooting and the two-point raytracing problems are solved. Most acquisition geometries can be simulated and quite general multi-modes can be traced.

The v(x,z) raytrace facility solves the ray-shooting problem in generally heterogeneous 2D media. The two-point problem is not solved. The facility is implemented on a gridded model using the ordinary differential equation for a raypath.

The acoustic finite difference modelling facility has been improved and upgraded. It solves the 2D variable-velocity scalar wave equation for any acquisition geometry. Additionally, an exploding reflector function is provided.

Finally, the v(x,z) raytracing has been extended to include normal incidence raytrace migrations and modelling. An interactive picking facility has been incorporated into *plotimage* that facilitates the real-time determination of picks and subsequent normal raytracing.

ACKNOWLEDGEMENTS

I wish to thank the sponsors of the CREWES project for their support of this research. Pat Daley provided valuable suggestions for the v(x,z) raytracer. Peter Manning and Zhengsheng Yao were also helpful.

REFERENCES

Aki, K., and Richards, P.G., 1980, Quantitative Seismology, Volume 1: W.H. Freeman and Co.

Clayton, R., and Engquist, B., 1977, Absorbing boundary conditions for acoustic and elastic wave equations: Bull. Seis. Soc. Am., 67, 1529-1540.

Lines, L. R. Slawinski, R. and Bording, R. P., 1999, A recipe for stability of finite-difference wave-equation computations: Geophysics, 64, 967-969.

Margrave, G. F., 2000, Numerical Methods of Exploration Seismology with algorithms in MATLAB: released to CREWES sponsors as a preprint.

Manning, P. M., and Margrave, G. F., 2000, Finite difference modelling perfected: 12th Annual Research Report of the CREWES Project.

Press, W.H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. 1992, Numerical Recipes in C: The Art of Scientific Computing: Cambridge University Press.

Shearer, P. M., 1999, Introduction to Seismology: Cambridge University Press.

Slotnick, M. M., 1959, Lessons in Seismic Computing: Society of Exploration Geophysicists.

Youzwishen, C.F., and Margrave, G.F., 1999, Finite difference modeling of acoustic waves in Matlab, in the 11th Annual Research Report of the CREWES Project.

APPENDIX A: THE MATLAB RAYTRACING TOOLBOX

Demo

RAYTRACE_DEMO: interactive demonstration of v(z) raytracing capabilities
RAYVXZ_DEMO: demo the v(x,z) raytrace code

Basic tools for v(z)

DRAWRAY: draws rays given their ray parameters
RAYFAN_A: similar to RAYFAN but the rays are specified by angle
RAYFAN: shoots a fan of rays given their ray parameters for v(z)
SHOOTRAY: similar to RAYFAN but with less error checking (faster)
TRACERAY: traces an arbitrary ray given its raycode for v(z)
TRACERAY_PP: traces a P-P (or S-S) reflection for v(z)
TRACERAY_PS: traces a P-S (or S-P) reflection for v(z)

Tools for v(x,z)

DRAYVEC: compute the derivative of ray vector (for v(x,z) raytracing)
DRAYVECLIN: compute the derivative of ray vector for v0=a*x+b*z
RAYVELMOD: establish a velocity model for v(x,z) raytracing
SHOOTRAYTOSURF: shoot a ray to z=0 in v(x,z)
SHOOTRAYVXZ: RK4 raytracing in v(x,z) with nearest neighbor int.
SHOOTRAYVXZ_G: more general raytracing in v(x,z).

Normal raytracing for v(x,z)

CLEARRAYS: clear (delete) the rays in a figure
EVENTRAYMIG: raytrace migrate a picked event assuming normal incidence
EVENTRAYMOD: raytrace model a picked event assuming normal incidence
NORMRAY: trace a normal ray to the surface
NORMRAYMIG: migrate a normal incidence ray

APPENDIX B: THE MATLAB ACOUSTIC FINITE DIFFERENCE TOOLBOX

Sample scripts

HIGHV_WEDGE: model an anticline beneath a high velocity wedge
VZANTICLINE: model an anticline beneath a v(z) medium
CHANNEL: model a channel beneath a few layers

Seismograms

AFD_SHOTREC ... makes finite difference shot records
AFD_EXPLODE ... makes exploding reflector models

Utilities

AFD_VMODEL ... makes simple polygonal velocity models
AFD_SOURCE ... generates a source array for uses with AFD_SHOTREC
CHANGE_GRID_SPACING ... example script to interpolate a velocity model
AFD_REFLECT ... calculate the reflectivity from a velocity model
AFD_MOVIESNAP: make movies of wavefield propagation

Basic time-stepping

AFD_SNAP ... take one finite difference time step
AFD_SNAPN ... time steps a wavefield "n" steps
DEL2_5PT ... compute the 5 point Laplacian
DEL2_9PT ... compute the 9 point Laplacian
AFD_BC_OUTER ... apply absorbing boundary condition to outer boundary
AFD_BC_INNER ... apply absorbing bcs to inner boundary