On the Implementation of Automatic Differentiation Tools

Christian H. Bischof (bischof@sc.rwth-aachen.de)
Institute for Scientific Computing
Aachen University of Technology
Seffenter Weg 23
52074 Aachen, Germany

Paul D. Hovland (hovland@mcs.anl.gov) and Boyana Norris (norris@mcs.anl.gov)
Mathematics and Computer Science Division
Argonne National Laboratory
9700 S. Cass Avenue
Argonne, IL 60439-4844

Abstract. Automatic differentiation is a semantic transformation that applies the rules of differential calculus to source code. It thus transforms a computer program that computes a mathematical function into a program that computes the function and its derivatives. Derivatives play an important role in a wide variety of scientific computing applications, including numerical optimization, solution of nonlinear equations, sensitivity analysis, and nonlinear inverse problems. We describe the forward and reverse modes of automatic differentiation and provide a survey of implementation strategies. We describe some of the challenges in the implementation of automatic differentiation tools, with a focus on tools based on source transformation. We conclude with an overview of current research and future opportunities.

Keywords: Semantic Transformation, Automatic Differentiation

1. Introduction

Derivatives play an important role in a variety of scientific computing applications, including optimization, solution of nonlinear equations, sensitivity analysis, and nonlinear inverse problems. Automatic, or algorithmic, differentiation technology provides a mechanism for augmenting computer programs with statements for computing derivatives (Griewank, 1989; Griewank, 2000). In general, given a subprogram that computes a function $f : x \in \mathbf{R}^n \mapsto y \in \mathbf{R}^m$ with n inputs and m outputs, automatic differentiation tools provide a subprogram that computes $f' = \partial y / \partial x$, or the derivatives of the outputs y with respect to the inputs x. In order to produce derivative computations automatically, automatic differentiation tools systematically apply the chain rule of differential calculus at the elementary operator level.

The basic ideas of automatic differentiation date back to the 1950s (Nolan, 1953; Kahrimanian, 1953; Beda et al., 1959). Over the past decade, however, the use of automatic differentiation increased in popularity, as robust tools such as ADIFOR (Bischof et al., 1992; Bischof et al., 1996) and ADOL-

```
subroutine foo(s,A,n)

integer i,n
double precision f,g,s,A(n)

g = 0
do i = 1,n
   g = g + A(i)*A(i)
enddo
g = sqrt(g)

s = 0
do i = 1,n
  call func(f,A(i),g)
  s = s + f
enddo

return
end
```

```
subroutine func(f,x,y)

double precision f,x,y,a,b

if (x .gt. y) then
  a = sin(y)
  b = log(x)*exp(x-y)
else
  a = x*sin(x)/y
  b = log(y)
end if
f = exp(a*b)

return
end
```

Figure 1. An example subprogram.

C (Griewank et al., 1996) became available. These tools have been applied to many applications with several hundred thousand lines of source code, including one (FLUENT) with over a million lines of source code (Bischof et al., 2001). Because automatic differentiation computes derivatives analytically and sytematically, it does not incur the numerical errors inherent in finite difference approximations, nor does it exhibit the propensity for mistakes characteristic of hand-coding. Also, if a program changes, as often occurs during code development, an up-to-date version of the derivative computation is immediately available.

The rest of the paper is organized as follows. Section 2 provides an introduction to automatic differentiation. Section 3 surveys the various implementation strategies for automatic differentiation tools. Section 4 describes the implementation of one compiler-based tool. Section 5 reviews the implementation of automatic differentiation tools and summarizes current research and future opportunities.

2. Introduction to Automatic Differentiation

Automatic differentiation is a family of methods for obtaining the derivatives of functions computed by a program (see (Griewank, 2000) for a detailed discussion). automatic differentiation couples rule-based differentiation of language built-ins (elementary operators and intrinsic functions) with derivative accumulation according to the chain rule of differential calculus. The associativity of the chain rule leads to many possible "modes" of combining partial derivatives. Consider, for example, the function `func` given in Figure 1, evaluated at $x = \frac{\pi}{2}, y = \pi$. If we follow the false branch (since $x < y$) and reduce to three address code, we have the following.

```
t0 = sin(x)
t1 = x*t0
a = t1/y
b = log(y)
t2 = a*b
f = exp(t2)
```

Then,

$$\frac{\partial f}{\partial (x,y)} = \frac{\partial f}{\partial t2}\left(\frac{\partial t2}{\partial a}\left(\frac{\partial a}{\partial t1}\left(\frac{\partial t1}{\partial t0}\frac{\partial t0}{\partial (x,y)} + \frac{\partial t1}{\partial (x,y)}\right) + \frac{\partial a}{\partial (x,y)}\right) + \frac{\partial t2}{\partial b}\frac{\partial b}{\partial (x,y)}\right).$$

So,

$$\dot{f} = f\left(b\left(\frac{1}{y}\left(x \cdot \cos x \cdot \dot{x} + t0 \cdot \dot{x}\right) - \frac{a}{y}\dot{y}\right) + a \cdot \frac{1}{y} \cdot \dot{y}\right),$$

where $\dot{x}$, $\dot{y}$, and $\dot{f}$ are total derivatives. Evaluating at $x = \frac{\pi}{2}, y = \pi$, and assuming the vectors $\dot{x}$ and $\dot{y}$ have been initialized appropriately,[1] one can compute $\dot{f}$ as

$$\dot{f} = \sqrt{\pi}\left(\ln(\pi)\left(\frac{1}{\pi}\left(\frac{\pi}{2} \cdot 0 \cdot \dot{x} + 1 \cdot \dot{x}\right) - \frac{1}{2\pi}\dot{y}\right) + \frac{1}{2} \cdot \frac{1}{\pi} \cdot \dot{y}\right). \tag{1}$$

The *forward mode* combines partial derivatives starting with the input variables and propagating forward to the output variables, or, in the parenthesized form of Equation 1, from the inside out. Thus, $\dot{f}$ could be computed as follows:

```
t0 = sin(x)                                              {1.0000, 1}
prtl0 = cos(x)                                           {0.0000, 1}
t0dot(1:n) = prtl0 * xdot(1:n)                           {0*xdot, N}
t1 = x*t0                                                {1.5708, 1}
t1dot(1:n) = t0*xdot(1:n) + x*t0dot(1:n)                  {xdot, 3N}
a = t1/y                                                 {0.5000, 1}
prtl0 = 1.0/y                                            {0.3183, 1}
prtl1 = -a/y                                            {-0.1591, 1}
adot(1:n) = prtl0*t1dot(1:n) + prtl1*ydot(1:n)
                                  {0.3182*xdot - 0.1591*ydot, 3N}
b = log(y)                                               {1.1447, 1}
prtl0 = 1.0 / y                                          {0.3183, 1}
bdot(1:n) = prtl0*ydot(1:n)                           {0.3183*ydot, N}
t2 = a*b                                                 {0.5724, 1}
t2dot(1:n) = b*adot(1:n) + a*bdot(1:n)
                                  {0.3644*xdot - 0.02303*ydot, N}
f = exp(t2)                                              {1.7725, 1}
fdot(1:n) = f*t2dot(1:n)          {0.6458*xdot - 0.04083*ydot, N}
```

with each statement annotated with the approximate numerical value at $x = \frac{\pi}{2}, y = \pi$ and the number of floating point operations. The reverse mode combines partial derivatives starting with the output variables and propagating backward to the input variables, or, in the parenthesized form of Equation 1, for the outside in. The *reverse mode* computes $\bar{x} = \partial f/\partial x$ and $\bar{y} = \partial f/\partial y$, which can be combined with $\dot{x}$ and $\dot{y}$ according to the chain rule to yield $\dot{f}$. The reverse mode can be implemented as follows.

