B.J. Guillot
GEOL6392
5/8/2003

2D Kirchhoff Migration in Java

1. Introduction

The objective of this project is to develop a simple 2D Kirchhoff migration program in Java that can be used as a tutorial for others to see how a simple summation-based migration algorithm can be coded. The algorithm will be described along with its assumptions and limitations. The Java application here will then be compared to an existing Java applet available online at Stanford. Next, possible future work will be discussed which includes ways to speed up the algorithm along with other enhancements that could be made.

2. Algorithm and Ratfor Code

The core piece of algorithmic code was taken from a subroutine called kirchslow() developed by Jon F. Clarebout, and first published in 1997. The tutorial code presented by Clarebout is very limited, and in his in words is "the simplest zero-offset Kirchhoff program (which is very slow)". It can be called a migration-modeling algorithm (or conjugate code) because it offers both a "forward" mode which can take a impulse model in (x,z) model space and give a result in (x,t) data space, and a "reverse" mode in which the (x,t) data space is used as input and the output is the impulse (x,z) model space.

The algorithm is derived from the circle-hyperbola relation, $t^2 = \tau^2 + x^2/v^2$. In the forward mode, the migration code is considered "hyperbola-like" because τ is given and t is solved for. The code is given in a specialty language called "Ratfor", Rationalized Fortran, which is a hybrid of Fortran and C. A homogenous velocity field is assumed. Clarebout's code:

```
# Kirchhoff migration and diffraction.  (tutorial, slow)
#
subroutine kirchslow(   adj, add,  velhalf, t0,dt,dx, modl,nt,nx,  data)
integer ix,iy,it,iz,nz, adj, add,                          nt,nx
real x0,y0,dy,z0,dz,t,x,y,z,hs,    velhalf, t0,dt,dx, modl(nt,nx), data(nt,nx)
call adjnull(           adj, add,                     modl,nt*nx,  data,nt*nx)
x0=0.;  y0=0;  dy=dx;  z0=t0;  dz=dt; nz=nt
do ix= 1, nx {  x = x0 + dx * (ix-1)
do iy= 1, nx {  y = y0 + dy * (iy-1)
do iz= 1, nz {  z = z0 + dz * (iz-1)            # z = travel-time depth
        hs=      (x-y) / velhalf
        t = sqrt( z * z  +  hs * hs )
        it = 1.5 + (t-t0) / dt
        if( it <= nt )
                if( adj == 0 )
                        data(it,iy) = data(it,iy) + modl(iz,ix)
```

```
                    else
                            modl(iz,ix) = modl(iz,ix) + data(it,iy)

            }}}
    return; end
```

The use of comments is not very helpful here, and the input parameters are not even described in his paper.

The subroutine adjnull() is not defined in the paper, nor described. I am able to deduce the following on the input parameters:

adj	Perform forward modeling is zero, perform migration if one
add	Add to the output if one, erase output if zero
t0	First sample time
dt	Sampling in time
dx	Sampling in x
nt	Number of time samples
nx	Number of samples in x

You can clearly see the lines of code above that solve for t in the $t^2 = \tau^2 + x^2/v^2$ equation:

```
hs=      (x-y) / velhalf                # represents the x/v term
t = sqrt( z * z  +  hs * hs )           # z represents the tau term
```

3. Java Code

The Java algorithm is a direct port of the Ratfor code. I have retained the same variable names to allow for direct comparison.

```
public void kirchslow (
        int adj,                //forward=0, adjoint=1
        double velhalf,         //velocity
        double t0,              //first sample time
        double dt,              //sampling in time
        double dx,              //sampling in x
        double[][] modl,        //model
        int nt,                 //number of time samples
        int nx,                 //number of samples in x
        double[][] data
) {
        double x0=0.0;  double y0=0.0;
        double dy=dx; double z0=t0;
        double dz=dt; int nz=nt;
        double x, y, z; double hs;  double t; int it;

        for (int ix=0; ix < nx; ix++) {
                x = x0 + dx * (double)ix;
                for (int iy=0; iy < nx; iy++) {
                        y = y0 + dy * (double)iy;
                        for (int iz=0; iz < nz; iz++) {
                                z = z0 + dz * (double)iz;
                                hs = (x-y) / velhalf;
                                t = Math.sqrt(z*z + hs*hs);
                                        it = (int)(((t-t0)+0.5) / dt);
                                if (it < nt) {

                                        if (adj == 0)
                                                data[it][iy] = data[it][iy] + modl[iz][ix];
                                        else
                                                modl[iz][ix] = modl[iz][ix] + data[it][iy];
                                }
                        }
                    }
                }
            }
```

