Stanford Exploration Project, Report 73, November 18, 1997, pages 471–479

Short Note

Automatic differentiation

Martin Karrenbach[1]

keywords: *algorithm, modeling, velocity*

INTRODUCTION

In the past the development of automatic differentiation algorithms was driven by the need for the efficient evaluation of exact derivative values. In atmospheric and oceanographic sciences, the need for exact derivative values is particularly great, because researchers in these areas use more and more sophisticated first- and higher-order derivatives in their model description. Although automatic differentiation tools have so far not been widely used in exploration geophysics, they offer possibilities in non linear optimization, complex algorithms, and statistical analysis.

THE PROCESS OF AUTOMATIC DIFFERENTIATION

Automatic differentiation is the process of producing the values of a function's derivative from some representation of the function itself. The main difference between automatic differentiation and symbolic differentiation (like Mathematica) is that automatic differentiation does not generate a symbolic representation of the function. Automatic differentiation is primarily concerned with the differentiation of *algorithms*. That algorithm can be written in a general programming language. This flexibility is particularly useful in exploration geophysics, because though we can often write down the algorithm, computing the Jacobian or even a good approximation of the Jacobian is very tedious, if not impossible analytically.

For example let's take a vector function of the form $f : R^m \rightarrow R^n$. If $X =< x_1, x_2, ..., x_m >$ is a vector of length m and $Y =< y_1, y_2, ..., y_n >$ a vector of length n, depending on what form f takes $Y = f(X)$ (scalar or vector), we have

$$\text{Gradient} \qquad G_i = \frac{\partial f}{\partial x_i} \tag{1}$$

[1]**email:** not available

or

$$\text{Hessian} \qquad H_{ij} = \frac{\partial f}{\partial x_i x_j} \tag{2}$$

or

$$\text{Jacobian} \qquad J_{ij} = \frac{\partial y_i}{\partial x_j} \tag{3}$$

In our application f can take many forms; common ones are global semblance and other objectivity functions used during a minimization process. We generally know very well how to code the objective function, linear or non linear. For example, it is clear how to write the algorithm for the prestack depth migration. However, it is not so easy to calculate the gradient with respect to the velocity model. In many cases the only choice is to approximate the gradient or Jacobian, analytically or numerically. In iterative problems, the convergence directly depends on the Jacobian.

Automatic differentiation applied to algorithms and functions has the advantage of yielding a derivative or Jacobian machine-precision. This method avoids approximations by using preprocessors to high-level languages like Fortran to parse the code. The code is logically split up and data dependency trees are built. One simply has to tell the preprocessor which variable is dependent on which other variable. Usually this is done at the beginning of the code in form of a comment. The automatic differentiator now has two basic choices: it can comb through the data dependency tree from top to bottom (forward propagation), or from bottom to top (reverse propagation). The choice between forward and reverse propagation is efficiently made according to the type of function to be differentiated. The automatic differentiator knows the basic rules of differentiation (the chain rule, the product rule, and so on). It creates additional code, which saves temporary variables that are necessary in order to propagate derivative values through the dependency tree; the details are explained in (Rall, 1981). This temporary storage may increase the overall memory needs of the subroutine, but, depending on your problem, this may be preferable to running your program several times and getting only an approximate value.

There are now several automatic differentiation preprocessors. I have choosen to look at one, which is called *JAKEF* (1990), which is freely available from *NETLIB*. This code has the advantages that it operates on Fortran subroutines and is easy to use. It has however, some severe limitations for geophysical algorithms. In the future I'd like to evaluate a preprocessor, *ADIFOR*, written by Griewank (1991). This new preprocessor is not yet released, but will soon be available. A third preprocessor is available from Kubota (1992); it is called *Padre2*.

