

# Hybrid Differentiation Strategies for Simulation and Analysis of Applications in C++

ROSCOE A. BARTLETT

BART G. VAN BLOEMEN WAANDERS

Sandia National Laboratories, Albuquerque NM 87185 USA

MARTIN BERGGREN

Department of Information Technology

Uppsala University, Sweden

---

Computationally efficient and accurate derivatives are important to the success of many different types of numerical methods. Automatic differentiation (AD) approaches compute truncation-free derivatives and can be efficient in many cases. Although present AD tools can provide a convenient implementation mechanism, the computational efficiency rarely compares to analytically derived versions that have been carefully implemented. The focus of this work is to combine the strength of these methods into a hybrid strategy that attempts to achieve an optimal balance of implementation and computational efficiency by selecting the appropriate components of the target algorithms for AD and analytical derivation. Although several AD approaches can be considered, our focus is on the use of template overloading forward AD tools in C++ applications. We demonstrate this hybrid strategy for a system of partial differential equations in gas dynamics. These methods apply however to other systems of differentiable equations, including DAEs and ODEs.

Categories and Subject Descriptors: G.1.0 [Numerical analysis]: General; G.1.4 [Numerical Analysis]: Quadratic and Numerical Differentiation - Automatic Differentiation; D.1.0 [Programming Techniques]: General; G.4 [Mathematical Software]: Efficiency

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Hybrid Differentiation Methods, Euler equations, automatic differentiation, finite volume methods, template overloading

---

## 1. INTRODUCTION

The derivative calculation represents a fundamental component in simulation and analysis codes. Although the differentiation of elementary operations is straightforward, multivariate derivative calculations in complicated algorithms can be time consuming to analytically derive and error-prone to implement. There are many approaches available to implement derivative calculations, including symbolic methods, finite differences, complex step, and automatic differentiation (AD). An appropriate calculation strategy depends on implementation issues, accuracy requirements, and the importance of computational efficiency.

---

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under Contract DE-AC04-94AL85000.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.

© 2007 ACM 0098-3500/2007/1200-0001 \$5.00

Symbolically derived analytical code can provide the most efficient results, but can be time-consuming to derive, especially for codes simulating complex physics. Furthermore, the implementation can be difficult and error prone. Finite-difference approximations are simpler to implement but result in less accurate derivative values. The selection of the perturbation step size is one of the fundamental problems associated with finite differencing and is difficult to perform a priori. The complex step approach is a potentially implementation-efficient approach with results accurate to machine precision in most cases. Even though the implementation is relatively straightforward, the disadvantage of adding a complex data type is in the redundancy of certain computations.

Automatic differentiation can potentially provide the optimal combination of accuracy, implementation simplicity, and computational efficiency. Significant work has been done on tools that use source transformation [Bischof et al. 1997], [Bischof et al. 1992], [Giering and Kaminski 1998], [Faure 2005], [Hascoet 2004] and these tools can be very effective as a general or initial approach to calculating derivatives. The standard source transformation tools are somewhat sensitive to simulation code implementations and can make for a cumbersome development environment because of separate recompilations to calculate the appropriate derivative values of new functions. An alternative strategy for applying AD in C++ based codes is to template the functions on the scalar type which can be instantiated on `double` or `Fad<double>` for instance. The template instantiation strategy provides a mechanism to implement AD at various levels of complexity. This removes certain code maintenance issues, provides machine precision derivative calculations, and most importantly provides an easy mechanism to control the level of intrusiveness of the AD calculation, which has implications to implementation effort and computational efficiency.

Even the most efficient AD implementations can not be faster than codes using optimized symbolically derived expressions. For large and complex simulation software a balance must be achieved between implementation, computational efficiency and accuracy. Our strategy is to consider a hybrid methodology driven by the complexities associated with symbolic derivations. The subject of this paper is the use of hybrid strategies involving the application of AD at various levels of functional complexity. We present the application of hybrid approaches to a relatively complicated function evaluation that discretizes systems of partial differentiable equations (PDEs). However, other models can also be considered such as solving differential-algebraic equations (DAEs) based network simulators. Although we demonstrate certain derivative calculation strategies using a specific example from compressible fluid dynamics, most of the statements and conclusions presented in this work are general and can be applied to a range of functions and numerical algorithms. The primary contribution of this work is the development of hybrid strategies that combine automatic and symbolic differentiation for complex functions to optimize the trade-off between implementation effort and the need for computational efficiency.

The remainder of this paper is organized as follows. Background information for various methods to differentiate ANSI C++ code is provided in section 2. Section 3 presents an overview of a large class of application areas where large-scale functions are assembled from a set of mostly independent “element” computations. This section uses the term “element” in the broadest sense and can be applied to PDE simulators as well as other types of models. A series of different levels of hybrid symbolic/AD methods for differentiation is defined. Section 4 provides a particular example using compressible flow equations and presents numerical results that compare and contrast many of the different differentiation approaches. Finally, in Section 5 we offer a number of conclusions and observations.

## 2. BACKGROUND

We focus on the differentiation of vector functions of the form

$$f(x) \in \mathbf{R}^n \rightarrow \mathbf{R}^m \tag{1}$$

where it is assumed that  $f(x)$  is at least once continuously differentiable. Many different types of numerical algorithms, such as linear solvers, nonlinear solvers, and optimization methods, require the application of the Jacobian-vector product

$$\delta f = \frac{\partial f}{\partial x} \delta x \quad (2)$$

evaluated at a point  $x$  where  $\delta x \in \mathbf{R}^n$  is an arbitrary vector. The application of the linear operator in (2) is required, for instance, in an iterative Krylov-Newton method for the solution of nonlinear equations of the form  $f(x) = 0$  where  $n = m$ . The basic linear operator in (2) can be used directly as the operator application in a Krylov linear solver, such as GMRES, or can be used to generate an explicit matrix given the structure of the function  $f(x)$ . We generalize (2) to a form that involves the multiple simultaneous application of this linear operator,

$$U = \frac{\partial f}{\partial x} S \quad (3)$$

where  $S \in \mathbf{R}^{n \times p}$  and  $U \in \mathbf{R}^{m \times p}$ . Note that (3) can be used to generate the entire Jacobian matrix  $\partial f / \partial x$  itself when  $S$  is the  $n$ -by- $n$  identity matrix.

There are a variety of methods that can be used to compute the linear operator in (2) for vector functions implemented in ANSI C++: 1) hand-coded symbolic derivatives (SD), 2) approximation by finite differences (FD), 3) complex-step (CS) [Martins et al. 2003], and 4) automatic (or algorithmic) differentiation (AD) [Griewank 2000]. The main focus of this work will be on the use of operator overloading methods for AD, but first other methods are reviewed so that the advantages of hybrid approaches can be fully appreciated.

The first method is referred to as symbolic differentiation (**SD**) and is based on analytically deriving (2). The derivative expressions can either be derived and simplified by hand or by using tools such as Maple<sup>1</sup> or Mathematica<sup>2</sup>. This approach can yield very accurate and efficient derivative computations but can require a tremendous amount of manual labor and can lead to implementation errors that can degrade or destroy numerical performance [Vanden and Orkwis 1996]. Even derivatives of moderately complicated functions can be difficult to implement so that good computer performance is achieved. However, because SD results in accurate derivatives and potentially may be implemented in a computational efficient manner, this approach can be used in combination with automatic differentiation methods to produce excellent results, as discussed in later sections.

The second (and perhaps the most popular) method to compute an approximation to (2) is the use of finite differencing (**FD**) of  $x \rightarrow f(x)$ . A one-sided first-order finite difference approximation to (2) at a point  $x$  is given by

$$\delta f \approx \frac{f(x + \epsilon \delta x) - f(x)}{\epsilon} \quad (4)$$

where  $\epsilon \in \mathbf{R}$  is the finite difference step length that should be selected to approximately minimize the sum of  $O(\epsilon)$  truncation errors and roundoff cancellation errors. This approach requires minimal implementation effort because it depends only on the original function  $x \rightarrow f(x)$ , the evaluation code, a single vector function evaluation  $f(x + \epsilon \delta x)$ , and several additional simple floating point operations. As a consequence of this simplicity, it is also a computationally efficient calculation. Higher-order finite-difference approximations can be used to reduce the truncation error. This allows larger finite difference step sizes  $\epsilon$  to decrease roundoff error and thereby reduces the overall approximation error but at greater

<sup>1</sup>Maple: <http://www.maplesoft.com>

<sup>2</sup>Mathematica: <http://www.wolfram.com>

computational expense. For example, the fourth-order central finite difference approximation

$$\delta f \approx \frac{f(x - 2\epsilon\delta x) - 8f(x - \epsilon\delta x) + 8f(x + \epsilon\delta x) - f(x + 2\epsilon\delta x)}{12\epsilon} \quad (5)$$

can be applied, yielding  $O(\epsilon^4)$  truncation errors but at the cost of four evaluations of  $x \rightarrow f(x)$ . The disadvantages of finite-difference approaches are that it is difficult to select a priori the optimal finite difference step length  $\epsilon$  such that the sum of truncation and roundoff errors are adequately minimized and more accurate approximations, such as (5), significantly increase the computational cost. In general, the unavoidable errors associated with finite-difference approaches can result in poorly performing numerical methods.

A third approach to differentiate  $x \rightarrow f(x)$  is the complex-step (CS) method [Squire and Trapp 1998]. This method relies on the concept of *analytic extensions*, through which the initial real  $f$  is extended into the complex plane in a neighborhood of  $x$ . The properties of analytic functions allows the approximation

$$\delta f \approx \frac{\text{Im}[f(x + i\epsilon\delta x)]}{\epsilon}, \quad (6)$$

where  $f$  stands for the extended function and  $\text{Im}$  denotes the imaginary value. Note that there is no subtraction involved in this numerical approximation and thus the calculation suffers no loss of significant digits in finite precision arithmetics. Approximation (6) requires  $f$  to be real analytic in the complex plane at  $(x, 0)$ , which is the case for most floating-point operations with a few important exceptions, as discussed below.