The Java code is very similar to the Ratfor code. I removed the input parameter called add which feed the adjnull() subroutine in the original Ratfor code. The routine and input was not necessary in my experiment because I would only need to call the migrate() routine once in forward mode and once in reverse mode. I had no need to add to an existing data or model array. In Java, when you allocate memory for a new array, all elements are initialized to zero, so I did not need to use that feature from the adjnull() routine either. I also attempted to use proper indentation in the for-loops. Clarebout purposely put all the do-loops at the same level to emphasize the fact that the loops were interchangeable, and that, the way the kirchslow() algorithm is constructed, it does not matter what the outside ("slow" index) and inside ("fast" index) indices are. Ratfor array indices run from 1 to N. Java indices run from 0 to N-1 (just like C). The for-loops in the Java code had to be adjusted. The line that calculates the t index also had to have a value adjusted from 1.5 to 0.5 in order for the calculation to select the right location in (x,t) space.

4. Using the Code and Results

Unfortunately, Clarebout did not explain what kind of input values needed to be supplied for the velhalf, dt, and dx variables. I stuck with a 200x200 grid (nt=200 and nz=200) for my model. Using trial and error, I settled on using velhalf=0.1, dt=8.0, and dx=4.0. With those values, I got reasonably shaped hyperbolas when running the forward code in the data-space. Notice dx < dt. The space mesh must be more refined than the time mesh for this algorithm to work, otherwise severe aliasing occurs.

The Java application was designed so that models could be developed using simple paint tools available in various operating systems, such as "mspaint" in Windows. This program can generate a GIF file and you can use various line and curve drawing tools to simplify model creation. After the in-class demonstration, I realized the reason for the bad forward modeling was due to the way that the model was loaded from disk in the GIF file into memory. The background color black was being read as an impulse of –16777216.0 instead of Null. Since it was being treated as a real trace value, hyperbolas were getting drawn everywhere in areas that were not expected. I changed the model load-from-disk code to treat that background black value as a Null value, and then was able to get the expected hyperbola shapes in the forward output.

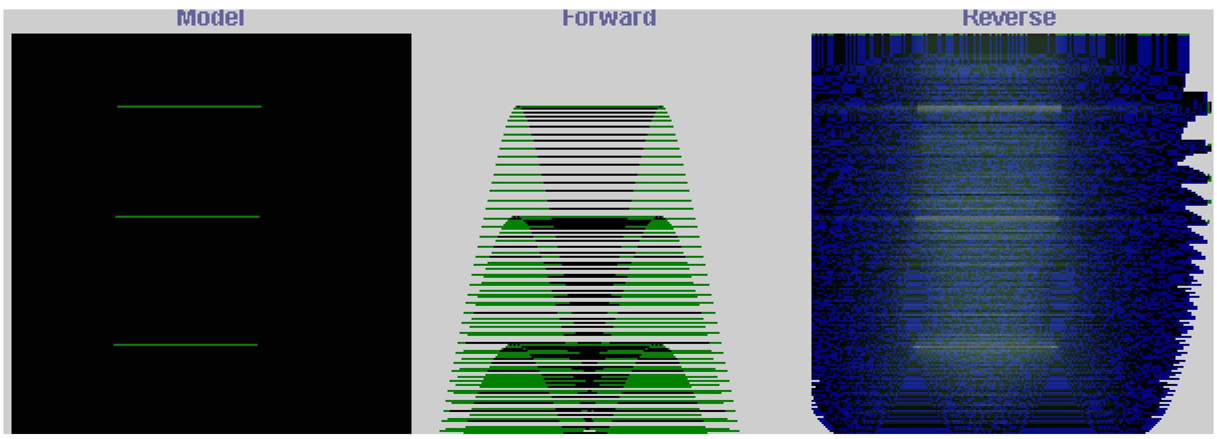

Figure 1. Input model of three horizontal reflectors into Java application to test the forward and reverse code. Notice the shape of the hyperbola in the forward result changes with time/depth, as you would expect. The reverse result, the actual time migration, has had the green color changed to blue in order for the migrated reflectors to stand out more.