AN EXAMPLE

For testing the preprocessor *JAKEF*, I chose to implement Toldi's (1985) objective function for velocity analysis. This example incorporates several numerical computations, which are typically used in exploration geophysics. Toldi's objective function (a simplified version for one CMP) can be described with the following formula and algorithm:

$$\text{semb} = \frac{\{\sum_x Data(x, t = \sqrt{(\tau_i^2 + w_i^2(\mathbf{s})x^2)}\ )\}^2}{\sum_x \{Data(x, t = \sqrt{(\tau_i^2 + w_i^2(\mathbf{s})x^2)}\ )\}^2} \tag{4}$$

This objective function measures the quality of a CMP gather stack by using semblance as a quality measure. Semblance is a nonlinear function of the input parameters, the interval slowness $\mathbf{s}$. In algorithmic terms we can write:

```
for each zero-offset time τ_i
     {
     calculate stacking slowness w_i  (a)
     calcluate moveout curve          (b)
     calculate semblance f(w_i(s))    (c)
     }
add up all semblance values
```

A subroutine (Appendix A) can be written that combines all these three steps. In order to see how we can use automatic differentiation, let's look at a subroutine for step (a):

```
        subroutine rmsv(nt,slowint,rms,dt)
#ONSTRUCT D(RMS)/D(SLOWINT) IN JACOB(NT,NT)
        integer nt
        real slowint(nt)
        real dt
        real rms(nt)
        integer it

        real sum,tau
        real jacob

        do it=1,nt {
           sum = 0.
           do j=1,it {
             sum = sum + dt/(slowint(j)*slowint(j))
           }
           tau = it*dt
           rms(it) =  sqrt ( tau / sum )

        }

#      return stacking velocities rms

        return
        end
```

This subroutine calculates the RMS slowness from a given interval slowness model. The second line in the routine is a keyword construct that tells the preprocessor that we want to calculate the Jacobian of the rms-slowness model (dependent variable) with respect to the interval slowness model (independent variable). The velocity model is a taken for only one CMP gather and constant velocity, just to illustrate the process of automatic differentiation. The preprocessor produces a subroutine in Fortran77 that simultaneously evaluates the RMS velocity function and calculates the Jacobian. Figure **??** shows the interval slowness model and the associated Jacobian for the RMS velocity model. The calculation is accurate to machine precision. We see that the Jacobian has values only below the diagonal, because an RMS slowness value at τ_i does not depend on interval slowness values below the point τ_i. As we can see, the automatic differentiation works well on that piece of the algorithm. The calculation for step (b) proceeds successfully.

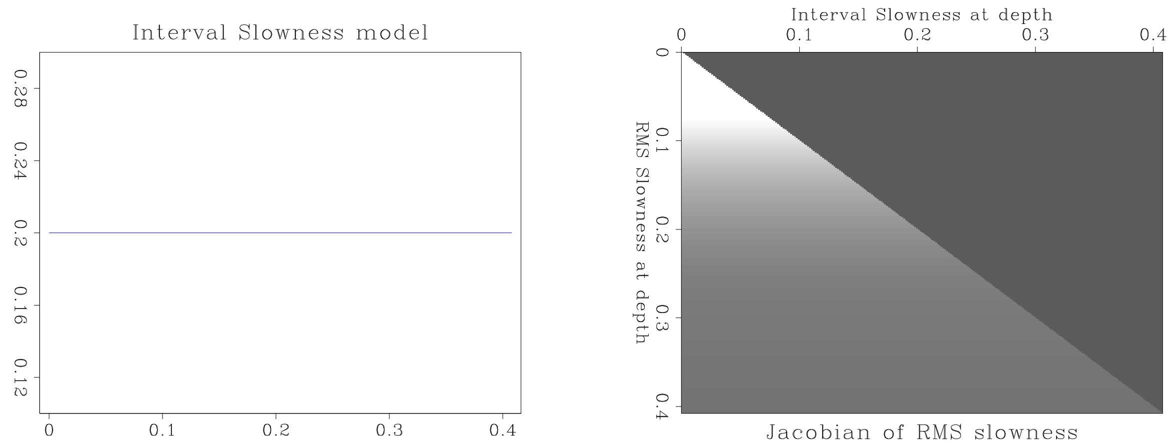

Figure 1: Comparison of the Jacobian of a RMS slowness model and the interval slowness model evaluated at the given model points. *JAKEF* produced a subroutine to evaluate the function and its Jacobian. martin1-fig1 [NR]

