A=1
K=0
R=11

AUTOMATIC DIFFERENTIATION IN PROSE

F. W. Pfeiffer
Memorex Corporation
Santa Clara, California

INTRODUCTION

Much of the popular numerical software for solving nonlinear equations and nonlinear optimization problems require arrays of partial derivatives. Computing these derivatives efficiently and accurately has been an obstacle since the time of Newton. A little-known approach to computing derivatives, however, has made this obstacle disappear for users of one programming language. This short article presents an overview of the approach and its implementation in the language.

AUTOMATIC DIFFERENTIATION

Automatic Differentiation (AD) is a relatively new approach to computing derivatives of known continuous functions numerically. It is efficient, and it is exact to machine accuracy. The approach can be used to compute derivatives of any order, and it can be applied to functions of real, complex, and interval variables. While the approach has remained unknown to most of those in the numerical community, it has actually been in use by a small number of developers and users since the 1960s.

The main idea that led to AD was that the formulas for the derivatives of elementary functions could be preprogrammed in a programming languages mathematical functions library -- right along with the formulas that evaluated the function f(x). Then when a particular function was evaluated in some computer program, several numerical values could be returned to the calling program. The number returned in this type of implementation would depend on the highest order derivative formula preprogrammed.

In addition, the four elementary arithmetic operations (+,-,*,/) on variables and functions, and the derivatives of these operations on variables and functions, were preprogrammed in the library as well. In other words, the following three operations might have (optionally) been performed in SUBROUTINE ADD.

U	+	V	(zeroth-order arithmetic)
U'	+	V'	(first-order arithmetic)
U''	+	V''	(second-order arithmetic)

In the early years of AD, code for the four elementary arithmetic operations (+,-,*,/) was moved into the subroutines where the code for elementary derivative

operations was located. Sometime in the future, however, the elementary derivative operations will be moved into hardware where the elementary arithmetic operations are performed. In either case, the numerical algorithms for the elementary arithmetic operations in AD are considerably different. This is a new kind of scientific or extended computer arithmetic which some implementors are now trying to put into firmware [4], [10] and eventually into hardware.

One of the chief advantages of AD over symbolic differentiation (the generation of inline derivative formulas) is that it is space efficient. Symbolic differentiation in the usual manner generates formulas at an explosive rate. AD does not generate formulas at an explosive rate because it requires only one copy of each elementary derivative formula. Twenty years ago, this gave AD a very important advantage over symbolic differentiation. Nowadays, however, this advantage does not seem quite as important. Another advantage of this newer approach -- an advantage which should turn out to be far more important -- is that it allows the independent variables of differentiation to change dynamically as many times as required during execution. This capability is needed when nesting numerical methods.

The basic ideas underlying AD are relatively useless, however, unless they are automated in computer software, and quite naturally, the real burden of AD falls upon the system software, i.e., the compilers, interpreters, linkers, and loaders -- not the user. The compiler for a language having AD imbedded in it, for example, must not only generate instructions to compute user's functions, but it must also generate instructions to compute derivatives of these functions. The generation of these machine instructions and their execution can occur in a variety of ways. Some of what has been published on these differentiating compilers can be found by referring to [3], [5], [6], [7], and [9]. Only five references are listed here; the reader will find pointers to others in these.

PROSE

PROSE is usually referred to as a calculus-level programming language. The language was developed in the early 1970s by J. M. Thames and M. N. Robinson while at PROSE Inc. in Los Angeles, California. Their objective was to develop a general-purpose programming language that would better satisfy the numerical needs of computer users in the engineering and scientific communities. Their desire to create a new language for scientific computing was inspired by the obvious mathematical deficiencies of languages like BASIC and FORTRAN. These two men reasoned that a new language which had easy-to-use, language-supplied features for the numerical solution of major classes of problems encountered often in engineering and science would be far more useful than the languages then in use. Some of the major classes of interest to them were:

