Preprint MCS-P358-0493, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., May 1993.

Operator Overloading as an Enabling Technology for Automatic Differentiation*

George F. Corliss
Marquette University and
Argonne National Laboratory

Andreas Griewank
Argonne National Laboratory

Abstract

We present an example of the science that is enabled by object-oriented programming techniques. Scientific computation often needs derivatives for solving nonlinear systems such as those arising in many PDE algorithms, optimization, parameter identification, stiff ordinary differential equations, or sensitivity analysis. Automatic differentiation computes derivatives accurately and efficiently by applying the chain rule to each arithmetic operation or elementary function. Operator overloading enables the techniques of either the forward or the reverse mode of automatic differentiation to be applied to real-world scientific problems. We illustrate automatic differentiation with an example drawn from a model of unsaturated flow in a porous medium. The problem arises from planning for the long-term storage of radioactive waste.

1 Introduction

Scientific computation often needs derivatives for solving nonlinear partial differential equations. One such problem currently under investigation at Argonne and Sandia National Laboratories involves planning for long-term storage of radioactive waste.

The specific problem is drawn from a mathematical model of unsaturated flow in a porous medium modeled as a system of nonlinear partial differential equations. Solving the model requires the repeated solution of large, sparse nonlinear systems of algebraic equations. Newton's method is preferred for its rapid convergence, but Newton's method requires the fast, accurate computation of derivatives.

Automatic differentiation provides an attractive mechanism for meeting these requirements. Automatic differentiation is a prime example of the benefits of object-oriented programming techniques. It computes derivatives accurately and efficiently by applying the chain rule to each arithmetic operation or elementary function. Moreover, because it propagates values rather than expressions, automatic differentiation is not subject to the expression swell common in symbolic differentiation.

Many researchers have used overloaded operators with the forward mode of automatic differentiation to attack problems of modest size. For problems with many independent variables, however, the reverse mode of automatic differentiation is required for efficiency.

*This work was supported in part by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38 and through NSF Cooperative Agreement No. CCR-8809615.

The software package ADOL–C [18] offers researchers a means for applying the reverse mode efficiently and effectively. ADOL–C includes overloaded operators that record each arithmetic operation and elementary function and write the operands and the results on a "tape." The tape is subsequently interpreted to provide the derivatives required for Newton's method. The same tools have been applied with algorithms for solving nonlinear equations, optimization, parameter identification, stiff ordinary differential equations, or sensitivity analysis.

In this paper, we show how operator overloading is a critical enabling technology for ADOL–C. We begin in Section 2 with a sketch of a specific case study. In Section 3, we compare automatic differentiation with traditional methods for computing derivatives; and we discuss the principles underlying automatic differentiation technology. Section 4 focuses on the role of object orientation. We show that automatic differentiation techniques can be implemented in various modes, requiring different tradeoffs between speed and ease of use. In Section 5, we present results obtained when the different modes are used, and we draw some conclusions about the future of automatic differentiation in scientific problem-solving.

2 Unsaturated Flow Problem

A mathematical model of steady-state flow in a porous medium involves an elliptic partial differential equation (PDE) that contains a conductivity coefficient [12, 24]. The coefficient is typically discontinuous across different materials and can vary greatly. For unsaturated flow, the conductivity is usually taken to be a function of pore pressure, which introduces a nonlinearity into the problem. The nonlinearity for materials such as tuff can become severe enough to dominate the problem, as conductivity can change markedly with a small change in pressure. Our interest was in modeling flow in a region consisting of fractured tuff with conductivities that vary by ten or more orders of magnitude, often over very short distances.

We began with a code in C furnished by Tom Robey. His code uses a mixed finite-element approach with a quasi-Newton iteration to handle the very high nonlinearity. He calculates a very sparse 1989×1989 Jacobian J by centered differences. The resulting linear equation is solved by a biconjugate gradient algorithm. We approached Robey's code hoping to show the superiority of the ADOL–C [18] implementation of automatic differentiation over centered differences. We believed that automatic differentiation could improve both the accuracy and the speed by using operator overloading in C++. We modified the original code of Robey in three steps (for further details, see [8, 10]).