The Problem

A big problem arises when calculating the semblance. At this step (c) we have to do index calculations and extract data points. The preprocessor does not know how to handle this instance and we have no way to tell it that the operation is actually a discrete function evaluation. We would like to either have the choice of telling it that this is a valid function or that we are providing another subroutine to evaluate the derivative numerically (eg. a derivative using Fourier transforms would be one alternative). At present *JAKEF* and any other preprocessor do not allow this possibility. Appendix A shows the subroutine that calculates the semblance function. Appendix B shows the *JAKEF* produced version. It proceeds correctly up to the point of stacking the data values along the computed RMS stacking trajectory. Since it does not know how to differentiate the numerical function, the gradient is calculated as zero, and using the chain rule, indicates that the total semblance gradient is zero.

CONCLUSIONS

Presently there are limits of automatic differentiation for the use in exploration geophysical problems. In order to be useful for automatic differentiation of algorithms a preprocessor has to satisfy the following 3 requirements:

1 Since seismic data usually consists of discrete numbers, the preprocessor must allow us to numerically evaluate a given function by supplying either the function or general rule for numerically differentiating the function.

2 Because many algorithms are written as a bundle of subroutine calls, the data dependency analysis must be able to differentiate the subroutines recursively.

3 Index lookup has to be recognized as a numerical function that can be differentiated. (This requirement is closely related to the first.) For example, $data(it, ix)$ must be recognized as a discrete function $d(t, x)$ that can be differentiated.

Preprocessors that meet these requirements would be an extremely useful tool for differentiation of algorithms in exploration geophysics. At present I am in contact with the authors of *Padre2* and *JAKEF* discussing the implementation of the enumerated requirements.

REFERENCES

Griewank, A., 1990, Jakef users guide: NETLIB.

Griewank, A., 1991, personal communication: email.

Kubota, K., 1992, personal communication: email.

Rall, L. B., 1981, Automatic differentiation: Techniques and applications: Springer Verlag.

APPENDIX A

Listing of the Objective Function

```
        subroutine toldi(nt,slowint,semblance,dt,data,nx,dx)
#ONSTRUCT D(SEMBLANCE)/D(SLOWINT) IN GRAD(NT)
        integer nt
```

```
        real slowint(nt)
        real semblance
        real dt
        integer nx
        real data(nt,nx)
        real dx

        real dataval,value
        integer it
        real rms

        real sum,w,tau,x,numerator,denominator
        real grad


        semblance = 0.

        do it=1,nt {
           sum = 0.
           do j=1,it {
             sum = sum + dt/(slowint(j)*slowint(j))
           }
           tau = it*dt
#          slowness at pseudodepth it
           w =  sqrt ( tau / sum )

#          calc  semblance for that pseudo depth, nearst neighbour
           numerator = 0.
           denominator = 0.
           do ix=1,nx {
                value = dataval(ix,it,w,data,nt,nx,dt,dx)
                numerator = numerator + value
                denominator = denominator + value*value
           }
           numerator = numerator*numerator

#          here we should  smooth numerator and denominator


           if (denominator.gt.0.001) {
           semblance = semblance + numerator/(denominator*nx)
                call srite('semb.h',numerator/(denominator*nx),4)
           }else{
                call srite('semb.h',0.,4)
           }

        }

#      return the global semblance value

        return
        end
```

