Received: 13 February 2019 | Revised: 26 June 2019 | Accepted: 2 August 2019
DOI: 10.1002/widm.1334

OVERVIEW

WIREs DATA MINING AND KNOWLEDGE DISCOVERY WILEY

An introduction to algorithmic differentiation

Assefaw H. Gebremedhin[1] | **Andrea Walther**[2]

[1]School of Electrical Engineering and Computer Science, Washington State University, Pullman, Washington

[2]Institute for Mathematics, Paderborn University, Paderborn, Germany

Correspondence
Assefaw H. Gebremedhin, School of Electrical Engineering and Computer Science, Washington State University, Pullman, WA, 99164, USA.
Email: assefaw.gebremedhin@wsu.edu

Funding information
National Science Foundation, Grant/Award Number: IIS-1553528

Abstract

Algorithmic differentiation (AD), also known as automatic differentiation, is a technology for accurate and efficient evaluation of derivatives of a function given as a computer model. The evaluations of such models are essential building blocks in numerous scientific computing and data analysis applications, including optimization, parameter identification, sensitivity analysis, uncertainty quantification, nonlinear equation solving, and integration of differential equations. We provide an introduction to AD and present its basic ideas and techniques, some of its most important results, the implementation paradigms it relies on, the connection it has to other domains including machine learning and parallel computing, and a few of the major open problems in the area. Topics we discuss include: forward mode and reverse mode of AD, higher-order derivatives, operator overloading and source transformation, sparsity exploitation, checkpointing, cross-country mode, and differentiating iterative processes.

This article is categorized under:

Algorithmic Development > Scalable Statistical Methods

Technologies > Data Preprocessing

KEYWORDS

adjoints, algorithmic differentiation, automatic differentiation, backpropagation, checkpointing, sensitivities

1 | INTRODUCTION

Efficient calculation of derivative information is an indispensable building block in numerous applications ranging from methods for solving nonlinear equations to sophisticated simulations in unconstrained and constrained optimization. Suppose a sufficiently smooth function $F : \mathbb{R}^n \to \mathbb{R}^m$ is given. There are a few different techniques one can use to compute, either exactly or approximately, the derivatives of F. In comparing and contrasting the different conceivable approaches, it is necessary to take into account the desired accuracy and the computational effort involved.

The *finite difference* method is a very simple way to obtain approximations of derivatives. Following a truncated Taylor expansion analysis, all that is needed for estimating derivatives using finite differences is function evaluations at different arguments, taking the difference and dividing by a step-length (see Quarteroni, Sacco, & Saleri, 2000, Chapter 10.10 for details on this class of methods). The runtime complexity of approximating derivatives using finite differences grows linearly with n, that is, the number of unknowns, which is quite often prohibitively expensive. So the finite difference method is simple, but inexact and slow. On the other hand, we should also note that some optimization algorithms are not that much affected by the inexactness of the derivative information provided by a finite difference method.

Alternatively, one can derive derivative expressions in a *symbolic fashion* using, for example, computer algebra tools. This approach provides closed-form expressions for the derivative calculation. However, it has difficulties in exploiting common subexpressions, which for complicated functions results in expensive derivative calculation in terms of runtime. Furthermore, it is difficult to apply the symbolic approach if the function to be differentiated is not available in a closed form, as for example is the case when one has code segments involving branches or one uses iterative solvers. At a high level, a variant of symbolic differentiation is also used in a class of strategies for constrained optimization that are known as *optimize-then-discretize*. In such methods, in the context of optimization tasks with partial differential equations as constraints, the sensitivity differential equation or the adjoint differential equation are derived in the corresponding function spaces using analytic optimality conditions (Gunzburger, 2003; Tröltzsch, 2010). Although this approach is elegant from a mathematical point of view, it has the drawback that the process of deriving the expressions for the derivatives is error prone. More importantly, the obtained derivative information may be inconsistent, leading for example to difficulties in the optimization process (Abraham, Behr, & Heinkenschloss, 2004).