1. **Convert to C++.** Robey's original code was written in C. We converted function headers to a form acceptable to C++, removed some system-dependent calls, and added system-dependent timing instrumentation.

2. **Change to derivative type.** We replaced the type `double` by the derivative type `adouble` inside the function to be differentiated.

3. **Compute derivatives.** Code for computing the Jacobian by centered differences was replaced by code to record the tape as described in Section 4 and to call an interpreter to generate the derivatives in either the forward or the reverse mode.

The automatic differentiation code used the same sparse matrix techniques as the original code. Since many components of F in the nonlinear system $F(u) = 0$ are linear in u, we need to compute only a 350×936 subblock $\widehat{D}$ at each Newton iteration. Further, $\widehat{D}$ is sparse and can be computed as three columns or as eight rows using matrix-coloring techniques [6].

3 Alternative Methods for Computing J

A common paradigm for the numerical solution of two-, three-, or higher-dimensional partial differential equations is as follows:

- Given a PDE and boundary conditions,
- apply finite-difference or finite-element approximations on some appropriate (frequently nonuniform) grid, and
- enforce an approximate solution by solving a nonlinear system $F(u) = 0$ for the residual by Newton's method.

The dimension of the nonlinear system $F(u) = 0$ is proportional to the number of grid points. Typically, the Jacobian $J = \partial F / \partial u$ required by Newton's method is computed by some combination of hand coding, symbolic processing, and divided differences. In this section, we survey these alternative methods for computing J. We conclude the section with a discussion of the two modes of automatic differentiation — forward and reverse. For both modes, we use operator overloading.

3.1 Hand Coding

Algorithms usually perform best when the analytic Jacobian J is hand coded. Unfortunately, hand coding of derivatives is a tedious, time-consuming, and error-prone task. However in many problems, hand coding is feasible because many dependencies of F on u are linear. If the linear dependencies can be separated from the nonlinear ones, hand coding of at least portions of J becomes easier. Also, certain finite elements or differencing stencils often appear in many copies or minor variations, so that the effort of hand differentiation is limited and does not increase with the size of the grid.

3.2 Symbolic Processing

Symbolic processors such as Maple or Mathematica are becoming more powerful in their ability to provide symbolic expressions for derivative objects. They are excellent tools for modest-sized problems. However, as the size and complexity of the code to be differentiated grow, one reaches the limits of symbolic processors. Maple addresses this limitation by providing a library for automatic differentiation to work on Maple procedures.

3.3 Divided Differences

The Jacobian J can be approximated by backward, centered, or forward divided differences. An appropriate choice of step size is difficult, however, especially when there may be differences of scale between different components. Therefore, the accuracy of divided-difference approximations is in doubt. A naive coding perturbing each component of u in turn is easy, but very expensive. Applications usually attempt to exploit any known sparsity structure.

3.4 Matrix Coloring and Partial Separability

Matrix coloring [6] and partial separability [20] are techniques commonly used in computing divided differences efficiently for sparse Jacobians. Coloring identifies several columns of J that can be computed simultaneously with the same computational effort as a single column. The Jacobian of a partially separable function can be written as a sum of several simple, smaller Jacobians. Matrix coloring and partial separability also work well with the techniques of automatic differentiation.