By replacing the floating-point real data type `double` with a floating-point complex data type using a type similar to `std::complex<double>` and using a very small  $\epsilon$  (e.g.  $\epsilon = 10^{-20}$ ) in equation (6), accurate derivatives can be calculated, free from the classical cancellation effects. However, there are disadvantages associated with the use of the complex step method. First, complex arithmetics are significantly more expensive than real arithmetics as the result of the additional operations needed in complex computations. Second, the technique requires all operations involved in calculating  $f(x)$  to be real analytic. For instance, the absolute value function  $|z| = \sqrt{a^2 + b^2}$  for a complex number  $z = a + ib$  is not analytic nor the analytic extension of the real absolute value. The analytic extension is  $\sqrt{z^2}$ , using the principal branch of the square root. Another complex extension of the real absolute value that is not the analytic one but that also gives the correct result using formula (6), is  $\text{abs}(a + ib) = a + ib$  if  $a \geq 0$  and  $\text{abs}(a + ib) = -(a + ib)$  if  $a < 0$ . Third, the relational operators, such as `<` and `>`, are typically not defined for complex numbers. For these reasons an existing C++ complex data type designed for complex arithmetic, such as the standard C++ data type `std::complex<>`, cannot be directly used for the CS method without making modifications to the underlying C++ code. The alternative, as advocated in [Martins et al. 2003], is to define a new C++ complex data type that properly defines the relational operators `<` and `>` and the absolute value function. The disadvantages of this approach is that an existing, possibly optimized, C++ complex data type cannot be used for the purpose of differentiation. The CS method, however, is conceptually similar to operator-overloading based automatic differentiation and regardless of the above described limitations, can be utilized as a simple verification of automatic differentiation calculations.

The fourth method for evaluating (2) is automatic differentiation (AD) [Griewank 2000], also referred to as algorithmic differentiation. Considerable advancements have been made since the initial development of code lists in the late 1950s and the early 1960s [Beda et al. 1959; Moore 1979; Wengert 1964]. Source transformation tools such as ADIFOR [Bischof et al. 1992] and ADIC [Bischof et al. 1997] have been the primary focus for FORTRAN and C based codes. More recently, operator overloading methods using expression templates have been effectively applied to C++ codes [Cesare and

Pironneau 2000]. The AD methodology mechanically applies the chain rule to all arithmetic operations used within a function and exploits the fact that computer code is based on a sequence of elementary arithmetic operations. By mechanically applying the chain rule to these basic operations within a function and applying the primitive rules for differentiation (e.g.  $(a + b)' = a' + b'$ ,  $(a - b)' = a' - b'$ ,  $(ab)' = a'b + ab'$ ,  $(a/b)' = (a'b - ab')/(b^2)$ ,  $\sin(a)' = \cos(a)a'$  etc.) accurate derivatives can be calculated. Consequently, these methods are free of truncation and roundoff errors that plague FD methods. Only standard floating point roundoff errors are present in AD. In fact, it can be shown that the roundoff errors in the AD derivative computations are bounded by the roundoff errors involved in computing  $x \rightarrow f(x)$  [Griewank 2000]. Therefore, a simple way to ensure that the AD derivatives are accurate is to ensure that the function evaluation is accurate.

AD can be performed in forward or reverse mode. The forward mode is the easiest to understand and is the most efficient for computing derivatives of the form (2). The general process consist of decomposing the function into elemental steps, applying simple derivative rules to individual operations and then using the chain rule to provide a final derivative calculation of the overall function. The reverse mode of automatic differentiation was introduced by Linnainmaa [1976] and later further exploited by Speelpenning [1980] and Courty et al. [2003]. Our efforts have focused on the forward mode of AD and therefore we do not consider reverse mode AD any further here. The forward mode of AD computes both the function value and Jacobian-vector products of the form (3) for one or more input or seed vectors. More specifically, forward-mode AD performs

$$(x, S) \rightarrow \left( f(x), \frac{\partial f}{\partial x} S \right) \quad (7)$$

where  $S \in \mathbf{R}^{n \times p}$  is known as the seed matrix. This form of AD only computes the function value  $f(x)$  once and uses it to propagate  $p$  different columns of sensitivities through the forward differentiation operator. The output is the Jacobian matrix  $\partial f / \partial x$  when the seed matrix  $S$  is set to  $I$ . Commonly, the Jacobian  $\partial f / \partial x$  exhibits significant sparsity when  $n$  and  $m$  are large. A considerable body of work [Griewank 2000, Chapter 7] has been devoted to developing methods that take advantage of the sparsity structure. Here we will only consider the direct use of the forward-mode of AD to explicitly form dense Jacobians when  $n$  and  $m$  are small. However, even in these smaller dimensional applications of forward-mode AD, it may still be beneficial to take advantage of the structure of  $f(x)$  when the seed matrix  $S = I$  is used to avoid derivative computations involving zeros.

There are two major approaches for implementing AD: source transformation and operator overloading. The source-transformation approach differentiates a function by parsing the code, performing various analyzes and writing a set of source files with new definitions to implement AD expressions. While the source transformation approach as been very successful and well implemented for languages such as Fortran 77/90 [Bischof et al. 1992], [Faure 2005], [Hascoet 2004], [Courty et al. 2003], as of the time of this writing, there are no such tools for differentiating ANSI/ISO C++. Therefore, we will not consider source transformation tools any further in this paper. The operator overloading approach for forward AD uses an abstract data-type to define all of the operators of a floating point scalar and to carry along one or more derivative components. More specifically, one derivative component is maintained for each column of the seed matrix  $S$  in (7). The operator overloading approach performs elementary derivative computations in addition to the elementary function evaluation computations. A straightforward implementation of forward-mode AD using operator overloading handles piecewise defined functions by navigating through conditional branches (e.g. `if` statements), and when combined with C++ function overloading, this approach can also be used to define rules to differentiate through non-C++ or third-party function calls.

Several different C++ classes use operator overloading to implement the forward mode of AD<sup>3</sup>. Although these classes use different approaches, we focus on the forward AD (Fad) suite of C++ classes [Cesare and Pironneau 2000]. These methods are templated on the scalar type and therefore allow great flexibility to compute, for example, second and higher-order derivatives by nesting AD types. The Fad classes use a C++ template programming technique called *expression templates* which results in assembler code that eliminates many of the temporary objects commonly created by operator overloading in C++. One of the classes (TFad) is also templated on the number of derivative components and therefore does not impose any runtime dynamic memory allocation. For the remainder of this paper, when we refer to AD approaches in C++ we will always be referring to operator overloading C++ AD classes and never will we consider possible source transformation approaches, again since they currently do not exist.

As pointed out in [Martins et al. 2003], the CS and operator-overloading AD approaches share common features with respect to computing the single vector right-hand side form of (2). Both use operator overloading, maintain two floating-point values (the function value and the single derivative component) for each scalar variable, and can produce essentially analytic derivative values. However, AD methods are more computationally efficient for a number of reasons. First, the CS method performs unnecessary floating point operations, as in the case of a multiplication operation

$$(a + ia')(b + ib') = (ab - a'b') + i(ab' + a'b).$$

The  $i(ab' + a'b)$  term maintains a derivative component but the elementary multiplication and subtraction of  $a'b'$  are unnecessary;  $ab \approx ab - a'b'$  is essentially the value  $ab$  because  $ab \gg a'b'$  and in floating point the extra computation term will be entirely lost when  $|a'b'| > \epsilon|ab|$  (where  $\epsilon$  is machine precision). AD avoids these types of extraneous calculations. Second, specially designed AD types, such as TFad, can carry multiple derivative components instead of just one which makes the computation of (3) for  $p > 1$  right-hand sides more efficient since the function value  $f(x)$  is only computed once and amortized over the  $p$  multiple derivative components. In the case of the CS method, the function value must be computed repeatedly for each of the  $p$  columns of  $S$ . Third, the use of multi-component AD types can result in better memory performance (i.e. cache) since the expressions for  $f(x)$  are only accessed once. The CS method evaluates the function  $f(x)$  independently for each derivative component  $p$ .

Although we have made a compelling case for AD as the preferred method, there is opportunity for additional computational and implementational improvements by considering a hybrid strategy. If we recognize that most production software codes consist of multiple levels of complexity, the ease of implementation of AD and the computational efficiency of SD can be leveraged to strike a balance between implementation effort and computational performance. Assuming an unlimited pool of skilled developer resources, SD approaches should always be more computationally efficient (memory and CPU time) than AD. This is because SD can simply mimic what AD does by performing exactly the same operations and discarding unnecessary computations. Also, forward-mode operator overloading AD computes the function value in addition to the derivatives as shown in (7) even though this function value is usually not used (i.e. since it is already known). However, for general functions  $f(x)$ , specialized SD approaches can be very difficult to implement and maintain (as the function  $f(x)$  changes due to new requirements) even though in some cases, such as discretization methods for PDE, the high-level structure of  $f(x)$  can be exploited fairly easily. Assuming the target software code consists of multiple levels of complexity, a hybrid approach consists of symbolically deriving derivatives for those portions of code that present acceptable levels of complexities and applying AD to those portions of the

<sup>3</sup><http://www.autodiff.org>

code that present too much complexity. In this fashion, the hybrid strategy offers a trade-off between developer time and computational performance.

### 3. HYBRID SYMBOLIC/AD APPROACHES FOR ELEMENT-BASED ASSEMBLY MODELS

The goal of this section is to describe levels of intrusive symbolic differentiation combined with automated methods that can be used to efficiently compute Jacobian-vector products (2). We first describe a general model for the assembly of vector functions of the form (1), which encompasses many different application areas from various discretization methods for PDEs (e.g. finite-element, finite-volume, discontinuous Galerkin etc.) to network models (e.g. electrical devices, utility networks, processing plants etc.). We start with an abstract model, Algorithm 3.1, to establish the general methodology before discussing more complicated issues.

ALGORITHM 3.1. GENERIC ELEMENT-BASED ASSEMBLY MODEL OF THE GLOBAL FUNCTION  $f(x)$

---

$x \rightarrow f(x)$

(1) *Initialization*  
 $f = 0$

(2) *Element assembly*  
 for  $e = 1 \dots N_e$   
   *Gather local variables*  
    $x_e = P_{x,e} x$   
   *Local computation*  
    $f_e = f_e(x_e)$   
   *Global scatter and assembly*  
    $f = f + P_{f,e}^T f_e$

---

The assembly model in Algorithm 3.1 shows a summation assembly of independent element computations. We use the term element in a general sense that is not restricted to just finite-element methods. A relatively small number of variables are gathered for each element by the operation  $x_e = P_{x,e} x$ , which we refer to as a “local” computation<sup>4</sup>. This vector of variables  $x_e \in \mathbf{R}^{n_e}$  is then used in a relatively compact, local computation  $x_e \rightarrow f_e(x_e)$  to form  $f_e \in \mathbf{R}^{m_e}$ . In general  $n_e \ll n$  and  $m_e \ll m$ . For each element, the local function  $f_e$  is assembled for the entire computational domain into  $f = f + P_{f,e}^T f_e$ , which is referred to as a “global” function. This assembly model uses abstractions of mapping matrices  $P_{x,e}$ , and  $P_{f,e}$  to represent the indexing of local/global variables and functions respectively. These matrices contain only columns of identity or zeros depending on the formulation of the problem. The non-transposed linear operator  $P$  performs global-to-local mappings as

$$v_l = P v_g \tag{8}$$

and the transposed operator  $P^T$  performs the local-to-global mappings as

$$v_g = P^T v_l. \tag{9}$$

The element functions  $f_e(x_e)$  ( $e = 1 \dots N_e$ ) may represent the physics computation and the discretization method of a PDE in which each computation can be processed independently and potentially

<sup>4</sup>Our term “local computation” or “element computation” is also know as an *interface contraction* in the AD research community[Griewank 2000].

in parallel. In the case of a finite-element PDE based simulation, the element loop in Algorithm 3.1 would involve both traditional internal element loops and one or more boundary loops. Typically, more data than just  $x_e$  is needed to define the functions  $f_e(\dots)$  and this abstract model assumes that this data is encapsulated in the mathematical functions  $f_e(\dots)$  themselves. The assembly in Algorithm 3.1 can be compactly written as

$$f(x) = \sum_{e=1}^{N_e} P_{f,e}^T f_e(P_{x,e}x). \quad (10)$$

This form will be useful in deriving the the various global derivative objects in the following section.