[1] For the ith invocation of func, xdot should be initialized to the ith unit vector and ydot to (1/g)A.

```
t0 = sin(x)                                              {1.0000, 1}
prtl0 = cos(x)                                           {0.0000, 1}
t1 = x*t0                                                {1.5708, 1}
prtl1 = t0                                               {1.0000, 0}
prtl2 = x                                                {1.5708, 0}
a = t1/y                                                 {0.5000, 1}
prtl3 = 1.0/y                                            {0.3183, 1}
prtl4 = -a/y                                            {-0.1591, 1}
b = log(y)                                               {1.1447, 1}
prtl5 = 1.0 / y                                          {0.3183, 1}
t2 = a*b                                                 {0.5724, 1}
prtl6 = b                                                {1.1447, 0}
prtl7 = a                                                {0.5000, 0}
f = exp(t2)                                              {1.7725, 1}
prtl8 = f                                                {1.7725, 0}
fbar = 1.0                                               {1.0000, 0}
t2bar = prtl8 * fbar                                     {1.7725, 1}
bbar = prtl7 * t2bar                                     {0.8862, 1}
abar = prtl6 * t2bar                                     {2.0290, 1}
ybar = prtl5 * bbar                                      {0.2821, 1}
ybar = ybar + prtl4 * abar                              {-0.04083, 2}
t1bar = prtl3 * abar                                     {0.6458, 1}
t0bar = prtl2 * t1bar                                    {1.0145, 1}
xbar = prtl1 * t1bar                                     {0.6458, 1}
xbar = xbar + prtl0 * t0bar                              {0.6458, 2}
fdot(1:n) = xbar*xdot(1:n) + ybar*ydot(1:n)
                                {0.6458*xdot - 0.04083*ydot, 3N}
```

The forward mode has a total cost of 12N+10 operations, and the reverse mode has a total cost of 3N+21 operations. For large N, therefore, the reverse mode is significantly cheaper than the forward mode. This is true for all functions with a single output variable, not just the example shown. This feature makes the reverse mode extremely attractive for computing the derivatives of scalar functions, especially functions with a large number of input variables (Griewank, 1989). A disadvantage of the reverse mode, however, is that (in a naive implementation), the storage requirements grow in proportion to the number of operations in the function evaluation. This is because partial derivatives (and the intermediate variables used in computing the partial derivatives) are used in the reverse order of that in which they are computed. Except for small programs or code segments, this cost is too high. Consequently, practical implementations of the reverse mode rely on checkpointing strategies (Griewank, 1992; Restrepo et al., 1998; Grimm et al., 1996; Faure, 2001) or use interprocedural dataflow analysis to determine what quantities need to be stored.

In the preceding discussion of the forward and reverse modes, we ignored the control flow and the differentiation of subroutine `foo`. We also simplified the presentation of the forward and reverse modes by relying on the fact that the three address code was in single assignment form. Figure 2 shows the code generated by the TAPENADE automatic differentiation tool (TAPENADE,

```
C        Generated by TAPENADE     (INRIA, Tropics team)
C  Version 2.0.6 - (Id: 1.14 vmp Stable -
Thu Sep 18 08:35:47 MEST 2003)
C
C  Differentiation of func in reverse (adjoint) mode:
C   gradient, with respect to input variables: x y
C   of linear combination of output variables: f x y
C
      SUBROUTINE FUNC_B(f, fb, x, xb, y, yb)
      DOUBLE PRECISION f, fb, x, xb, y, yb
      DOUBLE PRECISION a, ab, arg1, arg1b, b, bb
      INTEGER branch
      INTRINSIC EXP, SIN, LOG
C
C
      IF (x .GT. y) THEN
        a = SIN(y)
        arg1 = x - y
        b = LOG(x)*EXP(arg1)
        CALL PUSHINTEGER4(0)
      ELSE
        a = x*SIN(x)/y
        b = LOG(y)
        CALL PUSHINTEGER4(1)
      END IF
      CALL PUSHREAL8(arg1)
      arg1 = a*b
      f = EXP(arg1)
      arg1b = EXP(arg1)*fb
      CALL POPREAL8(arg1)
      ab = b*arg1b
      bb = a*arg1b
      CALL POPINTEGER4(branch)
      IF (branch .LT. 1) THEN
        arg1b = LOG(x)*EXP(arg1)*bb
        xb = xb + EXP(arg1)*bb/x + arg1b
        yb = yb + COS(y)*ab - arg1b
      ELSE
        yb = yb + bb/y - x*SIN(x)*ab/y**2
        xb = xb + (x*COS(x)/y+SIN(x)/y)*ab
      END IF
      END
```

Figure 2. Reverse mode for subroutine func.

2002) for the subroutine `func`. The complete code for `foo` and `func` using the forward and reverse modes is included in the Appendix.

The automatic differentiation community has developed its own terminology for concepts that may have other names in other research communities. We provide a few definitions to help guide the reader of this survey and other papers on automatic differentiation.

Independent variables A subset of the input variables for a program or subprogram and the set of variables with respect to which one wishes to

differentiate. By convention, we denote the number of independent variables using N.

Dependent variables A subset of the output variables for a program or subprogram and the set of variables whose derivatives we wish to compute. By convention, we denote the number of dependent variables using M.

Computational graph The directed acyclic graph (DAG) for a statement, basic block, or execution trace. Vertex labels are operators or functions and optionally variable names. This graph is often called the "DAG for an expression" or "DAG of a basic block" in the compiler literature (Muchnick, 1997; Aho et al., 1986). Figure 3 shows the computational graph for the simple example.

Linearized computational graph The computational graph with symbolic or numeric edge weights equal to the partial derivative of the target with respect to the source vertex. The derivative of a root vertex with respect to a leaf vertex is the sum over all paths of the product of the edge weights along that path (Rote, 1990). Figure 3 shows the linearized computational graph for the simple example.

Derivative accumulation Application of the chain rule, typically using either forward or reverse mode.

Preaccumulation Computing the partial derivatives for a statement, basic block, or other program subunit. The local partial derivatives are then used in the overall derivative accumulation. If the number of in or out variables for the subunit is significantly smaller than the number of total or directional derivatives being accumulated via the forward or reverse mode, preaccumulation can result in significant time savings.

Cross-country preaccumulation Combining the partial derivatives in an order other than forward or reverse. In terms of the linearized computational graph, this corresponds to multiplying edge weights in some order other than topological or reverse topological. There are exponentially many possibilities.

Activity analysis Identifying the relevant set of variables (and possibly statements) in the program chop from the independent variables to the dependent variables (Reps and Rosay, 1995; Binkley and Gallagher, 1996), that is, identifying the set of variables lying along a dataflow path from the independent variables to the dependent variables. Variables along a path are termed active, and variables not along any path are termed passive. There is no need to compute or store derivatives for passive variables, and automatic differentiation tools that can identify passive vari-

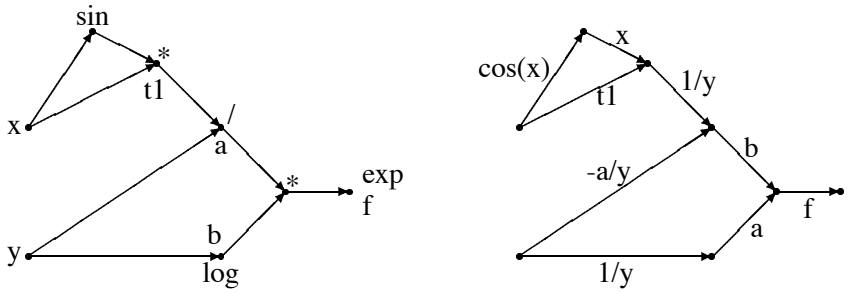

Figure 3. Computational graph and linearized computational graph for the simple example.

ables are able to achieve significant memory and time savings (Bischof et al., 1996).