3.5 Automatic Differentiation – Forward Mode

We illustrate automatic differentiation [13, 16, 23] with an example. Assume that we have the sample program shown in Figure 1 for the computation of a function $f : \mathbf{R}^2 \mapsto \mathbf{R}^2$. Here, the vector `x` contains the independent variables, and the vector `y` contains the dependent variables. The function described by this program is defined except at $\mathtt{x}[2] = 0$ and is differentiable except at $\mathtt{x}[1] = 2$. We can transform the program in Figure 1 into one for computing derivatives by associating a derivative object $\nabla\mathtt{t}$ with every variable `t`. Assume that $\nabla\mathtt{t}$ contains the derivatives of `t` with respect to the independent variables `x`, $\nabla\mathtt{t} = \left(\frac{\partial \mathtt{t}}{\partial \mathtt{x}[1]}, \frac{\partial \mathtt{t}}{\partial \mathtt{x}[2]}\right)^T$. We propagate these derivatives by using elementary differentiation arithmetic based on the chain rule [13, 23] for computing the derivatives of `y[1]` and `y[2]`, as shown in Figure 2. In this example, each assignment to a derivative is actually a vector assignment of length two. This example contains a branch and a loop to illustrate that the forward mode handles control structures. The only difficulty is that functions with branches may fail to be differentiable. Work is in progress to warn the programmer when this has occurred.

```
if x[1] > 2 {
   a = x[1] + x[2];
}
else {
   a = x[1] * x[2];
}
for (i = 1; i <= 2; i ++) {

   a = a * x(i);
}
y[1] = a / x[2];


y[2] = sin (x[2]);
```

Figure 1. Sample function $f : \mathrm{x} \mapsto \mathrm{y}$

```
if x[1] > 2.0 {
   a = x[1] + x[2];
   ∇a = ∇x[1] + ∇x[2]; }
else {
   a = x[1] * x[2];
   ∇a = x[2] * ∇x[1] + x[1] * ∇x[2]; }
for (i = 1; i <= 2; i ++) {
   temp = a;
   a = a * x(i);
   ∇a = x(i) * ∇a + temp * ∇x(i); }
y[1] = a / x[2];
∇y[1] = 1.0 / x[2] * ∇a
         - a / (x[2] * x[2]) * ∇x[2];
y[2] = sin (x[2]);
∇y[2] = cos (x[2]) * ∇x[2];
```

Figure 2. Augmented with derivative code

This mode of automatic differentiation, where we maintain the derivatives with respect to the independent variables, is called the *forward mode* of automatic differentiation. The forward mode is easy to implement using overloaded operators, and many authors have done so (see the survey in [21]).

3.6 Automatic Differentiation – Reverse Mode

The *reverse mode* of automatic differentiation maintains the derivative of the result with respect to intermediate quantities, called *adjoints,* which measure the sensitivity of the result with respect to some intermediate quantity. The reverse mode requires fewer operations than the forward mode if the number of independent variables is larger than the number of dependent variables. This is exactly the case for computing a gradient, which is a Jacobian matrix with only one row. For sparse problems, the number of variables alone is not the determining factor. Instead, the complexity of the forward and the reverse modes can be bounded in terms of the chromatic numbers of the column- and row-incidence graphs, respectively. This issue is discussed in more detail in [17, 16].

Wolfe observed [25], and Baur and Strassen confirmed [1], that if care is taken in handling quantities that are common to the (rational) function and its derivatives, then the cost of evaluating a gradient with n components is a small multiple of the cost of evaluating the underlying scalar function. Despite the advantages of the reverse mode from the viewpoint of complexity, the implementation for the general case is quite complicated. It requires the ability to access *in reverse order* the instructions executed for the computation of f and the values of their operands and results, as illustrated in Figure 3. Current tools achieve this by storing a record of every computation performed. An interpreter performs a backward pass on this "tape." Unfortunately, the resulting overhead sometimes destroys the complexity advantage of the reverse mode (see [11]). In the unsaturated flow problem, complexity analysis predict no clear advantage for either mode over the other. The forward mode proved to be only about 28 % faster than the reverse mode, although the reverse mode had to compute 7 times as many values!