### 3.1 Global derivative computations

In this section, we consider how local element-wise derivative computations can be used with the element assembly model in Algorithm 3.1 to compute the following global derivative objects:

(1) *Sparse Jacobian matrix:*

$$J = \frac{\partial f}{\partial x} \quad (11)$$

(2) *Jacobian-vector product:*

$$\delta f = \frac{\partial f}{\partial x} \delta x \quad (12)$$

These are the primary computations needed for a number of numerical algorithms including linear and nonlinear equation solvers, stability analysis methods, uncertainty quantification, and optimization. From the assembly model in (10), the Jacobian matrix is given by

$$\frac{\partial f}{\partial x} = \sum_{e=1}^{N_e} P_{f,e}^T \frac{\partial f_e}{\partial x_e} P_{x,e}. \quad (13)$$

The assembly of the sparse Jacobian matrix  $J = \partial f/\partial x$  follows directly from (13) and is given in Algorithm 3.2.

---

#### ALGORITHM 3.2. ASSEMBLY OF THE GLOBAL JACOBIAN MATRIX

---

$(x) \rightarrow J = \frac{\partial f}{\partial x}$

(1) *Initialization*

$$J = 0$$

(2) *Element assembly*

$$\text{for } e = 1 \dots N_e \\ J = J + P_{f,e}^T \frac{\partial f_e}{\partial x_e} P_{x,e}$$


---

Algorithm 3.2 requires that the element Jacobians  $\partial f_e/\partial x_e$  be explicitly computed using TFad [Cesare and Pironneau 2000]. The mapping matrices  $P_{x,e}$  and  $P_{f,b}$  simply define how the local element Jacobians  $\partial f_e/\partial x_e$  are scattered and added into the global sparse Jacobian matrix  $J$ . Interfaces supporting this type of mapping are common in many different types of codes. For the remainder of the paper it is assumed the element-level Jacobians  $\partial f_e/\partial x_e$  are computed at the current  $x_e$ .

The assembly of a Jacobian-vector product (2) follows directly from (13) as

$$\begin{aligned}\delta f &= \frac{\partial f}{\partial x} \delta x \\ &= \sum_{e=1}^{N_e} P_{f,e}^T \frac{\partial f_e}{\partial x_e} P_{x,e} \delta x.\end{aligned}\tag{14}$$

and is restated in Algorithm 3.3.

---

**ALGORITHM 3.3. ASSEMBLY OF THE GLOBAL JACOBIAN-VECTOR PRODUCT**

---

$(x, \delta x) \rightarrow \delta f = \frac{\partial f}{\partial x} \delta x$

(1) *Initialization*

$$\delta f = 0$$

(2) *Element assembly*

for  $e = 1 \dots N_e$

$$\delta x_e = P_{x,e} \delta x$$

$$\delta f_e = \frac{\partial f_e}{\partial x_e} \delta x_e$$

$$\delta f = \delta f + P_{f,e}^T \delta f_e$$


---

### 3.2 Storage and element derivative computations

The forward assembly Algorithm 3.3 can be used in one of two ways. The first general class will be referred to as **precomputed-storage** approaches, which involve computing the Jacobian matrices  $\partial f_e / \partial x_e$  upfront in a single loop and storing these element matrices in an element-wise data structure. The precomputed  $\partial f_e / \partial x_e$  matrices can then be used to assemble Jacobian-vector products in Algorithm 3.3 through local matrix-vector products. These repeated assembly loops only utilize the precomputed  $\partial f_e / \partial x_e$  matrices and therefore do not invoke the actual element functions  $f_e(x_e)$  themselves. The main advantage of these approaches are that they can result in dramatic reductions in cost of repeated Jacobian-vector products. The disadvantages include potentially expensive upfront AD computations and significant storage.

The second general class will be referred to as **storage-free** approaches and their main advantage is that they do not involve any upfront computation or storage. Instead applications of forward AD with  $f_e(x_e)$  are used to compute Jacobian-vector products. The disadvantage of these approaches over *precomputed-storage* approaches is that the formation of each product assembly is more expensive since the element functions  $f_e(x_e)$  must be called repeatedly using AD data types.

The global derivative assembly computations described in the preceding sections require the following types of element-wise derivative computations:

(1) *Element Jacobian matrix:*

$$J_e = \frac{\partial f_e}{\partial x_e} \in \mathbf{R}^{m_e \times n_e}\tag{15}$$

(2) *Element Jacobian-vector product:*

$$\delta f_e = \frac{\partial f_e}{\partial x_e} \delta x_e \in \mathbf{R}^{m_e}\tag{16}$$

Note that *storage-free* approaches only require directional derivatives (16) for Algorithm 3.3. As mentioned previously, we assume that the dimensions of each element computation  $n_e$  and  $m_e$  are

relatively small (e.g. order 100 or less) and that the mapping from the input element variables  $x_e$  to the output functions  $f_e = f_e(x_e)$  is fairly dense (i.e.  $\partial f_e / \partial x_e$  has mostly non-zero entries). This is generally true for many different types of applications but there are some exceptions (e.g. chemically reacting flows with lots of species and sparse reactions). Given that we are assuming that  $n_e$  and  $m_e$  are relatively small and  $x_e \rightarrow f_e(x_e)$  is fairly dense, these element-wise derivatives can be computed most efficiently and conveniently through AD.

### 3.3 C++ implementation

In C++ the implementation of AD is especially easy if the functions  $f_e(x_e)$  (or the classes that implement these functions) are templated on the scalar types for the input arguments  $x_e$  and the output vector arguments for  $f_e$ . For example, suppose the following non-member C++ function computes  $f_e(x_e, d_e)$ :

```
void eval_ele_func(
    const double      x_e[],
    const ElementData &d_e,
    double           f_e[]
);
```

where `d_e` is an object that defines the rest of the element-specific data (e.g. nodal coordinates etc. for a PDE discretization) for the element computation. To facilitate the use of automatic differentiation, the above function can be templated as follows:

```
template<class Scalar>
void eval_ele_func(
    const Scalar      x_e[],
    const ElementData &d_e,
    Scalar           f_e[]
);
```

Dense Jacobians and forward Jacobian-vector products can be efficiently computed using the forward mode of AD and easily implemented using the templated class `TFad<N, T>`. This AD type is templated both on the underlying scalar type `T` and the number of derivative components `N`. By templating on the number of derivative components, all memory allocation can be performed on the stack and therefore greatly improve performance by avoiding many small dynamic memory allocations that often occur with operator overloaded AD tools. The applied use of `TFad` consists of instantiating the code for  $f_e(\dots)$  using `TFad`, initializing the input independent `TFad` variables appropriately, executing the `TFad`-enabled function, and extracting the desired derivatives from the output arguments. By using only one derivative component (i.e. `TFad<1, double>`), a Jacobian-vector product in (16) can be cheaply computed at a cost of less than twice the storage and less than three times the flops of the function evaluation. However, generating a Jacobian with respect to the `N` variables requires using `N` derivative components (i.e. `TFad<N, double>`), and the resulting computation will, in theory, require up to  $3N$  more flops than the original function evaluation. However, certain operations, like square roots and exponentials, can reduce the relative cost of each derivative component.

The following C++ functions give examples of the use of `TFad` for computing Jacobian matrices (15) and Jacobian-vector products (16) for the templated function `eval_ele_func(\dots)`.

```
//
// Compute an ``element`` (column-major) Jacobian matrix
//
void eval_ele_state_jac(
    const double      x_e[],
```

```

const ElementData &d_e,
double          J_e[]
)
{
const int N_x_e = 10; // Number components in x_e[]
const int N_f_e = 5; // Number components in f_e[]
// Initialize AD argumets
TFad<N_x_e,double> ad_x_e[N_x_e];
TFad<N_x_e,double> ad_f_e[N_f_e];
for( int k = 0; k < N_x_e; ++k ) {
    ad_x_e[k].val() = x_e[k];           // Set independent var values
    ad_x_e[k].diff(k);                 // Setup identity seed matrix
}
// Run function in forward mode to compute the entire state Jacobian
eval_ele_func( ad_x_e, d_e, ad_f_e );
// Extract state Jacobian matrix in column-major format
for( int k1 = 0; k1 < N_f_e; ++k1 ) for( int k2 = 0; k2 < N_x_e; ++k2 )
    J_y_e[k1+k2*N_f_e] = ad_f_e[k1].fastAccessDx(k2);
}

//
// Compute an ``element`` Jacobian-vector product
//
void eval_ele_jac_vec(
    const double          x_e[],
    const ElementData &d_e,
    const double          delta_x_e[],
    double                delta_f_e[]
)
{
const int N_x_e = 10; // Number components in x_e[]
const int N_f_e = 5; // Number components in f_e[]
// Initialize AD argumets
TFad<1,double> ad_x_e[N_x_e], ad_f_e[N_f_e];
for( int k = 0; k < N_x_e; ++k ) {
    ad_x_e[k].val() = x_e[k];           // Set independent var values
    ad_x_e[k].fastAccesDx(0) = delta_x_e[k]; // Load single seed vector
}
// Run function in forward mode
eval_ele_func( ad_x_e, d_e, ad_f_e );
// Extract state Jacobian-vector product
for( int k = 0; k < N_f_e; ++k ) {
    delta_f_e[k] = ad_f_e[k].fastAccessDx(0);
}
}
}

```

As shown in the above code examples, using TFad is straightforward provided the number of independent variables is known at compile time. If the number of independent variables is not known at compile time then the (slightly) less efficient class Fad can be used instead. While the element derivative computations described above used only AD at the element level, these derivative computations

can also be performed using a hybrid symbolic/AD approach.