3. Brief Taxonomy of Automatic Differentiation Tools

Many implementation strategies exist for automatic differentiation. The various strategies frequently trade off simplicity, performance, elegance, and versatility. Furthermore, because automatic differentiation is inexorably tied to a function implemented in some programming language, the implementation strategy must be compatible with that language. For example, prior to Fortran 90, there was no way to implement automatic differentiation via operator overloading in Fortran. We discuss the various implementation strategies, using roughly the same taxonomy as Juedes (Juedes, 1991). We provide examples of each implementation strategy; additional information on available automatic differentiation software can be found at `http://www.-autodiff.org/Tools/`. We also note that several languages and environments, including AMPL (Fourer et al., 1993), GAMS (Brooke et al., 1988), and Maple (Monagan and Rodoni, 1996) provide built-in support for automatic differentiation.

3.1. Elemental Approaches

The first automatic differentiation tools (Wengert, 1964; Wilkins, 1964; Lawson, 1971; Jerrell, 1989; Hinkins, 1994; Hill and Rich, 1992; Neidinger, 1989) required the user to replace arithmetic operations (and possibly calls to intrinsic functions) with calls to a differentiation library. For example, the statement `a = x*sin(x)/y` would be rewritten as

```
call adsin(t1,t1dot,x,xdot)
call adprod(t2,t2dot,x,xdot,t1,t1dot)
call addiv(a,adot,t2,t2dot,y,ydot)
```

The library computes partial derivatives and applies the chain rule. For example, `addiv` might be implemented as

```
subroutine addiv(q,qdot,n,ndot,d,ddot)
double precision q,n,d,qdot,ndot,ddot
double precision t1,t2

t1 = 1.0/d
t2 = n/d          ! alternatively, n*t1
q = t2
qdot = t1*ndot - t1*t2*ddot
return
```

This strategy is invasive and not well suited to functions defined by large programs. For languages without operator overloading, however, it is the simplest strategy to implement.

3.2. Operator Overloading

For languages that support operator overloading, automatic differentiation may be implemented by using either a simple forward mode approach or a trace-based approach.

3.2.1. *Forward Mode*

In its simplest form, operator overloading introduces a new class that carries function values and derivatives (collectively called Rall numbers (Barton and Nackman, 1996) or doublets (Bartholomew-Biggs et al., 1995)). The arithmetic operators and intrinsic functions are overloaded to compute partial derivatives and apply the chain rule. An abbreviated example of such a class and its usage appears in Figure 4. Because the forward mode is very easy to implement with operator overloading, it is frequently used as an example in classrooms and textbooks (Barton and Nackman, 1996) and has been implemented in many different languages, including Ada (Huss, 1990; Maany, 1989), C++ (Martins et al., 2001; I. Tsukanov, 2003; Bendtsen and Stauning, 1996; Michelotti, 1991; Kalman and Lindell, 1991), Fortran 90 (Stamatiadis et al., 2000), Haskell (Karczmarczuk, 2001; Nilsson, 2003), MATLAB (Forth, 2001), and Python (Hinsen, 2003). Because automatic differentiation can be implemented in a few hours for most languages with operator overloading, there are probably countless “throwaway” implementations written for a single application or for the needs of an individul researcher. Many of the tools described in Section 3.2.2 also provide a simple implementation of the forward mode. The performance of operator overloading can be improved through a variety of standard techniques, including expression templates (Aubert and Di Césaré, 2001; Veldhuizen, 1995) and lazy evaluation (Christianson et al., 1996).

```
class adouble{
private:
  double value, grad[GRAD_LENGTH];
public:
  /* constructors omitted */
  friend adouble operator*(const
    adouble &,const adouble &);
  /* similar decs for other ops */
}
adouble operator*(const adouble &g1,
  const adouble &g2){
    int i;
    double newgrad[GRAD_LENGTH];
    for(i=0;i<GRAD_LENGTH;i++){
      newgrad[i] =
        (g1.value)*(g2.grad[i])+
        (g2.value)*(g1.grad[i]);
    }
 return
   adouble(g1.value*g2.value,newgrad);
}
```

```
main(){
  double temp[GRAD_LENGTH];
  adouble y;

  /* initialize
       x1 to (3.0,[1.0 0.0]),
       x2 to (4.0,[0.0 1.0])*/
  temp[0] = 1.0; temp[1] = 0.0;
  adouble *x1 =
    new adouble(3.0,temp);
  temp[0] = 0.0; temp[1] = 1.0;
  adouble *x2 =
    new adouble(4.0,temp);
  y = (*x1)*(*x2);

  cout << y;
  /* prints (12.0,[4.0 3.0])*/
}
```

Figure 4. A simplified example of operator overloading.

3.2.2. *Trace-Based Techniques*

An alternative strategy to computing derivatives directly with the forward mode is to use operator overloading to generate an execution trace (frequently called a "tape") of all mathematical operations and their arguments. This trace can subsequently be traversed in reverse order, accumulating derivatives with the reverse mode. Using this strategy, researchers have developed reverse mode tools for Ada (Christianson, 1991),C++ (Bendtsen and Stauning, 1996; Bell, 2003; Griewank et al., 1996), Fortran 90 (Bartholomew-Biggs, 1995; Brown, 1995; Pryce and Reid, 1998), MATLAB (Coleman and Verma, 2000), and Python (Frazier, 2003). Alternatively, the trace can be used to construct and linearize the computational graph of the function. The linearized computational graph can be reduced to bipartite form, yielding the Jacobian matrix, with a variety of heuristics (Griewank and Reese, 1991).

3.2.3. *Related Techniques*

Many early source transformation tools approximated the behavior of operator overloading by translating arithmetic operations into calls to an elemental differentiation library. Juedes (Juedes, 1991) used the term *extensional* to describe such tools.

In languages with complex arithmetic, a similar effect can be achieved through small, imaginary perturbations (Martins et al., 2000; Martins et al., 2001). These cause the complex numbers to behave like Rall numbers, carrying function values in the real field and (scaled) derivatives in the imaginary field (Griewank, 1998; Martins et al., 2001). Lesk proposed overloading the complex type to implement automatic differentiation directly (Lesk, 1967)

3.3. COMPILER-BASED STRATEGIES

Source-to-source transformation strategies rely on compiler technology to transform source code for computing a function into source code for computing the derivatives of the function (as a side effect of the automatic differentiation mechanism, the function itself is also computed). This approach offers the advantage of static analyses, such as identifying the active variables that lie along the computational path from independent variables to dependent variables. Also, because the analysis is performed at precompile time, the search for an effective cross-country ordering for combining partials can use expensive (polynomial time) algorithms that could not be used at run time.

Most source-to-source transformation tools (Bischof et al., 1996; Rostaing et al., 1993; Giering and Kaminski, 1998; Giering and Kaminski, 2002; TAPENADE, 2002; Tadjouddine et al., 2003; NAG-AD, 2003) have targeted Fortran, but tools have also been developed for C (Bischof et al., 1997), MATLAB (Bischof et al., 2002), and Mathematica (Korelc, 2001). These tools typically implement forward mode with statement-level reverse-mode preaccumulation, and several also implement reverse mode accumulation. They frequently use interprocedural dataflow analysis to identify active variables. Recent tools have added new analyses (Faure and Naumann, 2001; Naumann, 2002) or incorporated cross-country preacummulation at the basic block level.