Algorithmic differentiation (AD), in contrast to finite differences, provides *exact* derivative information about a function F given in a high-level programming language and it does so with time and space complexity that can be bounded by the complexity of evaluating the function itself. For example, using the reverse mode of AD it is possible to compute the gradient of a scalar-valued function with runtime cost of less than four function evaluations. In this Overview Article, we provide a gentle introduction to AD touching on a variety of its main topics and point out its connections to machine learning and parallel computing. Comprehensive introductions to AD can be found in the books (Griewank & Walther, 2008; Naumann, 2012).

A loosely related method for computing derivatives worth mentioning here is the *complex-step* method, first introduced in Squire and Trapp (1998) and brought to renewed attention by Nicholas Higham's June 2018 SIAM News article titled "Differentiation With(out) a Difference." The complex-step method can be interpreted as one variant of the forward mode of AD, which we will discuss in detail in a later section. To obtain the same accuracy as the forward mode AD, however, it is important that the evaluation of the function involves only real numbers (not complex numbers), since the inclusion of complex numbers would disturb the process.

1.1 | Key idea of AD and overview of the paper

The key idea in AD is the systematic application of the chain rule of calculus. To that end, the computation of the function F is decomposed into a (typically long) sequence of simple evaluations, for example, additions, multiplications, and calls to elementary functions such as $\sin(v)$ or $\cos(v)$. The derivatives with respect to the arguments of these simple operations can be easily calculated. A systematic application of the chain rule then yields the derivatives of the entire sequence with respect to the input variables $x \in \mathbb{R}^n$.

Over the last several decades, extensive research activities have led to a thorough understanding of the basic modes of AD. As detailed in Section 3, the *forward mode* of AD provides one column of the Jacobian $F^{'}(x)$ with a time complexity equal to no more than three function evaluations. The time needed to compute one row of the Jacobian $F^{'}(x)$, for example, the gradient of a scalar-valued component function of F, is less than four function evaluations using the *reverse mode* of AD in its basic form. Importantly, this bound for the reverse mode is completely independent of the number of input variables. This is highly attractive in contexts where there are many inputs—for example, design parameters, weights of neural networks and the like—and a scalar-valued function such as a scalar loss function to be minimized.

Along with the progress on theoretical foundation, numerous AD software tools have also been developed over the years. Many of these tools have matured to a state where they can be used to provide derivatives for large and unstructured codes. The implementation paradigms underlying most existing AD tools are source transformation and operator overloading. These paradigms are discussed in Sections 4.1 and 4.2 and a list of sample AD tools built around them is provided in Section 4.3. It should, however, also be noted that there are numerous software tools that are designed for other purposes but provide built-in AD support. There also are recent efforts around AD for emerging programming languages and environments that do not necessarily clearly fit into the traditional implementation paradigms. We provide in Section 4.4 a brief discussion of such higher-level AD implementations.

AD has interesting connections to machine learning that are not yet fully exploited. We highlight these connections in Section 5 and give in the same section an overview of machine learning libraries that have built-in derivative computation capabilities based on AD ideas.

In most large-scale applications, the first- and second-order derivative matrices—Jacobian and Hessian matrices—needed in the computation are sparse. Great savings in both runtime and memory can be attained by exploiting the sparsity structure in the computation of these matrices via AD. One way to achieve this is using a *compress-then-recover* strategy. This framework is briefly reviewed in Section 6.

As mentioned earlier, the reverse mode of AD has a very attractive time complexity for gradient computation, but it comes at a cost of increased memory. Checkpointing strategies developed over the years to strike a happy medium between these two pulls are reviewed in Section 7. The potential use of parallel computing for speeding up AD computations and the challenge of differentiating codes written in parallel computing settings is discussed briefly in Section 8. We conclude the paper in Section 9 by drawing attention to a few of the major open problems in AD.