### 3.4 Levels of hybrid symbolic/AD differentiation

Here we introduce a classification system for different levels of hybrid differentiation strategies applied to functions exhibiting hierarchical dependencies. Complicated models for  $f(x)$  will consist of higher level calculations that depend on lower level calculations. Hybrid differentiation exploits this hierarchical structure by selecting the appropriate amount of symbolic differentiation combined with AD for the remaining lower levels. Here we focus on the types of element assembly models shown in Algorithm 3.1 and focus on the different levels of symbolic/AD approaches for assembling  $\delta x \rightarrow (\partial f / \partial x) \delta x$ . For each level, higher-level expressions are symbolically differentiated and then all remaining lower-level expressions are differentiated using AD. These levels are described next and provide a classification of our hybrid differentiation strategy for systematic performance comparisons of our particular example.

**Level 0** denotes the application of directional AD to the global C++ function that computes  $x \rightarrow f(x)$  without any concern to underlying structure. Level 0 requires the least amount of intrusive implementation, is therefore the easiest to use, and requires little knowledge of the structure of  $f(x)$ . The disadvantage of this level is that the entire function  $x \rightarrow f(x)$  needs to be templated on the scalar type, which may not be practical for many codes. Only *storage-free* methods are used with this most basic level.

**Level 1** denotes the application of AD to the element level functions  $f_e(x_e)$ . Both the *storage-free* and *precomputed-storage* methods can be used for this level. In the case of the *storage-free* approach, directional AD is applied at the element assembly level to compute  $\delta x_e \rightarrow (\partial f_e / \partial x_e) \delta x_e$ , followed by an assembly of the global Jacobian-vector product as shown in Algorithm 3.3. In the case of *precomputed-storage* approaches, the seed matrix is set to identity  $S = I$  and the element Jacobians  $\partial f_e / \partial x_e$  are precomputed using the forward mode of AD applied to  $f_e(x_e)$ . These precomputed element Jacobian matrices are then used to assemble subsequent Jacobian-vector products. The main advantage of this level is that it requires little knowledge of the underlying structure of the expressions or mathematical problem and only involves manipulating the basic assembly process. In addition, only the element functions for  $f_e(x_e)$  and not the entire C++ function  $x \rightarrow f(x)$ , need be templated on the scalar type.

**Levels 2 and higher** denotes symbolically differentiating into the element-level expressions  $f_e(x_e)$  and applying AD on the remaining lower levels. Higher levels may involve symbolically differentiating deeper into the expressions for  $f_e(x_e)$ . The number of meaningful levels depends on the nature of the expressions for  $f_e(x_e)$ . These higher-level hybrid symbolic/AD methods may be applied using *precomputed-storage* or *storage-free* approaches. The *precomputed-storage* approaches may avoid storing the element Jacobians  $\partial f_e / \partial x_e$  and instead may only store contributions to these Jacobians potentially resulting in reduced storage and better performance. The *level-2* approach described in Section 4 is an example of this. In general, for each increasing level of hybrid symbolic/AD method, the amount of developer work will increase to implement and maintain additional code. However, these more invasive methods offer reduced storage (i.e. for *precomputed-storage* methods) and improved runtime (i.e. for *precomputed-storage* and *storage-free*).

Section 4 describes a concrete example for the use of several different levels of *precomputed-storage* and *storage-free* symbolic/AD approaches and demonstrates the potential improvements in storage cost and runtime performance for increasing levels of hybrid symbolic/AD methods.

## 4. FINITE-VOLUME DISCRETIZATION EXAMPLE

We present a concrete case study of various differentiation techniques and the application of different levels of hybrid symbolic/AD approaches. Specifically, hybrid differentiation methods are demon-

strated for a finite-volume discretization method of gas dynamics (Euler equations) which represents an example of a two-loop element assembly model. The single loop of independent elements shown in Algorithm 3.1 accurately applies to a large number of application areas such as finite-element methods and different types of network models. However, a single independent loop over a set of elements is not sufficient for certain discretization and application areas. One instance of this is the finite-volume example described in this section where multiple coupled loops are required.

We exploit the hierarchical structure by first assuming very little and thereby exposing the entire function evaluation to AD, followed by systematically considering more details of the function for symbolic differentiation opportunities. We start with level 0, but as additional levels of differentiation are considered, more details of the Euler equations will be revealed and discussed. This approach will demonstrate the utility of the hybrid strategy and provide an improved approach to balancing implementation effort against computational efficiency. In addition, we gloss over many details of the discretization, and provide only enough algorithmic information to describe the use of the different differentiation techniques. For additional details regarding computational fluid dynamics we refer the interested reader to Blazek [2001].

#### 4.1 Level 0 – the basic residual function

At the center of this finite-volume fluid flow code is the computation of the *residual function*

$$\mathbf{w} \rightarrow \mathbf{r}(\mathbf{w}) \in \mathbf{R}^{(n_d+2)n_N} \rightarrow \mathbf{R}^{(n_d+2)n_N} \quad (17)$$

where  $n_d$  is the spatial dimension ( $n_d = 2$  or  $n_d = 3$ ),  $n_N$  is the number of nodes in the computational mesh, and  $\mathbf{w} \in \mathbf{R}^{(n_d+2)n_N}$  is a vector of fluid states (density, velocity, pressure) in so-called conservative form (i.e. the formulation is written in terms of the so-called conservative variables). Our code implements the residual function (17), which is templated on the scalar types of each of the input vectors. It should also be noted that more data than just the vector of state variables  $\mathbf{w}$  is passed into this function but in the context of AD, this data is irrelevant. Templating this C++ function on the scalar type for  $\mathbf{w}$  facilitates the straightforward use of the AD type `TFad` at all levels of computation. While several derivative calculations are needed for gas dynamics, we focus on the computation of the state Jacobian-vector product

$$\delta \mathbf{r} = \frac{\partial \mathbf{r}}{\partial \mathbf{w}} \delta \mathbf{w} \quad (18)$$

which is needed by various solution strategies for the nonlinear equation  $\mathbf{r}(\mathbf{w}) = \mathbf{0}$ .

We start with applying AD to the entire residual evaluation function and thereby not requiring any lower level details. The residual evaluation code for (17) is templated on the scalar type. Thus, if we can assume that the function is first-order differentiable, then we do not need to know how the function (17) is evaluated in order to compute (18) using the forward mode of AD<sup>5</sup>. The templated residual evaluation function only needs to be instantiated with the scalar type `TFad<1, double>` for the input vector  $\mathbf{w}$  and the output vector  $\mathbf{r}$  in order to compute (18) using the forward mode of AD. All of the other data types for the passive input data (which are inconsequential for a *level-0* method) are left as `double`. This selective template instantiation prevents wasted derivative computations for inactive input data.

<sup>5</sup>While it is true that operator overloading C++ AD classes will correctly differentiate most C++ numerical operations, there are cases, such as the poor use of conditionals, where it will yield the wrong result. While many of these cases are well known to the AD community, they may not be as well known to the general numerical computing community. For example, applying AD to the function `Scalar func( Scalar x ) { return ( x==0.0 ? 0.0 : x ); }` will yield the wrong derivative at `x==0.0`.

In the following sections, we consider symbolically differentiating lower level calculations of the residual evaluation (17). For each level of the symbolic differentiation process, we describe the computations inside of (17) in only enough detail to be able to apply AD strategically to the remaining lower levels.

## 4.2 Level 1 - two-loop edge-based residual assembly

Implementing a basic *level-1* hybrid symbolic/AD method requires knowledge about the discretization method since this defines the element assembly process in Algorithm 3.1. In our code, the residual assembly is accomplished through two loops over all edges in the computational mesh. The kind of finite-volume discretization that we use forms a control volume around each mesh node, within which the fluid state is assumed to be constant. The physics consists of balancing the flux of mass, momentum, and energy along the edges that connect the control volumes. The edges may be considered as the basic “elements” of this particular discretization. The finite-volume discretization described here uses an unstructured meshing—with tetrahedra, hexahedra, or prisms—of the flow domain  $\Omega$ . We denote by  $\mathcal{V}(\Omega)$  the set of mesh nodes in the strict interior of the domain and by  $\mathcal{V}(\partial\Omega)$  the nodes on the boundary of the mesh. Thus, the set of all nodes is  $\mathcal{V}(\bar{\Omega}) = \mathcal{V}(\Omega) \cup \mathcal{V}(\partial\Omega)$ . Moreover, let  $\mathcal{N}_i$  denote the set of nodes that are nearest neighbors connected to node  $i$  by an edge. We associate a edge norm vector  $\mathbf{n}_{ij}$  with each directed edge  $\xrightarrow{i,j}$  in the mesh. The normals are computed from a dual mesh [Blazek 2001], and by construction,

$$\mathbf{n}_{ji} = -\mathbf{n}_{ij}. \quad (19)$$

The normals are not of unit length and therefore embody both direction and magnitude. We define  $V_i$  to be the volume of the control volume surrounding each mesh vertex  $i$ . Also note that the normals  $\mathbf{n}_{ij}$  are computed strictly from the geometry of the mesh and have no dependence on any state variables. The residual in (17) is assembled in two edge-based loops as shown in Algorithm 4.1.

---

### ALGORITHM 4.1. *Two-loop edge-based residual assembly*

---

*Compute the residual vector  $\mathbf{r}$  given the input vectors  $\mathbf{w}$ ,  $\mathbf{x}$ ,  $\mathbf{n}$  and  $V$ .*

*# Spatial gradient assembly*

- (1) Set  $\mathbf{g}_i \leftarrow \mathbf{0}$  for each  $i \in \mathcal{V}(\bar{\Omega})$
- (2) For each edge  $\xrightarrow{i,j}$ :
  - (a)  $\mathbf{g}_i \leftarrow \mathbf{g}_i + \hat{\mathbf{g}}(V_i, \mathbf{w}_i, \mathbf{w}_j, +\mathbf{n}_{ij})$
  - (b)  $\mathbf{g}_j \leftarrow \mathbf{g}_j + \hat{\mathbf{g}}(V_j, \mathbf{w}_j, \mathbf{w}_i, -\mathbf{n}_{ij})$
- (3) + *Boundary contributions*

*# Residual assembly*

- (1) Set  $\mathbf{r}_i \leftarrow \mathbf{0}$  for each  $i \in \mathcal{V}(\bar{\Omega})$
  - (2) For each edge  $\xrightarrow{i,j}$ :
    - (a)  $\hat{\mathbf{r}}_{ij} = \hat{\mathbf{r}}(\mathbf{w}_i, \mathbf{w}_j, \mathbf{x}_i, \mathbf{x}_j, \mathbf{g}_i, \mathbf{g}_j, \mathbf{n}_{ij})$
    - (b)  $\mathbf{r}_i \leftarrow \mathbf{r}_i + \hat{\mathbf{r}}_{ij}$
    - (c)  $\mathbf{r}_j \leftarrow \mathbf{r}_j - \hat{\mathbf{r}}_{ij}$
  - (3) + *Boundary contributions*
-

A single templated C++ function `assemble_grad(...)` is called for each edge in Algorithm 4.1 to simultaneously compute  $\hat{\mathbf{g}}(V_i, \mathbf{w}_i, \mathbf{w}_j, +\mathbf{n}_{ij})$  and  $\hat{\mathbf{g}}(V_j, \mathbf{w}_j, \mathbf{w}_i, -\mathbf{n}_{ij})$  in which some of the same computations are shared. A single templated C++ function `assemble_resid(...)` is called for each edge in Algorithm 4.1 to compute  $\hat{\mathbf{r}}(\mathbf{w}_i, \mathbf{w}_j, \mathbf{x}_i, \mathbf{x}_j, \mathbf{g}_i, \mathbf{g}_j, \mathbf{n}_{ij})$ . While the treatment of boundary conditions is critical to any discretization method, the calculation usually does not contribute significantly to the computational cost of the state residual (17). Therefore, for the purposes of this discussion, we will not discuss the details of boundary conditions, since our goal is to address the bulk computational work. Enforcement of boundary conditions would represent one or more element loops that would contribute to the residual assembly.