Contemporary tools typically employ a modular architecture that decouples the language-specific parsing, analysis, and unparsing from the language-independent differentiation algorithms. The first attempt at such an architecture was based on the Automatic Differentiation Intermediate Form (Bischof and Roh, 1996), used by ADIC version 1.1 and ADIFOR version 3 to share a Hessian module for computing second derivatives (Abate et al., 1997). The modular architecture facilitated experimentation with a variety of differentiation algorithms, an important capability because the best algorithm was not known a priori. However, the AIF representation suffered from many limitations, including a poorly defined syntax and no mechanism for describing control flow. Its successor, the XAIF (Hovland et al., 2002), is described in Section 4.5.1. At present, the XAIF is used by research groups at Argonne National Laboratory (USA), Rice University (USA), and the University of Hertfordshire (UK). The Argonne and Rice groups have developed prototype frontends/unparsers for Fortran 90, based on Open64, and C/C++, based on EDG (Edison Design Group, 2003) and Sage 3. The Argonne and Hertfordshire groups have developed differentiation modules implementing the forward mode, an optimal statement level preaccumulation algorithm (Naumann, 2003), and various basic block level cross-country preaccumulation strategies (Naumann and Gottschling, 2003). We anticipate support for the XAIF in the TAPENADE frontend/unparser for Fortran 95 developed by

INRIA (France); the Fortran 95 frontend/backend developed by NAG, the Numerical Algorithms Group (UK); and the EliAD transformation modules developed by the University of Shrivenham (UK).

3.4. HYBRID OPERATOR-OVERLOADING, SOURCE TRANSFORMATION

Because operator overloading works at the level of individual operations (or, when expression templates are used, single statements), certain performance optimizations are not available. Preaccumulation at the statement, basic block, or subroutine level must be deferred until run time, because the structure of the computational graph is not available to the automatic differentiation tool at compile time. Heuristics for cross-country preaccumulation must be cheap or able to be amortized over many executions of the same code segment. Because static dependence analysis is not available to the automatic differentiation tool, opportunities may be missed to avoid storing intermediate function values that can be cheaply recomputed or are not needed for derivative computations.

On the other hand, source-to-source transformation may fail (or become extremely difficult) for highly modular programs. For example, deferral of template instantiation until link time, as through the `export` keyword in C++, can interfere with the source-to-source transformation strategy. Precise dataflow analysis of programs with unconstrained pointers and reversal of control flow (as required in the reverse mode) must include a runtime component. Furthermore, the development of a complete infrastructure for parsing, analyzing, and unparsing a new programming language can be extremely time consuming.

For these reasons, automatic differentiation tools traditionally based on source-to-source transformation (such as ADIC) and operator overloading (such as ADOL-C) are evolving toward a hybrid strategy that mixes (the best of) both strategies. This strategy can be interpreted as falling back to operator overloading when source-to-source transformation fails or as using source-to-source transformation to improve the performance of operator overloading. Another interpretation is that source-to-source automatic differentiation tools are a sort of domain-specific compiler for a telescoping language (Guyer and Lin, 2001; Quinlan, 2000; Kennedy et al., 2001). This interpretation suggests that the implementation of automatic differentiation tools could be simplified through the use of tools such as Broadway (Guyer and Lin, 2001), CodeBoost (Bagge et al., 2003) and ROSE (Quinlan, 2000), especially as these technologies mature.

4. Implementation of ADIC 2.0

To illustrate the implementation of a source transformation automatic differentiation tool, we describe the implementation of ADIC 2.0, a second-generation tool for ANSI-C currently under development. ADIC 2.0 uses the following steps.

1. Preprocess
2. Parse
3. Canonicalize
4. Analyze
5. Convert to XAIF
6. Transform XAIF
7. Convert from XAIF
8. Unparse to C

We describe each of these steps, with the greatest emphasis on steps 5–7, since the XAIF is an automatic differentiation-specific program representation.

4.1. PREPROCESS

The C preprocessor expands macros and handles other directives embedded in source code. Preprocessing provides flexibility and enhances portability by isolating platform-dependent functions and data structures in include files. Unfortunately, preprocessing can significantly complicate source-to-source transformation systems, since directives and macro usage are normally lost in the preprocessed source file. Thus, to maintain portability of AD-generated code, we need to retain some of the C preprocessor directives and macros embedded in the original source code whose expansions are necessary to parse the program. Tools such as ADIC achieve this by marking up the locations of system includes with no mathematically relevant functions (e.g., stdio.h). The original directives are restored during the unparsing stage. See (Bischof et al., 1997) for a more detailed description of how the C preprocessor is handled by ADIC.

4.2. PARSE AND UNPARSE

ADIC 2.0 uses version 3 of the SAGE compiler toolkit, developed as part of the ROSE project (Quinlan, 2000). SAGE 3 is based on the robust and widely used EDG C++ Front End (Edison Design Group, 2003). SAGE 3 builds an

AST and provides mechanisms for traversing and modifying the tree. SAGE 3 also provides an unparser that can be used to generate C or C++ from the modified AST.

4.3. CANONICALIZE

Just as isolation of side effects can simplify the implementation of an optimizing compiler (Allen and Kennedy, 2002, p. 614), so can it simplify the task of a source transformation system, especially one that must introduce new semantics into a program. Therefore, ADIC and other automatic differentiation tools (Bischof et al., 1992) employ a canonicalization phase. During this phase, ADIC hoists all lvalue updates that may cause side effects out of expressions. The transformations are structured so as to not change the semantic meaning of the program. Figure 5 shows an example of canonicalization to isolate side effects.

Original Code:

```
(*f(i)) *= x[i++];
```

Canonicalized Code:

```
add1 = f(i);
(*add1) = (*add1) * x[i];
i++;
```

Figure 5. Isolating side effects

4.4. ANALYZE

At present, the analysis capabilities of ADIC 2.0 are quite primitive. However, we are developing a dataflow and alias analysis infrastructure that will enable activity analysis and other types of analyses that will improve the performance of the derivative code.

4.5. CONVERSION TO AND FROM XAIF

Experience has shown that the development of algorithms that exploit the chain rule can be decoupled from the infrastructure that deals with the language and the user interface. We have recently introduced a new, XML-based intermediate format, the XAIF (Hovland et al., 2002), that can express program structure at multiple granularities, from individual assignment statements to collections of subroutines. The XAIF attempts to represent the mathematically relevant program elements in XML. This representation is not intended as a replacement for SUIF (Amarasinghe et al., 1995), WHIRL (SGI,

1999), or other intermediate formats but rather as a special-purpose notation for the development of mathematics-based transformation algorithms.

Transformation modules operate at different levels of the graph hierarchy. For example, a forward-mode module using statement-level reverse mode needs access only to the XAIF for assignment statements. Other modules may implement strategies that require basic block-level XAIF, while some reverse-mode tools may need access to control flow or call graph information. The XAIF is flexible enough to allow the independent processing of different levels of the graph hierarchy.

4.5.1. *XAIF Definition*

The XAIF representation consists of a series of nested graphs. Figure 6 shows a high-level overview of the XAIF structure. All vertex and edge elements have identifiers that are unique within the scope defined by the parent graph element. Uniqueness of vertex and edge identifiers, as well as correctness of key references, is verified automatically by most validating parsers.

At the highest level, the program is represented by a `CallGraph` element, whose children are vertices corresponding to subroutines and edges signifying subroutine calls. The `CallGraph` element contains a hierarchy of variable scopes represented as trees. Each scope can have an associated symbol table containing information on symbols defined within that scope. The root node of the scope tree contains information on global symbols, in this case, the subroutines `head` and `comp`. Symbol table entries can contain a number of attributes, such as kind (default is variable), type (default is real), and shape (default is scalar). In addition to the scope hierarchy, the call graph also contains a specification of the independent and dependent variables by referring to the arguments of the top-level subroutine.

The vertices of the call graph are control flow graphs corresponding to subroutine definitions. In the rest of this section, we focus on the XAIF representation of the `head` subroutine and its derivatives. The vertices and edges of `ControlFlowGraph` elements represent the control flow of the program. Each `ControlFlowVertex` can contain a `BasicBlock`, a `ForLoop`, an `If`, or the graph corresponding to any other statement that affects the flow of control in the computation. `ControlFlowEdge` elements represent the flow of control between code fragments encapsulated in `ControlFlowVertex` or equivalent elements. The substitution group of the `ControlFlowVertex` element consists of `BasicBlock`, `Entry`, `Exit`, `If`, `ForLoop`, `PreLoop`, and `PostLoop`.