* Differentiation
* Integration (Quadrature)
* Algebraic Equations
* Ordinary Differential Equations
* Partial Differential Equations
* Optimization

Thames and Robinson had worked together on the automation of mathematical software while at TRW, Inc. in Redondo Beach, California during the late 1960s. Somewhere along the way,

they discovered that the automation of this software required the automation of derivative evaluation. They were driven to AD by the very real need to make algorithms for simulation, optimization, and parameter estimation work efficiently together in single large systems. So with the techniques of AD mastered after years of hard work in this area, these two software developers had little difficulty, evidently, in automating the mathematical software of their choice in PROSE years later. By 1976, they had automated software in all of the classes above, except PDEs, and then had to stop due to a lack of financial support.

AUTOMATIC DIFFERENTIATION IN PROSE

Nothing has been published on the details of automatic differentiation in PROSE, but a small number of facts are known to a handful of users. Among these are that PROSE makes use of preprogrammed formulas for first and second derivatives for its operators and library; although today, Thames and Robinson evidently prefer a recurrence formula approach which allows for higher order derivatives. Also, PROSE computes derivatives interpretively. (PROSE functions like an interpreter during execution.) This may seem inefficient, but this approach allows PROSE to change independent variables dynamically during execution, and this in turn allows PROSE users to nest numerical methods. This capability to nest methods efficiently is usually well worth the penalty of high-level interpretation.

PROSE can perform AD in two different ways. The first way is simply at the user's request. The user simply makes use of statements for computing and assigning derivatives. In this way, the user is free to print, plot, or use the derivative values in formulas or other algorithms. The second way is a result of using a numerical method in the PROSE library which requires derivatives. The use of these numerical methods is accomplished through the use of other statements in the language -- described in [2].

DERIVATIVES BY REQUEST

One can get derivatives of real and complex variables computed and stored by using statements like one of the following.

```
EXECUTE .EQUATIONS WITH GRADIENTS ON U, V, W
EXECUTE .OBJECTIVE WITH HESSIANS  ON X, Y, Z, T
```

First derivatives (GRADIENTS) of every variable in the block named .EQUATIONS that is a function of U, V, and W will be computed and stored. In a similar manner, first and second derivatives (HESSIANS) of every variable in block .OBJECTIVE that is a function of X, Y, Z, and T will be computed and stored. These derivatives will be evaluated at the current value of the independent variables. Occasional checks over the years have shown these derivative values to agree with values from symbolic derivatives to at least 14 digits on CDC and IBM computers.

Derivative values are assigned to variables in PROSE when one of three, built-in functions is executed. Four examples of these three functions are shown here:

```
E = .PARTIAL( U, X )          ∂U/∂X
F = .PARTIAL( U, X, Y )       ∂²U/∂X∂Y
```

```
G = .GRAD( V )          ∇V
H = .HESS( V )          ∇²V
```

The first two statements above are scalar assignment statements. The variable E will be assigned the value of the derivative of U with respect to X. Similarly, F will be assigned the value of the second mixed derivative of U with respect to X and Y. The last two statements are a little different, in that they are both array assignment statements. PROSE knows that .GRAD(V) is a vector, and so dynamically allocates the variable G as a vector with the same dimensions and then makes the assignment. In a similar manner, .HESS(V) is a matrix, and so H becomes a matrix just before the assignment.

The Jacobian matrix of a real function cannot be obtained by the user in the same way as the Hessian; instead, a sequence of gradients evaluations and array assignments must be made to construct the Jacobian dynamically, row by row. This is not to say, however, that PROSE does not compute Jacobians. PROSE will always compute the Jacobian automatically for any numerical method in its library that requires it.