### 4.3 Level 1 - state Jacobian-vector product assembly

By inspecting the edge-based assembly in Algorithm 4.1 and using the multi-variable chain rule, the edge-based assembly loops for the Jacobian-vector product can be derived as shown in Algorithm 4.2.

ALGORITHM 4.2. *Level-1 hybrid symbolic/AD Jacobian-vector product assembly*

---

Compute  $\delta \mathbf{r} = \frac{\partial \mathbf{r}}{\partial \mathbf{w}} \delta \mathbf{w}$  given input vector  $\delta \mathbf{w}$

# Linearized spatial gradient assembly

(1) Set  $\delta \mathbf{g}_i \leftarrow \mathbf{0}$  for each  $i \in \mathcal{V}(\bar{\Omega})$

(2) For each edge  $\overset{ij}{\rightarrow}$ :

(a)  $\delta \mathbf{g}_i \leftarrow \delta \mathbf{g}_i + \frac{\partial \hat{\mathbf{g}}_{ij}}{\partial \mathbf{w}_i} \delta \mathbf{w}_i + \frac{\partial \hat{\mathbf{g}}_{ij}}{\partial \mathbf{w}_j} \delta \mathbf{w}_j$

(b)  $\delta \mathbf{g}_j \leftarrow \delta \mathbf{g}_j + \frac{\partial \hat{\mathbf{g}}_{ji}}{\partial \mathbf{w}_i} \delta \mathbf{w}_i + \frac{\partial \hat{\mathbf{g}}_{ji}}{\partial \mathbf{w}_j} \delta \mathbf{w}_j$

(3) + Boundary contributions

# Linearized residual assembly

(1) Set  $\delta \mathbf{r}_i \leftarrow \mathbf{0}$  for each  $i \in \mathcal{V}(\bar{\Omega})$

(2) For each edge  $\overset{ij}{\rightarrow}$ :

(a)  $\delta \hat{\mathbf{r}}_{ij} = \frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{w}_i} \delta \mathbf{w}_i + \frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{w}_j} \delta \mathbf{w}_j + \frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{g}_i} \delta \mathbf{g}_i + \frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{g}_j} \delta \mathbf{g}_j$

(b)  $\delta \mathbf{r}_i \leftarrow \delta \mathbf{r}_i + \delta \hat{\mathbf{r}}_{ij}$

(c)  $\delta \mathbf{r}_j \leftarrow \delta \mathbf{r}_j - \delta \hat{\mathbf{r}}_{ij}$

(3) + Boundary contributions

---

In Algorithm 4.2, we use the notation

$$\frac{\partial \hat{\mathbf{g}}_{ij}}{\partial \mathbf{w}_i} = \frac{\partial}{\partial \mathbf{w}} \hat{\mathbf{g}}(V_j, \mathbf{w}, \mathbf{w}_j, \mathbf{n}_{ij})|_{\mathbf{w}=\mathbf{w}_i}, \quad \frac{\partial \hat{\mathbf{g}}_{ij}}{\partial \mathbf{w}_j} = \frac{\partial}{\partial \mathbf{w}} \hat{\mathbf{g}}(V_j, \mathbf{w}_i, \mathbf{w}, \mathbf{n}_{ij})|_{\mathbf{w}=\mathbf{w}_j}, \quad (20)$$

$$\frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{w}_i} = \frac{\partial}{\partial \mathbf{w}} \hat{\mathbf{r}}(\mathbf{w}, \mathbf{w}_j, \mathbf{x}_i, \mathbf{x}_j, \mathbf{g}_i, \mathbf{g}_j, \mathbf{n}_{ij})|_{\mathbf{w}=\mathbf{w}_i}, \quad \frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{w}_j} = \frac{\partial}{\partial \mathbf{w}} \hat{\mathbf{r}}(\mathbf{w}_i, \mathbf{w}, \mathbf{x}_i, \mathbf{x}_j, \mathbf{g}_i, \mathbf{g}_j, \mathbf{n}_{ij})|_{\mathbf{w}=\mathbf{w}_j}, \quad (21)$$

$$\frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{g}_i} = \frac{\partial}{\partial \mathbf{g}} \hat{\mathbf{r}}(\mathbf{w}_i, \mathbf{w}_j, \mathbf{x}_i, \mathbf{x}_j, \mathbf{g}, \mathbf{g}_j, \mathbf{n}_{ij})|_{\mathbf{g}=\mathbf{g}_i}, \quad \frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{g}_j} = \frac{\partial}{\partial \mathbf{g}} \hat{\mathbf{r}}(\mathbf{w}_i, \mathbf{w}_j, \mathbf{x}_i, \mathbf{x}_j, \mathbf{g}_i, \mathbf{g}, \mathbf{n}_{ij})|_{\mathbf{g}=\mathbf{g}_j}. \quad (22)$$

The Jacobian-vector product assembly in Algorithm 4.2 constitutes a *level-1* hybrid symbolic/AD differentiation into the residual evaluation. At this level, AD only needs to be applied to the functions  $\hat{\mathbf{g}}(\dots)$  and  $\hat{\mathbf{r}}(\dots)$  that operate on objects associated with an edge and not the entire residual assembly function for (17).

We consider two *level-1* hybrid symbolic/AD strategies at the edge level for assembling the Jacobian-vector product in expression (18). The first is the *precomputed-storage* approach and the second is the *storage-free* approach. The *precomputed-storage* approach assembles Jacobian-vector products by computing edge-based Jacobian sub-matrices and storing them. The *storage-free level-1* approach simply applies AD at the edge level to local edge-based functions  $\hat{\mathbf{g}}(\dots)$  and  $\hat{\mathbf{r}}(\dots)$  using a single directional derivative. The *precomputed-storage level-1* approach initially applies AD at the edge-level to explicitly compute local Jacobians, stores them, and then assembles the full Jacobian-vector product by matrix-vector multiplication through loops over the edges, utilizing the pre-stored local Jacobians.

When using the *precomputed-storage level-1* approach, the local edge-based gradient Jacobians

$$\begin{aligned} \frac{\partial \hat{\mathbf{g}}_{ij}}{\partial \mathbf{w}_i} &\in \mathbf{R}^{(n_d+2)n_d \times (n_d+2)}, & \frac{\partial \hat{\mathbf{g}}_{ij}}{\partial \mathbf{w}_j} &\in \mathbf{R}^{(n_d+2)n_d \times (n_d+2)}, \\ \frac{\partial \hat{\mathbf{g}}_{ji}}{\partial \mathbf{w}_j} &\in \mathbf{R}^{(n_d+2)n_d \times (n_d+2)}, & \frac{\partial \hat{\mathbf{g}}_{ji}}{\partial \mathbf{w}_i} &\in \mathbf{R}^{(n_d+2)n_d \times (n_d+2)}, \end{aligned} \quad (23)$$

are computed. Without additional knowledge about the structure of  $\hat{\mathbf{g}}(\dots)$ , the storage for these Jacobians requires  $4((n_d+2)(n_d)(n_d+2)) = 300$  doubles per edge in 3D. However, these matrices are actually diagonal, since the gradient is computed separately for each component of the conservative variables. The diagonal structure reduces the storage to  $4((n_d+2)(n_d)) = 4((3+2)(3)) = 60$  doubles per edge in 3D. In addition, when using AD to generate these matrices, all of the variables in either  $\mathbf{w}_i$  and  $\mathbf{w}_j$  can be perturbed simultaneously requiring just two columns in the seed matrix  $S$ .

After the Jacobians (23) have been computed, the edge-based residual Jacobians

$$\begin{aligned} \frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{w}_i} &\in \mathbf{R}^{(n_d+2) \times (n_d+2)}, & \frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{w}_j} &\in \mathbf{R}^{(n_d+2) \times (n_d+2)}, \\ \frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{g}_i} &\in \mathbf{R}^{(n_d+2) \times (n_d+2)n_d}, & \frac{\partial \hat{\mathbf{r}}_{ij}}{\partial \mathbf{g}_j} &\in \mathbf{R}^{(n_d+2) \times (n_d+2)n_d}, \end{aligned} \quad (24)$$

are computed and stored in a loop over all edges. Computing these edge-based Jacobians requires invoking forward AD (e.g. using `TFad<40, double>`) using a seed matrix with  $(n_d+2)(2+2n_d) = 40$  columns in 3D (one column of identity for each component in  $\mathbf{w}_i$ ,  $\mathbf{w}_j$ ,  $\mathbf{g}_i$  and  $\mathbf{g}_j$ ). These edge-based Jacobians require the storage of  $2(n_d+2)(n_d+2) + 2(n_d+2)(n_d+2)(n_d) = 200$  doubles per edge in 3D. Therefore, the total storage for edge-based Jacobians is  $60 + 200 = 260$  doubles per edge in 3D. This is a significant amount of storage but, as shown in Section 4.5, the use of pre-computed Jacobians results in much more rapid evaluations of (18).