The portions of the code that are actually augmented with derivative computations are contained within `BasicBlock` elements, which correspond to basic blocks in the code. A basic block consists of a sequence of assignment statements or subroutine calls. Each statement has a unique identifier and can optionally contain frontend-specific annotations in an `annotation` at-

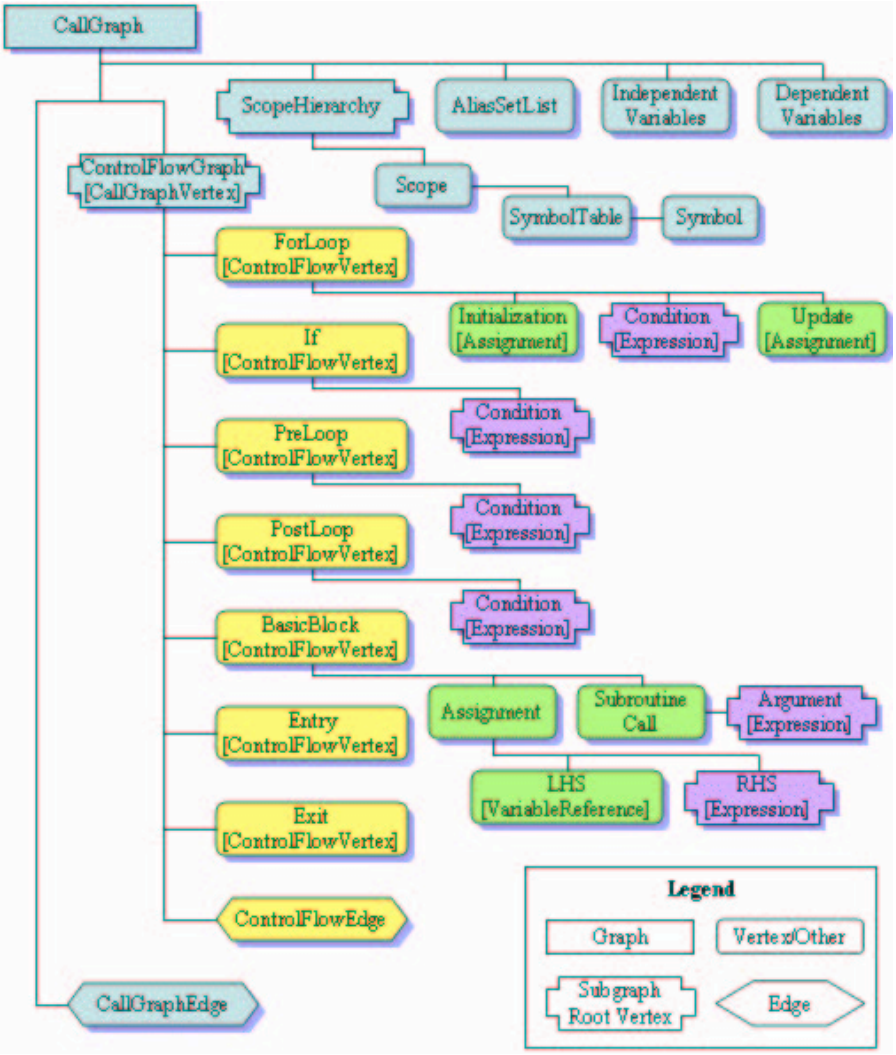

Figure 6. XAIF structure. Root vertices of key subgraphs, such as ControlFlowGraph and Expression, are outlined with a box shape different from that of other vertices.

tribute. These annotations can be used for storing information that would aid in the parsing of the XAIF and conversion to the source language. For example, in ADIC, frontend annotations are used to store the address of the AST node corresponding to the statement. When new statements are introduced by a transformation module, they are annotated with either the attribute of the originating statement or a special new attribute; this makes it possible to incorporate statements computing the derivatives in the correct location in the original AST.

Only the assignment statements containing active variables (or loop indices) are included in the XAIF as `AssignmentStatement` elements. The left-hand side of an assignment vertex is limited to a `VariableReference` (which can be used to define array references), while the right-hand side is in the equivalence class of `Expression`. Expression graph edges are annotated with a position attribute, which is used to specify operator precedence explicitly.

The representation of expressions in the `Expression` graph is straightforward, including both Boolean and arithmetic operators. `Expression` graph vertices can be variable or constant references, intrinsic operations, and subroutine calls. Intrinsic operations are subdivided into two main categories: inlinable and noninlinable. The difference between inlinable and noninlinable operations is that code computing the partials for the former can be inlined, while the code computing the derivatives of the latter requires one or more subroutine calls. The definition of the partials for these intrinsics is contained in a separate XAIF file, which generally includes all the standard intrinsic functions available in a given language.

4.5.2. *Transformation Modules*

ADIC 2.0 includes several differentiation algorithms and can interface to other modules based on the XAIF. The default transformation module implements forward mode overall with optimal statement level preaccumulation (Naumann, 2003). Other transformation modules implement reverse mode or forward mode with basic block-level preaccumulation. All transformation modules use the XAIF to communicate with the frontend/backend. Typically, a transformation module parses the XAIF, builds an internal representation of the linearized computational graph, implements an accumulation strategy in terms of the linearized computation graph, and generates XAIF corresponding to the derivative computation. The generated XAIF may include calls to a runtime library, an implementation of which must be provided for whatever language the frontend/backend supports. In the future, we anticipate the development of second derivative (Hessian) modules and a simple forward-mode module that can be used for teaching or as a foundation for more sophisticated transformations.

5. Conclusions

The need for accurate and fast derivatives for models presented as computer codes is ubiquitous in computational science. Automatic differentiation provides a mechanism for computing those derivatives accurately with minimal human effort. In this paper, we described the implementation of automatic differentiation tools in general, and the ADIC tool in particular. Current and

future research in automatic differentiation will lead to new heuristics for the combinatorial problem of how to combine partial derivatives, static and dynamic analyses to reduce storage costs for the reverse mode, new implementations for new programming languages, and new applications for the analytic derivatives that automatic differentiation can provide.

Acknowledgments

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, Office of Science, under Contract W-31-109-Eng-38. Christian Bischof's work was partially supported by the Deutsche Forschungsgemeinschaft within SFB 401 "Modulation of flow and fluid–structure interaction at airplane wings," Aachen University of Technology, Germany.

We are grateful to the anonymous reviewers, whose detailed recommendations substantially improved this article. We thank Gail Pieper and Michelle Mills Strout for their comments on drafts of this paper. We thank the many members of the automatic differentiation community whose work is reported here and inspired much of our own work.