The reverse mode associates an adjoint object with every variable u, $\overline{u} = \frac{dw}{du}$, for each dependent variable w. The key rule for the reverse mode is

$$s = f(t,u) \quad \Rightarrow \quad \begin{array}{l} \overline{t} \mathrel{+}= \ \overline{s} * (df/dt) \\ \overline{u} \mathrel{+}= \ \overline{s} * (df/du) \end{array} . \tag{1}$$

Figures 3 and 4 illustrate the application of the reverse mode of automatic differentiation to the function f given in Figure 1. This example contains a branch and a loop to illustrate the difficulties of the reverse mode. Straight-line code and basic blocks can be reversed by source-transformation tools, but code containing data-dependent branches and loops can be reversed only at run time. Therefore, we execute the code for the function f shown at the top of Figure 3 using operators overloaded to record the code list shown at the bottom of Figure 3. Since the code list is a recording of the operations actually executed at run time, it contains no branches, and all loops are fully unrolled.

Figure 4 propagates adjoint quantities to compute the derivatives of y with respect to x. Adjoint objects (vectors of length two in this example) are initialized. Then, the code list at the bottom of Figure 3 is processed *in reverse order* as shown by the comments in Figure 4. Formula (1) is applied to each entry from the code list. At the end of the computation, the desired derivatives are in `xbar` and `ybar`.

```
if x[1] > 2 {
   a = x[1] + x[2];
}
else {
   a = x[1] * x[2}
for (i = 1; i <= 2; i ++) {
   a = a * x(i);
}
y[1] = a / x[2];
y[2] = sin (x[2]);

Yields the code list:

// Assuming x[1] ≤ 2
a = x[1] * x[2];
t1 = a * x[1];
t2 = t1 * x[2];
y[1] = t2 / x[2];
y[2] = sin (x[2]);
```

Figure 3. Reversing execution of f

```
    // Initialize adjoint objects:
ybar[1] = (1, 0); ybar[2] = (0, 1);
xbar[1] = xbar[2] = abar
        = t1bar = t2bar = (0, 0);
    // y[2] = sin (x[2]);
xbar[2] += ybar[2] * cos (x[2]);
    // y[1] = t2 / x[2];
t2bar += ybar[1] / x[2];
xbar[2] += ybar[1] * (-t2 / (x[2] * x[2]);
    // t2 = t1 * x[2];
t1bar += t2bar * x[2];
xbar[2] += t2bar * t1;
    // t1 = a * x[1];
abar += t1bar * x[1];
xbar[1] += t1bar * a;
    // a = x[1] * x[2];
xbar[1] += abar * x[2];
xbar[2] += abar * x[1];
// ∇y[1] = (xbar[1][1], xbar[2][1]);
// ∇y[2] = (xbar[1][2], xbar[2][2]);
```

Figure 4. Augmented for the reverse mode

This discussion shows that the principles underlying automatic differentiation are not complicated. We associate extra computations (which are entirely specified on a statement-by-statement basis) with the statements executed in the original code, a task for which operator overloading is well suited. As a result, a variety of implementations of automatic differentiation have been developed over the years (see [21] for a survey). We also note that even though we showed the computation only of first derivatives, the automatic differentiation approach can easily be generalized to the computation of univariate Taylor series or Hessians and multivariate higher-order derivatives [5, 7, 9, 15, 17, 23].

4 Role of Object Orientation

The ADOL–C tool for automatic differentiation relies critically on overloaded operators in C++. In this section, we discuss the role of operator overloading. The details of the ADOL–C implementation can be found in [18].

4.1 Forward Mode

There are three alternatives for the forward-mode propagation of derivative objects: naive operator overloading, source transformation, and use of a tape. We discuss each alternative in turn.