#### 4.4 Level 2 - Jacobian-vector product

Symbolically differentiating deeper into the residual evaluation requires knowing more about the computations at the edge level. The next level of structure of these computations is more complicated, but great gains in speed and memory savings can be accomplished by symbolically differentiating deeper. First we describe the edge-based spatial gradient function  $\hat{\mathbf{g}}(\dots)$  and then the more complicated edge-based Euler residual function  $\hat{\mathbf{r}}(\dots)$ . The edge-based spatial gradient function  $\hat{\mathbf{g}}(\dots)$  takes the form

$$\hat{\mathbf{g}}(V_i, \mathbf{w}_i, \mathbf{w}_j, \mathbf{n}_{ij}) = \frac{1}{2|V_i|} (\mathbf{w}_i + \mathbf{w}_j) \otimes \mathbf{n}_{ij}. \quad (25)$$

where  $\otimes$  is a tensor product. The function  $\hat{\mathbf{g}}(\dots)$  is linear in  $\mathbf{w}_i$  and  $\mathbf{w}_j$  so simply passing in  $\delta\mathbf{w}_i$  and  $\delta\mathbf{w}_j$  into the `assemble_grad(...)` function returns the linearized spatial gradients  $\delta\mathbf{g}_i$  and  $\delta\mathbf{g}_j$ . Clearly AD is not needed for this computation.

The edge-based residual function  $\hat{\mathbf{r}}(\dots)$  uses a Roe dissipation scheme [Roe 1981] together with linear reconstruction and a smooth limiter as follows:

$$\begin{aligned}\hat{\mathbf{r}}_{ij} &= \hat{\mathbf{r}}(\mathbf{w}_i, \mathbf{w}_j, \mathbf{x}_i, \mathbf{x}_j, \mathbf{g}_i, \mathbf{g}_j, \mathbf{n}_{ij}) \\ &= \frac{1}{2} (\mathbf{f}(\mathbf{w}_{ij}^{i+}) \cdot \mathbf{n}_{ij}) + \frac{1}{2} (\mathbf{f}(\mathbf{w}_{ij}^{j-}) \cdot \mathbf{n}_{ij}) + \mathbf{d}(\mathbf{w}_{ij}^{i+}, \mathbf{w}_{ij}^{j-}, \mathbf{n}_{ij})\end{aligned}\quad (26)$$

where:

$$\mathbf{w} = \begin{pmatrix} \rho \\ \rho\mathbf{u} \\ \rho E \end{pmatrix}, \quad (27)$$

$$\mathbf{f}(\mathbf{w}) = \begin{pmatrix} \rho\mathbf{u} \\ \rho\mathbf{u} \otimes \mathbf{u} + \mathbf{I}p \\ \mathbf{u}(\rho E + p) \end{pmatrix}, \quad \text{where} \quad (28)$$

$$p = (\gamma - 1)(\rho E - \frac{1}{2}|\mathbf{u}|^2),$$

$$\mathbf{w}_{ij}^{i+} = \mathbf{w}^+(\mathbf{w}_i, \mathbf{w}_j, \mathbf{p}_{ij}^i), \quad (29)$$

$$\mathbf{w}_{ij}^{j-} = \mathbf{w}^-(\mathbf{w}_i, \mathbf{w}_j, \mathbf{p}_{ij}^j), \quad (30)$$

$$\mathbf{p}_{ij}^i = \mathbf{g}_i \cdot (\mathbf{x}_j - \mathbf{x}_i) \in \mathbf{R}^{(n_d+2)}, \quad (31)$$

$$\mathbf{p}_{ij}^j = \mathbf{g}_j \cdot (\mathbf{x}_j - \mathbf{x}_i) \in \mathbf{R}^{(n_d+2)}. \quad (32)$$

$$(33)$$

Above,  $\mathbf{f}(\mathbf{w})$  in (28) is the flux function of the Euler equations  $\nabla \cdot \mathbf{f}(\mathbf{w}) = 0$ . Moreover,  $\rho$ ,  $\mathbf{u} \in \mathbf{R}^d$ ,  $p$ , and  $E$  are the density, velocity vector, pressure, and total energy per unit volume, respectively. The vectors  $\mathbf{w}_{ij}^{i+}$  and  $\mathbf{w}_{ij}^{j-}$  are the reconstructed left and right states. These reconstructions use the spatial gradients  $\mathbf{g}_i$  and  $\mathbf{g}_j$  and are intended to increase the accuracy in regions where the solution is smooth at little additional computational cost but at the expense of effectively increasing the stencil, or footprint, of the operator. Embedded in the reconstruction functions  $\mathbf{w}^+(\dots)$  and  $\mathbf{w}^-(\dots)$  is a limiter that is needed to handle the very steep gradients around shocks. Several different limiters can be used and we used the van Albada limiter [van Albada B. van Leer and Jr. 1982] in our method. For the purposes of this discussion the details of the limiter are not important, except that we use limiters that are piecewise differentiable. Finally,  $\mathbf{d}(\dots)$  is a dissipation term (of Roe type) that provides stability and dominates the computational cost of the method. This term is algebraically complicated and would be time-consuming to symbolically differentiate and implement.

From the form of  $\hat{\mathbf{r}}_{ij}$  in (26), its linearization with respect to  $\mathbf{w}_i$ ,  $\mathbf{w}_j$ ,  $\mathbf{g}_i$  and  $\mathbf{g}_j$  immediately follows as:

$$\delta\hat{\mathbf{r}}_{ij} = \mathbf{J}_{ij}^{i+} \delta\mathbf{w}_{ij}^{i+} + \mathbf{J}_{ij}^{j-} \delta\mathbf{w}_{ij}^{j-} \quad (34)$$

where

$$\mathbf{J}_{ij}^{i+} = \frac{1}{2} \frac{\partial (\mathbf{f}_{ij}^{i+} \cdot \mathbf{n}_{ij})}{\partial \mathbf{w}_{ij}^{i+}} + \frac{\partial \mathbf{d}(\mathbf{w}_{ij}^{i+}, \mathbf{w}_{ij}^{j-}, \mathbf{n}_{ij})}{\partial \mathbf{w}_{ij}^{i+}}, \quad (35)$$

$$\mathbf{J}_{ij}^{j-} = \frac{1}{2} \frac{\partial (\mathbf{f}_{ij}^{j-} \cdot \mathbf{n}_{ij})}{\partial \mathbf{w}_{ij}^{j-}} + \frac{\partial \mathbf{d}(\mathbf{w}_{ij}^{j-}, \mathbf{w}_{ij}^{i+}, \mathbf{n}_{ij})}{\partial \mathbf{w}_{ij}^{j-}}, \quad (36)$$

$$\delta \mathbf{w}_{ij}^{i+} = \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_i} \delta \mathbf{w}_i + \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_j} \delta \mathbf{w}_j + \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{p}_{ij}^i} \delta \mathbf{p}_{ij}^i, \quad (37)$$

$$\delta \mathbf{w}_{ij}^{j-} = \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_i} \delta \mathbf{w}_i + \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_j} \delta \mathbf{w}_j + \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{p}_{ij}^j} \delta \mathbf{p}_{ij}^j, \quad (38)$$

$$\delta \mathbf{p}_{ij}^i = \delta \mathbf{g}_i \cdot (\mathbf{x}_i - \mathbf{x}_j), \quad (39)$$

$$\delta \mathbf{p}_{ij}^j = \delta \mathbf{g}_j \cdot (\mathbf{x}_j - \mathbf{x}_i), \quad (40)$$

and where we have used a notation for the derivatives analogous to expressions (20)–(22).

We now describe how this additional knowledge of the structure of  $\hat{\mathbf{r}}(\dots)$  is used to implement a *level-2* hybrid symbolic/AD Jacobian-vector product more efficiently. The *storage-free level-2* approach stores nothing up front and computes local Jacobian-vector products at the edge level symbolically for all terms except for the dissipation term  $\mathbf{d}(\dots)$ . In all cases, we used AD with `TFad<-double>` to perform all derivative computations with  $\mathbf{d}(\dots)$ . The *precomputed-storage level-2* approach initially computes and stores the sub-Jacobian matrices

$$\begin{array}{ll} \mathbf{J}_{ij}^{i+} \in \mathbf{R}^{(n_d+2) \times (n_d+2)} & \mathbf{J}_{ij}^{j-} \in \mathbf{R}^{(n_d+2) \times (n_d+2)} \\ \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_i} \in \mathbf{R}^{(n_d+2) \times (n_d+2)} & \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_j} \in \mathbf{R}^{(n_d+2) \times (n_d+2)} \\ \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{w}_j} \in \mathbf{R}^{(n_d+2) \times (n_d+2)} & \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{w}_i} \in \mathbf{R}^{(n_d+2) \times (n_d+2)} \\ \frac{\partial \mathbf{w}_{ij}^{i+}}{\partial \mathbf{p}_{ij}^i} \in \mathbf{R}^{(n_d+2) \times (n_d+2)} & \frac{\partial \mathbf{w}_{ij}^{j-}}{\partial \mathbf{p}_{ij}^j} \in \mathbf{R}^{(n_d+2) \times (n_d+2)} \end{array} \quad (41)$$

and performs matrix-vector multiplication with these sub-matrices to assemble subsequent Jacobian-vector products of the form (18). The sub-Jacobians for  $\mathbf{w}_{ij}^{i+}$  and  $\mathbf{w}_{ij}^{j-}$  are diagonal since the reconstructions and the limiters are component-wise operations. All of these Jacobian evaluations were manually derived and coded in C++ except for the dissipation term  $\mathbf{d}(\dots)$ . This scheme requires the storage of  $2(n_d+2)(n_d+2) + 4(n_d+2) + 2(n_d+2) = 2(3+2)(3+2) + 4(3+2) + 2(3+2) = 80$  doubles per edge in 3D. Once these edge-based sub-Jacobians are computed, they are utilized in the edge-based loops shown in Algorithm 4.3 to assemble the Jacobian-vector product (18).