Appendix

The full forward-mode code generated by TAPENADE for the simple example is as follows.

```
C        Generated by TAPENADE     (INRIA, Tropics team)
C  Version 2.0.6 - (Id: 1.14 vmp Stable - Thu Sep 18 08:35:47 MEST 2003)
C
C  Differentiation of foo in forward (tangent) mode: (multi-directional mode)
C   variations  of output variables: s
C   with respect to input variables: a
      SUBROUTINE FOO_DV(s, sd, a, ad, n, nbdirs)
      INCLUDE 'DIFFSIZES.inc'
C  Hint: NBDirsMax should be the maximum number of differentiation directions
      INTEGER n, nbdirs
      DOUBLE PRECISION a(n), ad(NBDirsMax, n), s, sd(NBDirsMax)
      DOUBLE PRECISION f, fd(NBDirsMax), g, gd(NBDirsMax)
      INTEGER i, nd
      INTRINSIC SQRT
C
C
      g = 0
      DO nd=1,nbdirs
        gd(nd) = 0.D0
      ENDDO
      DO i=1,n
        DO nd=1,nbdirs
```

```
          gd(nd) = gd(nd) + ad(nd, i)*a(i) + a(i)*ad(nd, i)
        ENDDO
        g = g + a(i)*a(i)
      ENDDO
      DO nd=1,nbdirs
        IF (gd(nd) .EQ. 0.0) THEN
          gd(nd) = 0.D0
        ELSE
          gd(nd) = gd(nd)/(2.0*SQRT(g))
        END IF
      ENDDO
      g = SQRT(g)
C
      s = 0
      DO nd=1,nbdirs
        sd(nd) = 0.D0
      ENDDO
      DO i=1,n
        CALL FUNC_DV(f, fd, a(i), ad(1, i), g, gd, nbdirs)
        DO nd=1,nbdirs
          sd(nd) = sd(nd) + fd(nd)
        ENDDO
        s = s + f
      ENDDO
C
      RETURN
      END

C  Differentiation of func in forward (tangent) mode: (multi-directional mode)
C   variations  of output variables: f
C   with respect to input variables: x y
C
      SUBROUTINE FUNC_DV(f, fd, x, xd, y, yd, nbdirs)
      INCLUDE 'DIFFSIZES.inc'
C  Hint: NBDirsMax should be the maximum number of differentiation directions
      INTEGER nbdirs
      DOUBLE PRECISION f, fd(NBDirsMax), x, xd(NBDirsMax), y, yd(
     +                 NBDirsMax)
      DOUBLE PRECISION a, ad(NBDirsMax), arg1, arg1d(NBDirsMax), b, bd(
     +                 NBDirsMax)
      INTEGER nd
      INTRINSIC EXP, SIN, LOG
C
C
      IF (x .GT. y) THEN
        a = SIN(y)
        arg1 = x - y
        DO nd=1,nbdirs
          arg1d(nd) = xd(nd) - yd(nd)
          ad(nd) = yd(nd)*COS(y)
          bd(nd) = xd(nd)*EXP(arg1)/x + LOG(x)*arg1d(nd)*EXP(arg1)
        ENDDO
        b = LOG(x)*EXP(arg1)
      ELSE
        DO nd=1,nbdirs
          ad(nd) = ((xd(nd)*SIN(x)+x*xd(nd)*COS(x))*y-x*SIN(x)*yd(nd))/y
     +      **2
          bd(nd) = yd(nd)/y
```

```
        ENDDO
        a = x*SIN(x)/y
        b = LOG(y)
      END IF
      arg1 = a*b
      DO nd=1,nbdirs
        arg1d(nd) = ad(nd)*b + a*bd(nd)
        fd(nd) = arg1d(nd)*EXP(arg1)
      ENDDO
      f = EXP(arg1)
C
      RETURN
      END
```

The full reverse-mode code generated by TAPENADE for the simple example is as follows.

```
C        Generated by TAPENADE     (INRIA, Tropics team)
C  Version 2.0.6 - (Id: 1.14 vmp Stable - Thu Sep 18 08:35:47 MEST 2003)
C
C  Differentiation of foo in reverse (adjoint) mode:
C   gradient, with respect to input variables: s a
C   of linear combination of output variables: s
      SUBROUTINE FOO_B(s, sb, a, ab, n)
      INTEGER n
      DOUBLE PRECISION a(n), ab(n), s, sb
      INTEGER adto, i, ii1
      DOUBLE PRECISION f, fb, g, gb
      INTRINSIC SQRT
C
C
      g = 0
      DO i=1,n
        g = g + a(i)*a(i)
      ENDDO
      CALL PUSHINTEGER4(i - 1)
      CALL PUSHREAL8(g)
      g = SQRT(g)
C
      s = 0
      DO i=1,n
        CALL PUSHREAL8(g)
        CALL FUNC(f, a(i), g)
        s = s + f
      ENDDO
      CALL PUSHINTEGER4(i - 1)
      DO ii1=1,n
        ab(ii1) = 0.D0
      ENDDO
      gb = 0.D0
      CALL POPINTEGER4(adTo)
      DO i=adTo,1,-1
        fb = sb
        CALL POPREAL8(g)
        CALL FUNC_B(f, fb, a(i), ab(i), g, gb)
      ENDDO
      CALL POPREAL8(g)
      gb = gb/(2.0*SQRT(g))
      CALL POPINTEGER4(adTo)
```

```
      DO i=adTo,1,-1
        ab(i) = ab(i) + (a(i)+a(i))*gb
      ENDDO
      sb = 0.D0
      END

C  Differentiation of func in reverse (adjoint) mode:
C   gradient, with respect to input variables: x y
C   of linear combination of output variables: f x y
C
      SUBROUTINE FUNC_B(f, fb, x, xb, y, yb)
      DOUBLE PRECISION f, fb, x, xb, y, yb
      DOUBLE PRECISION a, ab, arg1, arg1b, b, bb
      INTEGER branch
      INTRINSIC EXP, SIN, LOG
C
C
      IF (x .GT. y) THEN
        a = SIN(y)
        arg1 = x - y
        b = LOG(x)*EXP(arg1)
        CALL PUSHINTEGER4(0)
      ELSE
        a = x*SIN(x)/y
        b = LOG(y)
        CALL PUSHINTEGER4(1)
      END IF
      CALL PUSHREAL8(arg1)
      arg1 = a*b
      f = EXP(arg1)
      arg1b = EXP(arg1)*fb
      CALL POPREAL8(arg1)
      ab = b*arg1b
      bb = a*arg1b
      CALL POPINTEGER4(branch)
      IF (branch .LT. 1) THEN
        arg1b = LOG(x)*EXP(arg1)*bb
        xb = xb + EXP(arg1)*bb/x + arg1b
        yb = yb + COS(y)*ab - arg1b
      ELSE
        yb = yb + bb/y - x*SIN(x)*ab/y**2
        xb = xb + (x*COS(x)/y+SIN(x)/y)*ab
      END IF
      END
```

References

Abate, J., C. Bischof, A. Carle, and L. Roh: 1997, 'Algorithms and Design for a Second-Order Automatic Differentiation Module'. In: *Proc. Int. Symposium on Symbolic and Algebraic Computing (ISSAC) '97*. New York, pp. 149–155, Association of Computing Machinery.

Aho, A. V., R. Sethi, and J. D. Ullman: 1986, *Compilers: Principles, Techniques and Tools*. Reading, MA: Addison-Wesley.

Allen, R. and K. Kennedy: 2002, *Optimizing Compilers for Modern Architectures: a Dependence-based Approach*. San Mateo, CA: Morgan Kaufmann Publishers.

Amarasinghe, S. P., J. M. Anderson, M. S. Lam, and C. W. Tseng: 1995, 'The SUIF Compiler for Scalable Parallel Machines'. In: *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*.

Aubert, P. and N. Di Césaré: 2001, 'Expression Templates and Forward Mode Automatic Differentiation'. In (Corliss et al., 2001), Chapt. 37, pp. 311–315.

Bagge, O. S., K. T. Kalleberg, M. Haveraaen, and E. Visser: 2003, 'Design of the CodeBoost Transformation System for Domain-Specific Optimisation of C++ Programs'. In: D. Binkley and P. Tonella (eds.): *Third International Workshop on Source Code Analysis and Manipulation (SCAM 2003)*. Amsterdam, The Netherlands, IEEE Computer Society Press. (To appear).

Bartholomew-Biggs, M.: 1995, 'OPFAD - A Users Guide to the OPtima Forward Automatic Differentiation Tool'. Technical report, Numerical Optimization Centre, University of Hertfordsshire.

Bartholomew-Biggs, M. C., S. Brown, B. Christianson, and L. C. W. Dixon: 1995, 'The Efficient Calculation of Gradients, Jacobians and Hessians'. Technical Report NOC TR301, The Numerical Optimisation Center, University of Hertfordshire, Hatfield, U.K.