Figures 1 and 2 implicitly give the specifications for a naive operator overloading for the forward-mode propagation of derivative objects. It is straightforward to write overloaded operators and functions to propagate simple derivatives, gradients, Jacobians, Hessians, single-, or multi-variable Taylor series. One defines a class containing the desired derivative type and overloaded operations and elementary functions according the recurrence relations given by Rall [23], for example. Figure 5 shows part of such a class for interval-valued functions written in C–XSC [22]. Many authors have written such a package (see the survey in [21]). Corliss [7] described an implementation in Ada of real- and interval-valued Taylor series with storage management for efficient one-term-at-a-time generation.

```
class autodiff {
    //  Author:   Andreas Wiethoff, University of Karlsruhe
  private:
    interval f, df;
  public:
    autodiff();                        // Constructors
    autodiff(interval&, interval&);
    ~autodiff();                       // Destructors
    autodiff& operator = (interval&);  // Assignment
    autodiff& operator = (autodiff&);
      // Arithmetic operators and functions:
    friend autodiff independent(interval&);
    friend autodiff operator + (autodiff&, autodiff&);
    friend autodiff operator - (autodiff&, autodiff&);
    friend autodiff operator * (autodiff&, autodiff&);
    friend autodiff operator / (autodiff&, autodiff&);
    friend autodiff sin (autodiff&);
    friend autodiff exp (autodiff&);

    friend interval Value (autodiff& a) {  return a.f;  }
    friend interval Deriv (autodiff& a) {  return a.df; }
};

autodiff operator * (autodiff& a, autodiff& b)
{  // Derivative of a product:  c = a * b  ==>  c' = a * b' + a' * b
  autodiff c;
  c.f = a.f * b.f;  c.df = a.f * b.df + a.df * b.f;
  return c;
}

autodiff sin (autodiff& a)
{ // Derivative of sine:  c = sin(a)  ==>  c' = a' * cos(a)
  autodiff c;
  c.f = sin(a.f);  c.df = a.df * cos(a.f);
  return c;
}
```

Figure 5. Prototype for naive forward mode operator overloading

Naive operator overloading is easy to implement and easy to use because the program for the function to be differentiated is very nearly the same as the program for evaluating the function at a real argument. Overloading has an advantage of flexibility. For example, operators for automatic differentiation can easily be modified to use complex, interval, or multiple precision arithmetic. The programs tend to execute slowly because they contain many function calls that interfere with compile-time code optimization. Speed can be improved by placing many of the operators in line, but that causes code growth. Speed can be improved and code growth reduced by better compilation techniques. In principle, programs that place function calls for overloaded operators in line and then do code optimization should execute as fast as conventional compiled code.

Source transformation uses a preprocessor (ADIFOR [3], for example) to generate code similar to that shown in Figure 2. The source transformation approach is conceptually similar to the operator overloading approach in the sense that the programmer writes code in the original development language without being directly concerned that derivatives will be generated using the same code. Source transformation has several advantages over naive overloading. The output from the source transformation is compilable. The resulting code usually runs faster than the equivalent overloaded code because the source transformation tool can do code optimizations such as common subexpression removal or loop mining. The speed advantage of source transformation is likely to be reduced as better compilers for overloading become available. The source transformation tool can generate code for the reverse mode evaluation of basic blocks and parallel loops. Even the partial use of the reverse mode in the context of an overall forward-mode propagation of derivatives gives a significant speed improvement in some examples [3].

The disadvantage of source transformation techniques is that it is difficult to handle the full reverse mode in the presence of branches and data-dependent loops. The reverse mode for functions containing branches and data-dependent loops requires the run time support furnished by operator overloading.

The third alternative for the forward-mode propagation of derivative objects involves use of a tape. As we execute the function to be differentiated, the overloaded arithmetic operators and elementary functions record a code list (operation, operands, result) on a tape. Constructors and destructors also record "birth" and "death" notices on the tape. The tape is a complete record of the computation with loops unrolled and branches actually taken. It is accessed in a strictly sequential mode, and it may be in main memory, on a disk file, or on a magnetic tape, depending on the running time of the program. The length of the tape is usually proportional to the running time of the function to be differentiated. Work is in progress for a checkpointing option in ADOL–C that reduces the length of the tape required to a logarithm of the original running time at the expense of increasing the undifferentiated running time [14]. Once the tape has recorded the sequence of operations in the evaluation of f, a separate function call interprets the code list in order and propagates the desired derivatives.