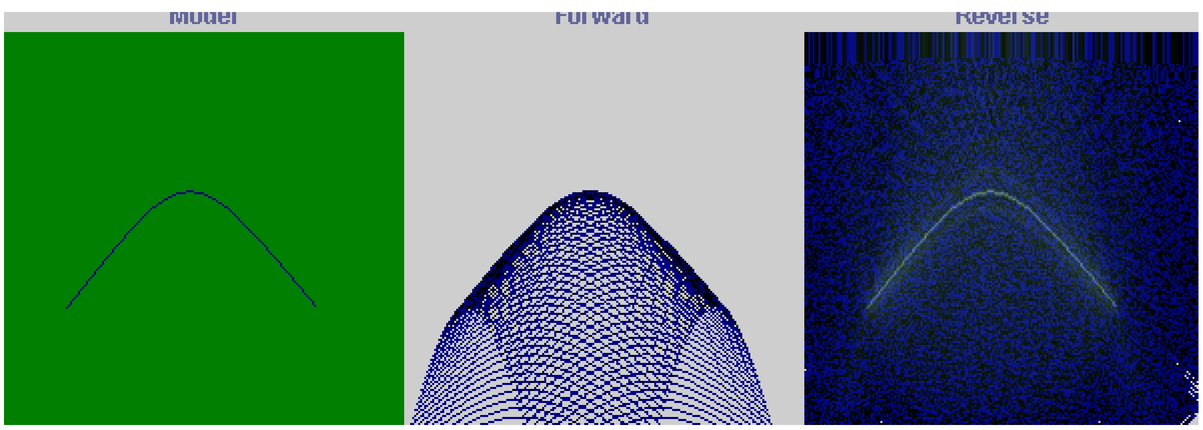

Figure 2. Input model with a single antiform. The migrated image is very well-focused (“crispy”) in this test. Again, the colors have been modified to make the images stand out more.

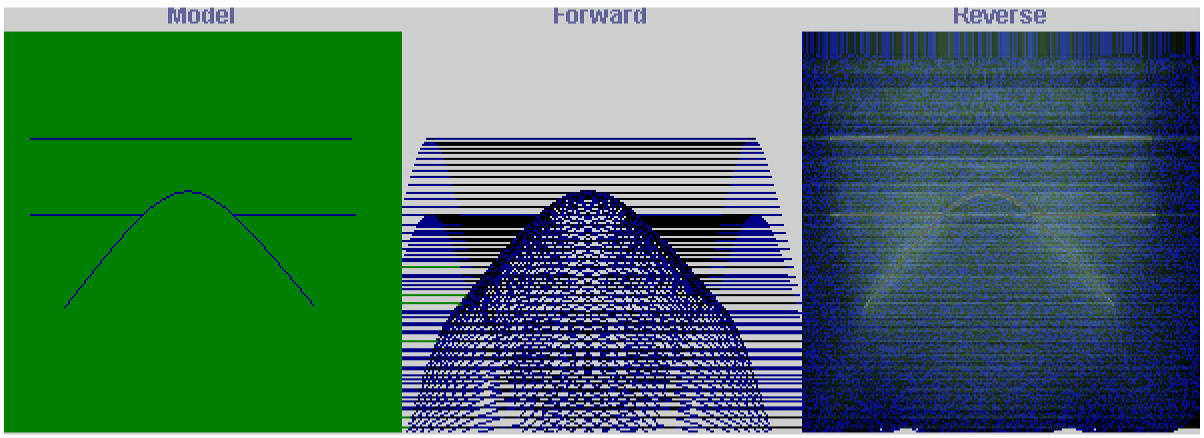

Figure 3. Input model with single antiform cross-cutting a horizontal reflector, with a complete shallow reflector present at top. The migration result is not as well-focused this time. (Colors +have been modified.)

5. Conclusion and Areas for Improvement

The kirchslow() subroutine is basically a brute-force algorithm that does no intelligent checking to see if the values that are being computed actually need to be computed (i.e., it might fall off the buttom of the mesh in model-space or data-space). This happens frequently when $x_{max} >> vt_{max}$. As depth increases, the calculated value will eventually fall off the mesh. Instead of blindly completing the iz for-loop, the algorithm could be modified to break out of the iz loop at that point. However, the simplicity of the algorithm is harmed by this change, and the order of three for-loops no longer become completely interchangeable.

The code can also be speed up if square roots are reused rather than recalculated each time. Since the value depends only on the offset and depth, it can be reused at each ix. The square root calculation is an expensive operation for a computer.

It would not be too terribly difficult to modify the algorithm such that velocity varies with depth. A more accurate migration would result if this were done instead of relying on the current homogenous velocity field assumption.

6. References

Claerbout, J.F. “Introduction to Kirchhoff migration programs”. Stanford Exploration Report 73. November 18, 1997, pages 385-391.

Gray, S.H. “Speed and Accuracy of Seismic Migration Methods”. Mathematical Geophysics Summer School (MGSS). Stanford University.

“Chapter 1. Kirchhoff Migration”. http://sepwww.stanford.edu/sep/prof/waves/krch.pdf