Barton, J. J. and L. R. Nackman: 1996, 'Automatic Differentiation'. *C++ Report* **8**(2), 61–63.

Beda, L. M., L. N. Korolev, N. V. Sukkikh, and T. S. Frolova: 1959, 'Programs for automatic differentiation for the machine BESM'. Technical Report, Institute for Precise Mechanics and Computation Techniques, Academy of Science, Moscow, USSR. (In Russian).

Bell, B. M.: 2003, 'CppAD User Manual'. Available at http://www.seanet.com/ bradbell/CppAD/.

Bendtsen, C. and O. Stauning: 1996, 'FADBAD, a Flexible C++ Package for Automatic Differentiation'. Technical Report IMM–REP–1996–17, Department of Mathematical Modelling, Technical University of Denmark, Lyngby, Denmark.

Berz, M., C. Bischof, G. Corliss, and A. Griewank (eds.): 1996, *Computational Differentiation: Techniques, Applications, and Tools*. Philadelphia, PA: SIAM.

Binkley, D. W. and K. B. Gallagher: 1996, 'Program Slicing'. *Advances in Computers* **43**, 1–50.

Bischof, C., A. Carle, G. Corliss, A. Griewank, and P. Hovland: 1992, 'ADIFOR: Generating Derivative Codes from Fortran Programs'. *Scientific Programming* **1**(1), 11–29.

Bischof, C., A. Carle, P. Khademi, and A. Mauer: 1996, 'ADIFOR 2.0: Automatic Differentiation of Fortran 77 Programs'. *IEEE Computational Science & Engineering* **3**(3), 18–32.

Bischof, C. and L. Roh: 1996, 'The Automatic Differentiation Intermediate Form (AIF)'. Unpublished Information.

Bischof, C., L. Roh, and A. Mauer: 1997, 'ADIC — An Extensible Automatic Differentiation Tool for ANSI-C'. *Software–Practice and Experience* **27**(12), 1427–1456.

Bischof, C. H., H. M. Bücker, B. Lang, and A. Rasch: 2001, 'An Interactive Environment for Supporting the Paradigm Shift from Simulation to Optimization'. In: *4th Workshop on Parallel/High-Performance OO Scientific Computing (POOSC'01) 14 October 2001, at the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01) 14–18 October, Tampa Bay, FL.* To appear.

Bischof, C. H., H. M. Bücker, B. Lang, A. Rasch, and A. Vehreschild: 2002, 'Combining Source Transformation and Operator Overloading Techniques to Compute Derivatives for MATLAB Programs'. Preprint RWTH-CS-SC-02-04, Institute for Scientific Computing, Aachen University of Technology.

Brooke, A., D. Kendrick, and A. Meeraus: 1988, *GAMS: A User's Guide*. South San Francisco, CA: The Scientific Press.

Brown, S.: 1995, 'OPRAD - A Users Guide to the OPtima Reverse Automatic Differentiation Tool'. Technical report, Numerical Optimization Centre, University of Hertfordsshire.

Christianson, B., L. C. W. Dixon, and S. Brown: 1996, 'Sharing Storage Using Dirty Vectors'. In (Berz et al., 1996), pp. 107–115.

Christianson, D. B.: 1991, 'Automatic Hessians by Reverse Accumulation in Ada'. *IMA J. on Numerical Analysis*. Presented at SIAM Workshop on Automatic Differentiation of Algorithms, Breckenridge, CO, January 1991.

Coleman, T. F. and A. Verma: 2000, 'ADMIT-1: Automatic Differentiation and MATLAB Interface Toolbox'. *ACM Trans. Math. Softw.* **26**(1), 150–175.

Corliss, G., C. Faure, A. Griewank, L. Hascoët, and U. Naumann (eds.): 2001, *Automatic Differentiation: From Simulation to Optimization*, Computer and Information Science. New York, NY: Springer.

Edison Design Group: 2003, 'EDG C++ Front End'. www.edg.com/cpp.html.

Faure, C.: 2001, 'Adjoining Strategies for Multi-Layered Programs'. *Optimisation Methods and Software*. To appear. Also appeared as INRIA Rapport de recherche no. 3781, BP 105-78153 Le Chesnay Cedex, FRANCE, 1999.

Faure, C. and U. Naumann: 2001, 'Minimizing the Tape Size'. In (Corliss et al., 2001), Chapt. 34, pp. 293–298.

Forth, S.: 2001, 'An efficient implementation of AD in MATLAB'. Presentation at Joint University of Hertfordshire/Cranfield University (RMCS Shrivenham) Automatic Differentiation Symposium. Available at http://www.rmcs.cranfield.ac.uk/esd/amor/workshop/alldatastore/ADDAYmay01forth.pdf.

Fourer, R., D. M. Gay, and B. W. Kernighan: 1993, *AMPL: A Modeling Language for Mathematical Programming*. South San Francisco, CA: The Scientific Press.

Frazier, Z.: 2003, 'PyAD User manual'. Available at http://students.washington.edu/zfrazier/-projects/pyad/pyad-doc/.

Giering, R. and T. Kaminski: 1998, 'Recipies for Adjoint Code Construction'. *ACM TOMS* **24**(4), 437–474.

Giering, R. and T. Kaminski: 2002, 'Applying TAF to Generate Efficient Derivative Code of Fortran 77-95 programs'. In: *Proceedings of GAMM 2002, Augsburg, Germany*.

Griewank, A.: 1989, 'On Automatic Differentiation'. In: *Mathematical Programming: Recent Developments and Applications*. Amsterdam, pp. 83–108, Kluwer Academic Publishers.

Griewank, A.: 1998. Personal communication.

Griewank, A.: 2000, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Philadelphia, PA: SIAM.

Griewank, A., D. Juedes, and J. Utke: 1996, 'ADOL-C, A Package for the Automatic Differentiation of Algorithms Written in C/C++'. *ACM Transactions on Mathematical Software* **22**(2), 131–167.

Griewank, A.: 1992, 'Achieving Logarithmic Growth of Temporal and Spatial Complexity in Reverse Automatic Differentiation'. *Optimization Methods and Software* **1**, 35–54.

Griewank, A. and G. F. Corliss (eds.): 1991, *Automatic Differentiation of Algorithms: Theory, Implementation, and Application*. Philadelphia, PA: SIAM.

Griewank, A. and S. Reese: 1991, 'On the Calculation of Jacobian Matrices by the Markowitz Rule'. In (Griewank and Corliss, 1991), pp. 126–135.

Grimm, J., L. Pottier, and N. Rostaing-Schmidt: 1996, 'Optimal Time and Minimum Space-Time Product for Reversing a Certain Class of Programs'. In (Berz et al., 1996), pp. 95–106.

Guyer, S. Z. and C. Lin: 2001, 'Optimizing the Use of High Performance Software Libraries'. *Lecture Notes in Computer Science* **2017**, 227–243.

Hill, D. R. and L. C. Rich: 1992, 'Automatic Differentiation in MATLAB'. *Applied Numerical Mathematics* **9**, 33–43.

Hinkins, R. L.: 1994, 'Parallel Computation of Automatic Differentiation Applied to Magnetic Field Calculations'. Master's thesis, University of California, Berkeley, CA.

Hinsen, K.: 2003, 'Scientific Python collection'. Module Scientific.Functions.Derivatives. Available at http://starship.python.net/~hinsen/ScientificPython/.

Hovland, P. D., U. Naumann, and B. Norris: 2002, 'An XML-Based Platform for Semantic Transformation of Numerical Programs'. Preprint ANL/MCS-P950-0402, Mathematics and Computer Science Division, Argonne National Laboratory. To appear in Proceedings of Software Engineering and Applications (SEA 2002).