Forward-mode propagation using the tape is good for repeated evaluations that follow exactly the same path of control through the execution of f. This happens when f has no branches or data-dependent loops, when some iterative process has nearly converged, or when Taylor series must be generated one term at a time (as for solving ODEs). Using the tape for the forward mode also has the advantage of presenting an interface that is consistent with the interface for the reverse mode.

The disadvantage of using the tape is that it is slower than the naive overloaded operators, because the tape approach incurs the significant added overhead in writing and reading the tape. However, run time access to the code list provides opportunities for significant improvements in efficiency, including

- run time code optimizations such as removal of common subexpressions,
- parallel scheduling [2, 4], or

- Markowitz elimination of nodes from the computational graph [19].

4.2 Reverse Mode

Figure 4 implicitly gives the specification for the reverse-mode propagation of derivatives. The increased complexity of the implementation is justified by the result of Baur and Strassen [1] that a gradient can be evaluated at a cost of about five times the cost of a function evaluation, *independent* of the number of independent variables. As discussed in Section 3.6, the reverse mode requires the ability to reverse the order of execution of the code list. We use a tape recorded by overloaded operators as described above. Derivatives are computed by a separate function call during which the code list is processed *in reverse order.* Formula (1) is applied to each entry, and we propagate a vector of adjoints with one element for each dependent variable.

5 Results

The problem considered here is a test problem with one spatial dimensional exhibiting a particularly simple structure. We have illustrated a strategy that generalizes to higher-dimensional problems of practical interest. We make several direct comparisons of automatic differentiation with more traditional methods.

- In comparison with centered differences, the values of the derivatives computed in double precision by automatic differentiation agreed to at least five digits. (In general, the derivatives computed by automatic differentiation are more accurate than those computed by divided differences. In some applications, the improved accuracy enables Newton's method to converge in fewer iterations, but that was not significant in this application.)

- In comparison with centered differences, the fastest ADOL–C code took 56 % longer. Table 1 gives timing comparisons for some of the versions tested on a SPARC 1+ workstation using the GNU g++ compiler, version 2.3.3. We report the average times for ten Jacobian evaluations.

Table 1. CPU Times for Jacobian computation

Method	Seconds
Three-color centered differences	1.30
Three-color forward mode	2.03
Eight sweeps of reverse mode	5.88
Eight-vector reverse mode	2.60

The Jacobian J is a highly structured 1989×1989 matrix. The nonlinear contributions are contained in a 350×936 subblock that is sparse and can be computed as three columns (1050 values) or as eight rows (7488 values) using matrix coloring techniques [6]. The forward mode propagated three gradient vectors of length 350. The tape for the three-color forward mode evaluation was 520 Kbytes long. The reverse mode computed eight gradient vectors of length 936. The reverse mode computed 7 times as many values as the forward mode in only 28 % more time.

Acknowledgments

The work described in this paper was done in collaboration with Thomas Robey, Christian Bischof, and Steve Wright. We offer a special thanks to Andreas Wiethoff for permission to include a portion of his `class autodiff`.

References

[1] W. Baur and V. Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22:317–330, 1983.

[2] Christian Bischof. Issues in parallel automatic differentiation. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 100–113. SIAM, Philadelphia, Penn., 1991.

[3] Christian Bischof, Alan Carle, George Corliss, and Andreas Griewank. ADIFOR – Generating derivative codes from Fortran programs. *Scientific Programming*, 1:11–29, 1992.

[4] Christian Bischof, Andreas Griewank, and David Juedes. Exploiting parallelism in automatic differentiation. In E. Houstis and Y. Muraoka, editors, *Proceedings of the 1991 International Conference on Supercomputing*, pages 146–153. ACM Press, Baltimore, Md., 1991.

[5] Bruce D. Christianson. Automatic Hessians by reverse accumulation. Technical Report NOC TR228, The Numerical Optimisation Center, Hatfield Polytechnic, Hatfield, U.K., April 1990.

[6] T. F. Coleman and J. J. Moré. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, 20:187–209, 1984.