APPENDIX B

Preprocessed Subroutine

```
      SUBROUTINE TOLDIJ(NT,SLOWINT,SEMBLANCE,DT,DATA,NX,DX,GRAD,YGRAD,LY
     *GRAD,RFS,IFS,LFS)
      INTEGER LFS,IFS(LFS)
      REAL RFS(LFS),TGRA(543)
      INTEGER NT,NX,IT,LQ00,LQ01,LQ02,LQ03,J,LQ04,LQ05,IX,LYGRAD,IGRAD,R
     *GRAD,ISLOWI
      REAL SLOWINT(NT),SEMBLANCE,DT,DATA(NT,NX),DX,GRAD(NT),DATAVAL,VALU
     *E,RMS,SUM,W,TAU,X,NUMERATOR,DENOMINATOR,SQRT,YGRAD(LYGRAD)
      ISLOWI = 549
      CALL SPINIT(ISLOWI+NT,LYGRAD)
      CALL EMIT0(1,RFS,IFS,LFS)
      SEMBLANCE = 0.
      LQ00 = 1
      LQ01 = NT
      DO 90001 IT = LQ00,LQ01
      CALL EMIT0(3,RFS,IFS,LFS)
      SUM = 0.
      LQ02 = 1
      LQ03 = IT
      DO 90002 J = LQ02,LQ03
      CALL EMIT2(ISLOWI+J,SLOWINT(J),ISLOWI+J,SLOWINT(J),8,RFS,IFS,LFS)
      TGRA(2) = SLOWINT(J)*SLOWINT(J)
      CALL EMIT1(8,-(DT/(TGRA(2)**2)),7,RFS,IFS,LFS)
      TGRA(1) = DT/TGRA(2)
      CALL EMIT2(3,1.,7,1.,3,RFS,IFS,LFS)
      SUM = SUM+TGRA(1)
90002 CONTINUE
      TAU = IT*DT
      CALL EMIT1(3,-(TAU/(SUM**2)),7,RFS,IFS,LFS)
      TGRA(1) = TAU/SUM
      W = SQRT(TGRA(1))
      CALL EMIT1(7,.5/W,4,RFS,IFS,LFS)
      CALL EMIT0(5,RFS,IFS,LFS)
      NUMERATOR = 0.
      CALL EMIT0(6,RFS,IFS,LFS)
      DENOMINATOR = 0.
      LQ04 = 1
      LQ05 = NX
      DO 90003 IX = LQ04,LQ05
      CALL EMIT0(2,RFS,IFS,LFS)
      VALUE = DATAVAL(IX,IT,W,DATA,NT,NX,DT,DX)
      CALL EMIT2(5,1.,2,1.,5,RFS,IFS,LFS)
      NUMERATOR = NUMERATOR+VALUE
      CALL EMIT2(2,VALUE,2,VALUE,7,RFS,IFS,LFS)
      TGRA(1) = VALUE*VALUE
      CALL EMIT2(6,1.,7,1.,6,RFS,IFS,LFS)
      DENOMINATOR = DENOMINATOR+TGRA(1)
90003 CONTINUE
      CALL EMIT2(5,NUMERATOR,5,NUMERATOR,5,RFS,IFS,LFS)
```

```
      NUMERATOR = NUMERATOR*NUMERATOR
      IF (.NOT.(DENOMINATOR.GT.0.001))GO TO 23006
      CALL EMIT1(6,NX+0.,8,RFS,IFS,LFS)
      TGRA(2) = DENOMINATOR*NX
      CALL EMIT2(5,1./TGRA(2),8,-(NUMERATOR/(TGRA(2)**2)),7,RFS,IFS,LFS)
      TGRA(1) = NUMERATOR/TGRA(2)
      CALL EMIT2(1,1.,7,1.,1,RFS,IFS,LFS)
      SEMBLANCE = SEMBLANCE+TGRA(1)
      CALL SRITE('SEMB.H',NUMERATOR/(DENOMINATOR*NX),4)
      GO TO 23007
23006 CONTINUE
      CALL SRITE('SEMB.H',0.,4)
23007 CONTINUE
90001 CONTINUE
90000 CONTINUE
      RGRAD = 0
      CALL SPGRAD(YGRAD,LYGRAD,1,RGRAD,IGRAD,RFS,IFS,LFS)
      CALL SPCOPY(GRAD,IGRAD,1,YGRAD(ISLOWI+1),NT)
      RETURN
      END
```