Huss, R. E.: 1990, 'An ADA Library for Automatic Evaluation of Derivatives'. *Applied Mathematics and Computation* **35**(2), 103–123.

I. Tsukanov, M. H.: 2003, 'Data Structure and Algorithms for Fast Automatic Differentiation'. *International Journal for Numerical Methods in Engineering* **56**(13), 1949–1972.

Jerrell, M.: 1989, 'Automatic Differentiation Using Almost Any Language'. *ACM SIGNUM Newsletter* pp. 2–9.

Juedes, D. W.: 1991, 'A Taxonomy of Automatic Differentiation Tools'. In (Griewank and Corliss, 1991), pp. 315–329.

Kahrimanian, H. G.: 1953, 'Analytical Differentiation by a Digital Computer'. Master's thesis, Temple University.

Kalman, D. and R. Lindell: 1991, 'Automatic Differentiation in Astrodynamical Modeling'. In (Griewank and Corliss, 1991), pp. 228–243.

Karczmarczuk, J.: 2001, 'Functional Differentiation of Computer Programs'. *Journal of HOSC* **14**, 35–57.

Kennedy, K., B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon: 2001, 'Telescoping Languages: A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries'. *Journal of Parallel and Distributed Computing* **61**(12), 1803–1826.

Korelc, J.: 2001, 'Hybrid System for Multi-Language and Multi-Environment Generation of Numerical Codes'. In: *Proceedings of the 2001 International Symposium on Symbolic and Algebraic Computation*. pp. 209–216, ACM Press.

Lawson, C. L.: 1971, 'Computing Derivatives Using W-Arithmetic and U-Arithmetic'. Internal Computing Memorandum CM–286, Jet Propulsion Laboratory, Pasadena, CA.

Lesk, A. M.: 1967, 'Dynamic computation of derivatives'. *Communications of the ACM* **10**(9), 571–572.

Maany, Z.: 1989, 'Ada Automatic Differentiation Package for the Optimization of Functions of Many Variables'. Technical Report NOC TR209, The Numerical Optimisation Center, Hatfield Polytechnic, Hatfield, U.K.

Martins, J. R. R. A., I. M. Kroo, and J. J. Alonso: 2000, 'An Automated Method for Sensitivity Analysis Using Complex Variables'. In: *Proceedings of the 38th Aerospace Sciences Meeting, Reno, NV*.

Martins, J. R. R. A., P. Sturdza, and J. J. Alonso: 2001, 'The Connection Between the Complex-Step Derivative Approximation and Algorithmic Differentiation'. In: *Proceedings of the 39th Aerospace Sciences Meeting, Reno, NV*. Complexify.h and derivify.h available at http://mdolab.utias.utoronto.ca/c++.html.

Michelotti, L.: 1991, 'MXYZPTLK: A C++ Hacker's Implementation of Automatic Differentiation'. In (Griewank and Corliss, 1991), pp. 218–227. Software available at http://www.netlib.org/c++/mxyzptlk/.

Monagan, M. and R. R. Rodoni: 1996, 'An Implementation of the Forward and Reverse Mode of Automatic Differentiation in Maple'. In: M. Berz, C. Bischof, G. Corliss, and A. Griewank (eds.): *Computational Differentiation: Techniques, Applications, and Tools*. Philadelphia, PA: SIAM, pp. 353–362.

Muchnick, S. S.: 1997, *Advanced Compiler Design and Implementation*. San Mateo, CA: Morgan Kaufmann Publishers.

NAG-AD: 2003, 'Differentiation Enabled Fortran Compiler Technology'. http://-www.nag.co.uk/nagware/research/ad_overview.asp.

Naumann, U.: 2002, 'Reducing the Memory Requirement in Reverse Mode Automatic Differentiation by Solving TBR Flow Equations'. In: P. M. A. Sloot, C. J. K. Tan, J. J. Dongarra, and A. G. Hoekstra (eds.): *Proceedings of the International Conference on Computational Science, Amsterdam, The Netherlands, April 21–24, 2002. Part II*, Vol. 2330 of *Lecture Notes in Computer Science*. Berlin, pp. 1039–1048, Springer.

Naumann, U.: 2003, 'Statement Level Optimality of Tangent-Linear and Adjoint Models'. Preprint ANL-MCS/P1066-0603, Argonne National Laboratory.

Naumann, U. and P. Gottschling: 2003, 'Simulated Annealing for Optimal Pivot Selection in Jacobian Accumulation'. In: A. Albrecht (ed.): *Stochastic Algorithms, Foundations and Applications – SAGA'03*. Berlin, Springer. To appear. See also http://angellib.sourceforge.net/.

Neidinger, R. D.: 1989, 'Automatic Differentiation and APL'. *College Mathematics J.* **20**(3), 238–251.

Nilsson, H.: 2003, 'Functional Automatic Differentiation with Dirac Impulses'. *ACM SIGPLAN Notices* **38**(9), 153–164.

Nolan, J. F.: 1953, 'Analytical Differentiation on a Digital Computer'. Master's thesis, Massachusetts Institute of Technology.

Pryce, J. D. and J. K. Reid: 1998, 'ADO1, a Fortran 90 Code for Automatic Differentiation'. Technical Report RAL-TR-1998-057, Rutherford Appleton Laboratory, Chilton, Didcot, Oxfordshire, OX11 OQX, England.

Quinlan, D.: 2000, 'ROSE: Compiler Support for Object-Oriented Frameworks'. *Parallel Processing Letters* **10**(2/3), 215–??

Reps, T. and G. Rosay: 1995, 'Precise Interprocedural Chopping'. In: G. E. Kaiser (ed.): *SIGSOFT'95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*. pp. 41–52, ACM Press.

Restrepo, J. M., G. K. Leaf, and A. Griewank: 1998, 'Circumventing Storage Limitations in Variational Data Assimilation'. *SIAM Journal on Scientific Computing* **19**, 1586–1605.

Rostaing, N., S. Dalmas, and A. Galligo: 1993, 'Automatic Differentiation in Odyssee'. *Tellus* **45a**(5), 558–568.

Rote, G.: 1990, 'Path Problems in Graphs'. In: G. Tinhofer, E. Mayr, H. Noltemeier, and M. M. S. in cooperation with R. Albrecht (eds.): *Computational Graphs Theory, Springer-Verlag Computing Supplementum 7*. Springer.

SGI: 1999, 'WHIRL Intermediate Language Specification'. Available at http://open64.sourceforge.net/documentation.html.

Stamatiadis, S., R. Prosmiti, and S. C. Farantos: 2000, 'AUTO_DERIV: Tool for Automatic Differentiation of a FORTRAN Code'. *Comput. Phys. Commun.* **127**(2&3), 343–355. Catalog number: ADLS.

Tadjouddine, M., S. A. Forth, and J. D. Pryce: 2003, 'Hierarchical Automatic Differentiation by Vertex Elimination and Source Transformation'. In: V. Kumar, M. L. Gavrilova, C. J. K. Tan, and P. L'Ecuyer (eds.): *Proceedings of the International Conference on Computational Science and its Applications, Montreal, Canada, May 18–21, 2003. Part II*, Vol. 2668 of *Lecture Notes in Computer Science*. Berlin, pp. 95–104, Springer.

TAPENADE: 2002, 'TAPENADE Tutorial'. http://www-sop.inria.fr/tropics/tapenade/-tutorial.html.

Veldhuizen, T.: 1995, 'Expression Templates'. *C++ Report* **7**(5), 26–31.

Wengert, R. E.: 1964, 'A Simple Automatic Derivative Evaluation Program'. *Comm. ACM* **7**(8), 463–464.

Wilkins, R. D.: 1964, 'Investigation of a New Analytic Model for Numerical Derivative Evaluation'. *Commun. ACM* **7**(8), 465–471.