[7] George Corliss. Overloading point and interval Taylor operators. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 139–146. SIAM, Philadelphia, Penn., 1991.

[8] George Corliss, Andreas Griewank, Tom Robey, and Steve Wright. Automatic differentiation applied to unsaturated flow — ADOL–C case study. Technical Memorandum ANL/MCS–TM–162, Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Ill., April 1992.

[9] George Corliss and Louis B. Rall. Automatic generation of Taylor series in Pascal–SC: Basic operations and applications to differential equations, in *Trans. of the First Army Conference on Applied Mathematics and Computing (Washington, D.C., 1983)*, pages 177–209. ARO Rep. 84-1, U. S. Army Res. Office, Research Triangle Park, N.C., 1984.

[10] George Corliss, Thomas Robey, Christian Bischof, Andreas Griewank, and Steve Wright. Automatic differentiation for PDEs — Unsaturated flow case study. In Robert Vichnevetski, Doyle Knight, and Gerard Richter, editors, *Advances in Computer Methods for Partial Differential Equations – VII*, pages 150–156. IMACS, New Brunswick N.J., 1992.

[11] Lawrence C. W. Dixon. Use of automatic differentiation for calculating Hessians and Newton steps. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 114–125. SIAM, Philadelphia, Penn., 1991.

[12] Richard Ewing and Mary Wheeler. Computational aspects of mixed finite element methods. In R. Stepleman et al., editors, *Scientific Computing*. IMACS/North-Holland Publishing Company, 1983.

[13] Andreas Griewank. On automatic differentiation. In M. Iri and K. Tanabe, editors, *Mathematical Programming: Recent Developments and Applications*, pages 83–108. Kluwer Academic Publishers, Dordrecht, 1989.

[14] Andreas Griewank. Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. *Optimization Methods and Software*, to appear.

[15] Andreas Griewank. Automatic evaluation of first- and higher-derivative vectors. In R. Seydel, F. W. Schneider, T. Küpper, and H. Troger, editors, *Proceedings of the Conference at Würzburg, Aug. 1990, Bifurcation and Chaos: Analysis, Algorithms, Applications*, volume 97, pages 135–148. Birkhäuser Verlag, Basel, Switzerland, 1991.

[16] Andreas Griewank. The chain rule revisited in scientific computing. *SIAM News*, 24, May & July 1991. No. 3, p. 20 & No. 4, p. 8.

[17] Andreas Griewank. Some bounds on the complexity of gradients, Jacobians, and Hessians. To appear in P. M. Pardalos, editor, *Complexity in Numerical Optimization*, World Scientific Publishers, 1993.

[18] Andreas Griewank, David Juedes, Jay Srinivasan, and Charles Tyner. ADOL-C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software*, to appear.

[19] Andreas Griewank and S. Reese. On the calculation of Jacobian matrices by the Markowitz rule. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 126–135. SIAM, Philadelphia, Penn., 1991.

[20] Andreas Griewank and Ph. L. Toint. On the unconstrained optimization of partially separable functions. In M. J. D. Powell, editor, *Nonlinear Optimization 1981*, NATO Conference Series II: Systems Science, pages 301–321. Academic Press, New York, 1982.

[21] David Juedes. A taxonomy of automatic differentiation tools. In Andreas Griewank and George F. Corliss, editors, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*, pages 315–329. SIAM, Philadelphia, Penn., 1991.

[22] R. Klatte, U. Kulisch, C. Lawo, M. Rauch, and A. Wiethoff. *C–XSC : A C++ Class Library for Extended Scientific Computing*, Springer Verlag, Berlin, 1993.

[23] Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1981.

[24] Mary Wheeler and Ruth Gonzalez. Mixed finite element methods for petroleum reservoir engineering problems. In R. Glowinski and J.-L. Lions, editors, *Computing Methods in Applied Sciences and Engineering, VI*. Elsevier, New York, 1984.

[25] Philip Wolfe. Checking the calculation of gradients. *ACM Trans. Math. Softw.*, 6(4):337–343, 1982.