Let us now examine a very simple example of computing first and second derivatives in PROSE. Suppose that we wanted to compute the gradient vector and the Hessian matrix of the real scalar function

```
F = SINH( X ) * COSH( Y ) + COSH( X ) * SINH( Y )
```

at X = 1 and Y = 2. The following program would do just this.

```
PROBLEM .DIFFERENTIATION
  X = 1
  Y = 2
  EXECUTE .EQUATIONS WITH HESSIANS ON X, Y
  VECTOR PRINT G
  MATRIX PRINT H
END

MODEL .EQUATIONS
  F = .SINH(X) * .COSH(Y) + .COSH(X) * .SINH(Y)
  G = .GRAD(F)
  H = .HESS(F)
END
```

In all PROSE programs, execution begins with the first executable statement in the PROBLEM block, i.e., the statement "X = 1" in this case. Execution of the EXECUTE statement causes PROSE to execute the block, MODEL .EQUATIONS, with first and second derivative evaluation turned on. As PROSE is computing F in .EQUATIONS, it will also compute the six derivatives requested simultaneously and store them. The necessary instructions to do this were interleaved by the PROSE compiler earlier. (PROSE functions like a compiler during setup.) AD will be turned on only during the execution of this block. Execution of the assignment statements for G and H result in the dynamic allocation of G as a 2 element vector containing the gradient of F and H as a 2X2 matrix containing the Hessian of F. Execution of this program then returns to the PROBLEM block, where the preformatted output statements VECTOR PRINT and MATRIX PRINT print G and H.

DERIVATIVES AUTOMATICALLY

The example above illustrates how one could get first and second derivatives if this is all one wanted. The reader will now see that the job of computing derivatives is performed automatically by PROSE whenever numerical methods that require them are used. As a result, there may be no visible evidence that derivatives are being computed in some PROSE programs, while in fact, derivative evaluation may be quite heavy.

Let us now look at an example of PROSE computing derivatives automatically. Suppose that we wanted to estimate the parameters A and B to fit the following ODE to some measurements.

$$\frac{d^2Y}{dT^2} = B^2 * Y - 2 * A * B * EXP(-B * T)$$

For this problem, we will use two numerical methods in nested combination. The outer method will be used to minimize the absolute error between the measured and computed values of Y. The inner method will be used to solve the ODE IVP at the current values of A and B -- thus giving us the computed value of Y at each measured value of T. For this parameter estimation problem, the author wrote the following program.

```
PROBLEM .DIFFERENTIATION.2
  ALLOT TM(11), YM(11)
  READ DATA
  A = 2   B = 1   DT = 0.1
  FIND A, B IN .DIFF.EQUATION
    BY JOVE
    TO MINIMIZE ERROR
END

MODEL .DIFF.EQUATION
  T = 0   Y = 0   YDOT = 5   ERROR = 0
  INITIATE ISIS FOR .EQUATION
    EQUATIONS Y2DOT/YDOT, YDOT/Y OF T STEP DT
  FOR I = 1 TO 11 DO
    UNTIL T GE TM(I) INTEGRATE .EQUATION
    ERROR = ERROR + ( YM(I) - Y )**2
  REPEAT
END

MODEL .EQUATION
  Y2DOT = B**2 * Y - 2 * A * B * .EXP( -B * T )
END
```

A short description of what happens during the execution of this program will be given block by block. For additional details, the reader is asked to consult [1] and [2].

PROBLEM .DIFFERENTIATION.2 -- Execution of this block begins with the ALLOT statement. ALLOT dynamically allocates TM and YM as 11-element vectors. The READ DATA statement reads in the measured values of T and Y and stores them in TM and YM. (For details on this I/O operation, consult Chapter 8 in [1].) The next line contains three statements which provide initial guesses for A and B, and the nominal stepsize to be used by the ODE solver. The FIND statement partially defines the minimization problem to be solved. This statement declares (1) that the unknowns are A and B, (2) that a SUMT/Newton solver name JOVE is to be used, and

(3) that the objective function to be minimized is ERROR. Execution of the FIND statement leads to the execution of JOVE, and this in turn, leads to the iterative execution of the next block.