---

ALGORITHM 4.3. *Level-2 hybrid symbolic/AD Jacobian-vector product assembly*

---

Compute  $\delta \mathbf{r} = \frac{\partial \mathbf{r}}{\partial \mathbf{w}} \delta \mathbf{w}$  given input vector  $\delta \mathbf{w}$

# Linearized spatial gradient assembly

(1) Set  $\delta \mathbf{g}_i \leftarrow \mathbf{0} \quad \forall i \in \mathcal{V}(\overline{\Omega})$

(2) For each edge  $\overset{i,j}{\rightarrow}$ :

Method	Precomputed-storage	Storage-free
Level-0	-	0
Level-1	260	0
Level-2	80	0

Table I. Storage of edge-based Jacobian contributions in number of doubles per edge

- (a)  $\delta \mathbf{g}_i \leftarrow \delta \mathbf{g}_i + \hat{\mathbf{g}}(V_i, \delta \mathbf{w}_i, \delta \mathbf{w}_j, +\mathbf{n}_{ij})$   
(b)  $\delta \mathbf{g}_j \leftarrow \delta \mathbf{g}_j + \hat{\mathbf{g}}(V_j, \delta \mathbf{w}_j, \delta \mathbf{w}_i, -\mathbf{n}_{ij})$   
(3) + *Boundary contributions*
- # *Linearized residual assembly*
- (1) Set  $\delta \mathbf{r}_i = \mathbf{0} \quad \forall i \in \mathcal{V}(\bar{\Omega})$
- (2) For each edge  $\overset{ij}{\rightarrow}$ :
- (a)  $\delta \mathbf{r}_{ij} \leftarrow \mathbf{J}_{ij}^{i+} \delta \mathbf{w}_{ij}^{i+} + \mathbf{J}_{ij}^{j-} \delta \mathbf{w}_{ij}^{j-}$ , where  $\delta \mathbf{w}_{ij}^{i+}$  and  $\delta \mathbf{w}_{ij}^{j-}$  are computed using (37)-(40).  
(b)  $\delta \mathbf{r}_i \leftarrow \delta \mathbf{r}_i + \delta \mathbf{r}_{ij}$ ,  
(c)  $\delta \mathbf{r}_j \leftarrow \delta \mathbf{r}_j - \delta \mathbf{r}_{ij}$ ,  
(3) + *Boundary contributions*
- 

#### 4.5 Results for various Jacobian-vector product assembly methods

Strategies to assemble Jacobian-vector products of the form (18) were discussed in the preceding sections and can be summarized as follows:

**Level-0:** Apply forward AD (i.e. using the seed matrix  $S = \delta \mathbf{w}$ ) at the global residual function call level (i.e. AD the entire C++ function for (17)).

**Level-1:** AD is applied at the edge assembly level.

**Precomputed-storage Level-1:** AD is applied at the edge level to generate the sub-matrices in (23) and (24); then the Jacobian-vector products are assembled with these sub-matrices (Algorithm 4.2).

**Storage-free Level-1:** Directional AD is applied at the edge assembly level on the functions  $\hat{\mathbf{g}}(\dots)$  and  $\hat{\mathbf{r}}(\dots)$ .

**Level-2:** Hybrid symbolic/AD is applied at the edge level by symbolically differentiating everything except the dissipation term  $\mathbf{d}(\dots)$ , which is differentiated using AD.

**Precomputed-storage Level-2:** The sub-matrices in (41) are computed symbolically up front with AD used only for the dissipation term  $\mathbf{d}(\dots)$ . Once these edge-level sub-matrices are computed and stored then Jacobian-vector products are assembled (Algorithm 4.3).

**Storage-free Level-2:** Edge-level Jacobian-vector products are computed using symbolically derived and hand-coded directional derivatives except for the dissipation term  $\mathbf{d}(\dots)$  where directional AD is used.

In this section we characterize the performance of the different levels of differentiation in addition to the consideration of different storage strategies. Tables I and II give storage and computational results, respectively, for *precomputed-storage* and *storage-free level-0, level-1* and *level-2* hybrid symbolic/AD schemes used to compute Jacobian-vector products of the form (18).

Method	Precomputation	Precomputed-storage Jac-vec	Storage-free Jac-vec
Level-0	-	-	3.14
Level-1	24.8	0.21	3.14
Level-2	4.73	0.19	1.93

Table II. Ratio of CPU time for residual Jacobian-vector product to residual evaluation for a small 3-D example with 25 mesh points.

We used a small, non-physical test mesh so that all of the computations easily fit in cache. All computations were run several times in a loop for a total of 1.0 CPU seconds in order to minimize the effects of random noise in the timing values. These tests therefore compare the floating point efficiency of the implemented code and not the quality of the data structures with respect to the memory-subsystem usage (e.g. cache misses). CPU times reported for all derivative computation are relative to the time for the residual evaluation that used the scalar type `double`. For all AD computations, the class `TFad<double>` was used as the scalar type for the active variables. All of the results in this section were generated by compiling code using GNU g++ version 3.1 and Red Hat Linux 7.2 on a 1.7 GHz Intel Pentium IV processor machine.

First we compare and contrast approaches that assemble Jacobian-vector products requiring no up-front storage or computation. These *storage-free* strategies may be preferred when memory is at a premium. The *storage-free Level-0* method required more CPU time by a factor of 3.14 in comparison to a residual evaluation. The *storage-free Level-0* is therefore only about 50% more expensive than a central FD approximation and the Jacobian-vector products are accurate to machine precision. The *storage-free Level-1* method gives exactly the same relative CPU time of 3.14 as the *storage-free Level-0* approach. This should not be surprising since this approach performs almost exactly the same computations as automatically differentiating the entire residual assembly function. In fact, in our C++ implementation, the Jacobian-vector products for the *storage-free Level-0* and *storage-free Level-1* approaches gave the same resultant vectors (to the last binary bit). The primary advantage of the *storage-free Level-1* approach over the *storage-free Level-0* approach is that the entire residual evaluation function does not have to be templated on the scalar type.

The *storage-free Level-2* approach exploits the gradient and residual computations and thereby reduces the relative CPU time from 3.14 to just 1.93. This is faster than a central FD and gives exact Jacobian-vector products. However, this hybrid SD/AD approach is still almost twice as expensive as one-sided finite differences. To obtain better performance we need to consider methods that perform some initial computations up front. Consequently, we next compare and contrast the various levels of *precomputed-storage* approaches that compute and store Jacobian sub-matrices at the edge level and then uses them to assemble repeated Jacobian-vector products. These methods require more storage but can significantly improve the computational performance of repeated Jacobian-vector product assemblies.

The *precomputed-storage Level-1* approach requires 24.8 times the cost of the residual assembly in order to precompute the edge-level matrices and stores 260 `doubles` per edge. Once these matrices are computed and stored, the relative CPU time for each subsequent Jacobian-vector product assembly is only 0.21 times the cost of a residual assembly. This is almost five times faster than a one-side FD approximation. Note that the relative CPU time of 24.8 is actually quite impressive considering that `TFad<40, double>` is used on the function  $\hat{r}(\dots)$  with 40 derivative components. Therefore, the cost for each derivative component is less than the cost of the function evaluation. The relative CPU time of 24.8 for the creation of these matrices, however, may still seem too high but in many situations where Jacobian-vector products are performed at a particular point  $\mathbf{w}$  (such as in an iterative

linear solver like GMRES in a Krylov–Newton method) the overall reduction in runtime over even one-side FDs can be significant. If the memory capacity is limited or the reduction in CPU is insufficient, additional symbolic differentiation can be considered where other Jacobian sub-matrices at the edge level are stored.

Now consider results for the *precomputed-storage Level-2* approach. The Jacobian sub-matrices for the dissipation term  $\mathbf{d}(\dots)$  in (35) and (36) are computed using `TFad<10, double>` (using 10 columns in the seed matrix). This approach results in dramatic reductions in relative CPU time for pre-computing edge-based Jacobian sub-matrices and storage of these sub-matrices over the *precomputed-storage Level-1* approach. The relative CPU time is reduced from 24.8 to 4.73 and the storage for the Jacobian sub-matrices is reduced from 260 to 80 `doubles` per edge. However, we only see a minor reduction in relative CPU time from 0.21 to 0.19 for subsequent Jacobian-vector product assemblies. Even though the reduction in Jacobian-vector product computations is not significantly reduced, the dramatic reductions in upfront cost and storage makes this scheme more attractive.

#### 4.6 Comparison of differentiation strategies for the dissipation term

We also experimented with several different approaches to automatically differentiate the Roe-type dissipation term  $\mathbf{d}(\dots)$  shown in (26) and also attempted to derive the Jacobians symbolically. It required approximately a month to derive expressions for the Jacobians of  $\mathbf{d}(\dots)$  in (35) and (36) and after approximately two weeks of implementation the resulting hand-derived and hand-coded Jacobian evaluation C++ code was 2.5 times slower than the automatically generated AD code (using `TFad<-double>`) and was not giving the correct Jacobians. The likely reason for the degradation in performance of the hand-coded Jacobian evaluation function is that there are many common expressions involved in the Jacobian entries that are automatically exploited in the AD code that were not exploited in the hand-coded derivative function. The attempt to symbolically derive and implement these Jacobians was therefore abandoned to avoid expending large amounts of additional, open-ended developer effort.

Here, results are presented for the use of several different automated strategies to compute the Jacobian of the dissipation term with respect to one of the states  $\partial\mathbf{d}/\partial\mathbf{w}_i \in \mathbf{R}^{(n_d+2) \times (n_d+2)}$  ( $n_d = 3$ , then the Jacobian is of size  $5 \times 5$ ). Note that the computation of both  $\partial\mathbf{d}/\partial\mathbf{w}_i$  and  $\partial\mathbf{d}/\partial\mathbf{w}_j$  are used in the *precomputed-storage level-2* method described above. However, the computation of only  $\partial\mathbf{d}/\partial\mathbf{w}_i$  involves many of the same issues as computing both these Jacobians together. Table III gives operation counts for the computation of  $\partial\mathbf{d}/\partial\mathbf{w}_i$  using one-sided finite difference (FD), automatic differentiation (AD) using `TFad<5, double>`, and the complex-step (CS) using `std::complex<double>`. These operation counts were generated automatically using a templated abstract data type called `ScalarFlopCounter`. Operation counts do not completely determine the CPU compute time of a code as many other issues must be considered, even for cache-resident data [Goedecker and Hoisie 2001]. However, these operation counts provide a basis for comparison of the various differentiation methods and for runtime performance using different compilers and different platforms. The difference in the ratios of relative operation counts and relative CPU times give a measure of how well a particular C++ compiler deals with abstract data types (that is the basis for the AD and CS methods in C++) with respect to built-in data types.

A number of interesting observations can be made in regard to the numerical results in Table III:

- (1) The dissipation term is 3.2 times more expensive than the two flux calculations required by the residual evaluation. We suspect the CPU times do not reflect the same ratio because the dissipation terms involves five square roots which are typically more expensive in comparison to other functions of similar operation counts.

Operation	$\mathbf{f}_i^{i+}$	$\mathbf{d}_{ij}$	FD $\partial\mathbf{d}/\partial\mathbf{w}_i$	AD $\partial\mathbf{d}/\partial\mathbf{w}_i$	CS $\partial\mathbf{d}/\partial\mathbf{w}_i$
=	14	65	446	819	2018
+	4	29	174	459	714
+=	6	24	149	114	609
unary +	0	7	42	3	22
-	1	14	115	85	620
-=	0	0	0	0	150
unary -	0	9	54	59	89
*	15	102	637	947	2707
*=	0	5	30	5	65
/	3	8	49	48	134
/=	0	0	0	0	40
sqrt	0	5	30	5	50
abs	0	0	0	0	75
>	0	3	18	3	38
<	0	3	18	3	53
==	0	0	0	0	45
all ops	43	274	1762	2550	7429
rel ops	0.16	1.0	6.4	9.3	27.1
rel CPU	0.11	1.0	6.5	10.9	19.5

Table III. Operation counts for the computation of the flux function  $\mathbf{f}_i^{i+} = \mathbf{f}(\mathbf{w}_{ij}^{i+}) \cdot \mathbf{n}_{ij}$ , the dissipation term  $\mathbf{d}_{ij}$  and the  $5 \times 5$  Jacobian  $\partial\mathbf{d}/\partial\mathbf{w}_i$  using one-sided finite difference (FD), automatic differentiation (AD) using `TFad<double>`, and the complex-step (CS) using `std::complex<double>`. These results were obtained using `g++ 3.2.2` on the same Red Hat Linux 7.2 platform shown in Table IV.

- (2) The actual operation counts for  $\partial\mathbf{d}/\partial\mathbf{w}_i$  are less than predicted by a total operation count of the  $\mathbf{d}_{ij}$  evaluation, because the code was carefully templated to avoid derivatives of inactive variables.
- (3) The relative operation count and the relative CPU time for the FD computation of  $\partial\mathbf{d}/\partial\mathbf{w}_i$  were nearly identical at 6.4 and 6.5 respectively. This suggests that the relative operation count in the FD case is a very good prediction of the relative CPU time, which seems reasonable since the majority of the computation in the FD method is performed in the same C++ function that computes  $\mathbf{d}_{ij}$ .
- (4) The AD computation of  $\partial\mathbf{d}/\partial\mathbf{w}_i$  requires 1.44 times the operations of the the FD method. The increased operation count is typical and expected for an AD method. Note that some individual derivative operations are more efficient than their function evaluations, in particular the square root operation.
- (5) As expected the CS method results in a significant increase in operation count over the AD method, as a result of extra (unnecessary) floating-point computations and recomputations of the value of the dissipation function  $\mathbf{d}$  repeatedly for each separate column of  $\partial\mathbf{d}/\partial\mathbf{w}_i$ .

These results are for a specific compiler on a particular platform, but it is important to evaluate the ability of different compilers to implement efficient C++ code for derived types and expression templates. Table IV shows relative CPU times on a variety of platforms for computing the  $5 \times 5$  Jacobian  $\partial\mathbf{d}/\partial\mathbf{w}_i$  using the FD, AD, and CS methods.

The relative CPU time of performing one-side finite differences varies by as much as 6.2 for ICL to 7.4 for SGI even through exactly the same C++ source code was compiled on each of these platforms with similar optimization levels. A relative CPU time of slightly greater than 6.0 for the one-sided finite difference would be ideal since six evaluations of  $\mathbf{d}(\dots)$  are performed (one for the base point and five others for each perturbation in  $\mathbf{w}_i$ ). The additional overhead is incurred in performing the

Compiler/platform	One-sided finite differences	Automatic differentiation	Complex step
g++	7.0	7.5	39.9
kcc	7.0	12.5	31.6
ICL	6.2	10.3	59.3
SGI	7.4	19.5	39.0
cxx	6.7	57.4	197.5

### Platforms

g++ : GNU g++ version 3.1, Red Hat Linux 7.2, 1.7 GHz Intel P IV

kcc : KAI C++ version 4.0e, Red Hat Linux 7.2, 1.7 GHz Intel P IV

ICL : Intel C++ version 5.0, MS Windows 2000, 1.0 GHz Intel P IV

SGI : MipsPro C++ version 7.3.1

cxx : Compaq C++ version, Sandia National Laboratories Cplint

Table IV. Relative CPU times for the computation of the  $5 \times 5$  Jacobian of the dissipation term  $\partial \mathbf{d} / \partial \mathbf{w}_i$  using one-sided finite difference (FD), automatic differentiation (AD) using `TFad<double>` and the complex-step (CS) using `std::complex<double>`. All CPU times are relative to evaluation of the dissipation term `d(...)`.

vector subtraction and scaling as shown in (4).

These results also show that the CS method using `std::complex<double>` is much less efficient than the AD method using `TFad<5, double>` on all of these platforms (as predicted by the relative increase in total operations). The greatest disparity between AD and CS occurred with the ICL compiler where the relative CPU time for CS was almost six times greater than for AD. The difference between the relative CPU times for CS and AD was almost twice that what would be predicted by the relative operation counts.

Finally, a large difference in performance is observed for the handling of the scalar types `TFad<double>` and `std::complex<double>` in comparison to `double`. The relative performance of `TFad<double>` varies from 7.5 for g++ 3.1 to 57.4 for cxx. The relative performance of 7.5 for g++ 3.1 was very good considering that the relative total operation count is 9.3 (as shown in Table III). The AD code actually exceeds what would be predicted from the ideal operation count in this one case.

## 5. CONCLUSIONS

A hybrid differentiation strategy is presented to compute Jacobian-vector products for an ANSI C++ implementation of a finite-volume discretization of the Euler equations. Non-invasive and invasive hybrid symbolic/AD strategies that pre-compute and store Jacobian sub-matrices at the mesh-object level and then assemble Jacobian-vector products may result in computations nearly five times faster than one-sided finite differences. In addition, these derivative computations are accurate to machine precision. No-storage use of AD at the edge level can result in derivative computations that are very competitive in speed with FD methods and require little knowledge of the actual edge-level computations. However, exploiting the structure of the computations at a lower level and combining symbolically derived and coded derivatives with AD resulted in significant improvements in CPU time and storage. Conclusions and observations are summarized as follows:

- (1) Hybrid symbolic/AD approaches are well suited to discretization methods for PDEs due to their structure. Hybrid symbolic/AD methods can result in very accurate and affordable derivative computations without having to rewrite an entire code. Instead, differentiation tools can be applied at the mesh object level in a general way.
- (2) Computing Jacobian-vector products and performing like computations by applying AD at the mesh object level generally does not require sophisticated AD features such as the special sparsity

and check-pointing handling for storage management that are the focus of several research groups [Bischof et al. 1997], [Griewank et al. 1996].

- (3) One of the main advantages of the hybrid strategy is the flexibility with which one can apply AD so that the right balance between implementation effort and computational efficiency can be achieved.
- (4) Any new or existing C++ project that is considering derivative computations should template on the scalar type as much computational C++ code as possible. Templating code by the scalar type not only allows the use of AD but other techniques such as interval analysis, extended precision arithmetic, complex arithmetic, and uncertainty quantification.
- (5) Symbolic differentiation is not always efficient or affordable even though theoretically such an approach should result in the best quality differentiation code. In many cases, too much implementation effort is necessary to provide symbolic derivatives that are more efficient than those generated by AD.
- (6) The complex-step method for differentiation is analogous to automatic differentiation but less efficient (and slightly less accurate). Therefore, since many good AD classes for C++ are available, the complex-step method for the computation of derivatives should never be seriously considered for production use in ANSI C++ codes. However, as an extra validation approach, the complex-step method may be very reasonable. In other languages that lack support for operator overloading but yet support complex arithmetic, such as Fortran 77, the complex-step method is a much more attractive alternative.

## REFERENCES

- BEDA, L. M., KOROLEV, L. N., SUKKIKH, N. V., AND FROLOVA, T. S. 1959. Programs for automatic differentiation for the machine BESM. Tech. rep., Institute for Precise Mechanics and Computation Techniques, Academy of Science.
- BISCHOF, C., CARLE, A., CORLISS, G., GRIEWANK, A., AND HOVLAND, P. 1992. ADIFOR - generating derivative codes from Fortran programs. *Scientific Programming* 1, 1–29.
- BISCHOF, C. H., ROH, L., AND MAUER, A. 1997. ADIC — An extensible automatic differentiation tool for ANSI-C. *Software—Practice and Experience* 27, 12, 1427–1456.
- BLAZEK, J. 2001. *Computational Fluid Dynamics: Principles and Applications*. Elsevier.
- CESARE, N. AND PIRONNEAU, O. 2000. Flow control problem using automatic differentiation in C++. Tech. rep., Unversite Pierre et Marie Curie, LAN-UPMC report 99013.
- COURTY, F., DERVIEUX, A., KOOBUS, B., AND HASCOET, L. 2003. Reverse automatic differentiation for optimum design: from adjoint state assembly to gradient computation. *Optimization Methods and Software* 18, 5, 615–627.
- FAURE, C. 2005. An automatic differentiation platform: Odyssee. *Future Generation Computer Systems* 21, 8, 1391–1400.
- GIERING, R. AND KAMINSKI, T. 1998. Recipes for adjoint code construction. *ACM Trans. Math. Software* 24, 4, 437–474.
- GOEDECKER, S. AND HOISIE, A. 2001. *Performance Optimization of Numerically Intensive Codes*. SIAM.
- GRIEWANK, A. 2000. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. SIAM.
- GRIEWANK, A., JUEDES, D., AND UTKE, J. 1996. ADOL—C, a package for the automatic differentiation of algorithms written in C/C++. *ACM Trans. Math. Software* 22, 2, 131–167.
- HASCOET, L. 2004. Tapenade: a tool for automatic differentiation of programs. In P. Neittaanm aki, T. Rossi, S. Korotov, E. Onate, J. Periaux, and D. Knorzer, editors, *4th European Congress on Computational Methods in Applied Sciences and Engineering (ECCOMAS) 2*.
- LINNAINMAA, S. 1976. Taylor expansion of the accumulated rounding error. *BIT (Nordisk Tidskrift for Informationsbehandling)* 16, 146–160.
- ACM Transactions on Mathematical Software, Vol. V, No. N, September 2007.

- MARTINS, J. R. R. A., STURDZA, P., AND ALONSO, J. J. 2003. The complex-step derivative approximation. *ACM Trans. Math. Softw.* 29, 3, 245–262.
- MOORE, R. 1979. *Methods and applications of interval analysis*. SIAM.
- ROE, P. 1981. Approximate riemann solvers, parameter vectors, and difference schemes. *Journal of Computational Physics* 43, 357–372.
- SPEELPENNING, B. 1980. Compiling fast partial derivatives of functions given by algorithms. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.
- SQUIRE, W. AND TRAPP, G. 1998. Using complex variables to estimate derivatives of real functions. *SIAM Rev.* 40, 1, 110–112.
- VAN ALBADA B. VAN LEER, G. AND JR., W. R. 1982. A comparative study of computational methods in cosmic gas dynamics. *Astronomy and Astrophysics* 108, 76–84.
- VANDEN, K. AND ORKWIS, P. 1996. Comparison of numerical and analytical jacobians. *AIAA Journal* 34, 6, 1125–1129.
- WENGERT, R. E. 1964. A simple automatic derivative evaluation program. *Comm. ACM* 7, 8, 463–464.

Received: ???; revised: ???; accepted: ???