MODEL .DIFF.EQUATION -- Execution of this block begins with the resetting of the initial conditions for Y and YDOT at T = 0 at each iteration of the minimization method. This block is called 121 times by JOVE. The INITIATE statement activates and initializes the ODE solver named ISIS, a fourth-order Runge-Kutta-Gill method. The FOR-DO loop, and the statements therein, allow PROSE to calculate the absolute error between the measured and computed values of Y at each measured value of time TM(I) by solving to ODE IVP up to TM(I) and then stopping to add the square of the I-th absolute error to ERROR. Upon termination of this loop, execution returns to the minimization solver with the value of the objective function at the current values of A and B. Also returned to the minimization solver will be the first and second derivatives of ERROR with respect to A and B.

MODEL .EQUATION -- This block contains the definition of the ODE in the initial value problem. The ODE solver (ISIS) calls this block 401 times for each iteration the minimization solver (JOVE).

Fitting this ODE to data using nested numerical methods is made possible by PROSE's ability to compute the derivatives of ERROR in the minimization problem with respect to the parameters A and B in the ODE problem. One of the internal design features that makes this possible is that the ODE solver is written in the PROSE language, and any code written in PROSE -- even PROSE's own numerical methods -- can be differentiated using the language's own AD capabilities. PROSE's approach to computing derivatives in this particular situation allows for the efficient nesting of methods which require derivatives -- a situation in which numerical and symbolic differentiation would very likely be impractical as ways of evaluating derivatives efficiently.

CONCLUDING REMARKS

Some readers may find the implementation of automatic differentiation in PROSE not quite as thorough as they might like. The implementation is limited to first and second derivatives (gradient, Jacobian, and Hessian), and so any need for higher-order derivatives cannot be satisfied directly. This limitation to first and second derivatives is a result of the designers' main objective, i.e., the automation of the popular mathematical software -- not computing arbitrary order derivatives. For arbitrary order derivatives, the implementation is not as thorough as it is in Pascal-SC [6], [7], [8].

The author has used the PROSE implementation of automatic differentiation on CDC, IBM, and Univac computers since 1974. The availability of AD in this programming language, even though limited to first and second derivatives, has virtually eliminated his need for numerical differentiation and substantially reduced his need for symbolic differentiation. It has been a great relief during these last 12 years to have one of the major obstacles to the use of many powerful and general numerical methods removed from one's own scientific programming activities.

REFERENCES

1. Control Data Corporation, PROSE Procedure Manual, Pub. No. 84003000, 1977

2. Control Data Corporation, PROSE Calculus Operations Manual, Pub. No. 84003200, 1977

3. Kedem, G. Automatic Differentiation of Computer Programs, ACM TOMS, June 1980

4. Krinsky, B. and Thames, J. M. The Structure of Synthetic Calculus. In Proceedings of the International Workshop on High-Level Language Computer Architecture, Los Angeles, Calif. 1984

5. Rall, L. B. Automatic Differentiation: Techniques and Applications. Lecture Notes in Computer Science, vol. 120, Springer-Verlag, New York, 1981.

6. Rall, L. B. Differentiation and Generation of Taylor Coefficients in Pascal-SC. In A New Approach to Scientific Computation, U. W. Kulisch and W. L. Miranker, Eds., Academic Press, New York, 1983

7. Rall, L. B. Differentiation in Pascal-SC: Type Gradient, ACM TOMS, June 1984

8. Rall, L. B. Computational Implementation of the Multivariate Halley Method for Solving Nonlinear Systems of Equations, ACM TOMS, March 1985

9. Speelpenning, B. Compiling Fast Partial Derivatives of Functions Given by Algorithms, Ph.D. dissertation, University of Illinois, 1980

10. Thames, J. M.
The Evolution of Synthetic Calculus: A Mathematical Technology for Advanced Architecture
Proceedings of the International Workshop on High-Level Language Computer Architecture
Fort Lauderdale, Florida, 1982

11. Wengert, R. E. A Simple Automatic Derivative Evaluation Program, Comm. ACM, August 1964