1.2 | Historical note

Although powerful computers started to become widely available only a few decades ago, AD in contrast has quite a long history. One might argue that even Newton and Leibniz seem to envisage applying the derivative calculations to numbers given their treatment of differential equations. Since then both modes of AD have been rediscovered and reinvented several times making it difficult to give the credit to a specific publication. The backward error analysis of Wilkinson in the beginning of the 1960s certainly bears similarities with the reverse mode of AD. Shortly thereafter the fast development of compiler technology pushed the algorithmic transformation of programs to compute derivatives. Early steps in this direction, using forward mode, were taken already around 1960. The tremendous potential of the reverse mode was discovered independently by several researchers in the 1970s. Since then, AD has seen extensive development with respect to theoretical foundation as well as tool development. The history of the reverse mode is reviewed thoroughly in Griewank (2012).

1.3 | A note on references and notations

For much of the technical presentations and formal statements in this paper, we follow the notations in the book (Griewank & Walther, 2008). In most of the sections, we have made effort to cite original papers introducing an approach for the first time, but in the interest of space, we avoid citing (otherwise worthy) further and more recent developments. In some cases, we refer the reader to the book (Griewank & Walther, 2008) instead. To make the material more accessible, we include plenty of examples and figures. Finally, we note that the AD-community website www.autodiff.org provides an up-to-date overview of AD publications and a fairly comprehensive list of available AD tools.

2 | FUNCTION REPRESENTATION

Throughout this paper, we consider only functions $F:\mathbb{R}^n \to \mathbb{R}^m, y = F(x)$, whose evaluation at a specific argument can be represented by an evaluation procedure as introduced in (Griewank & Walther, 2008, Chapter 2). That is, all quantities calculated during the function evaluation are numbered such that

$$\left[\underbrace{v_{1-n}, \ldots, v_0}_{x}, v_1, \ldots, v_{l-m-1}, v_{l-m}, \underbrace{v_{l-m+1}, \ldots, v_l}_{y}\right].$$

Each value $v_i \in \mathbb{R}$ is obtained by applying an elemental function φ_i to some set of arguments $v_j, j < i$, that is,

$$v_i = \varphi_i\left(v_j\right)_{j \prec i}, \qquad \varphi_i : \mathbb{R}^{n_i} \to \mathbb{R}.$$

Here, n_i denotes the number of arguments required to evaluate φ_i. In most cases, n_i equals either one or two, but n_i could be larger when, for example, vector operations like dot products are considered. The precedence relation $j \prec i$ denotes that v_i depends directly on v_j and $j \prec i$ implies $j < i$ just for simplicity. More complex situations could also be covered but since they complicate the discussion without adding any benefit we will ignore them. Hence, the function evaluation can always be described by an evaluation procedure of the general form given in Table 1. Using the induction principle, then one can show that the overall function $F:\mathbb{R}^n \to \mathbb{R}^m$ is differentiable up to a certain order on a given open domain $D \subset \mathbb{R}^n$, if the elemental functions comprising the evaluation of F are differentiable up to the same order with respect to their resulting arguments. This fact is exploited heavily by AD.

$v_{i-n} = x_i$	$i = 1,\ldots,n$
$v_i = \varphi_i(v_j)_{j \prec i}$	$i = 1, \ldots, l$
$y_{m-i} = v_{l-i}$	$i = 0,\ldots,m-1$

TABLE 1 General evaluation procedure for a function $F:\mathbb{R}^n \to \mathbb{R}^m$

v_{-2}	=	x_1
v_{-1}	=	x_2
v_0	=	x_3
v_1	=	$v_{-2} + v_{-1}$
v_2	=	$\exp(v_1)$
v_3	=	$\sin(v_1)$
v_4	=	v_0^2
v_5	=	$v_2 + v_3$
v_6	=	$v_4 * v_1$
y_1	=	v_5
y_2	=	v_6

```
double x1,x2,x3;
double v1,v2,v3,v4,v5,v6;
double y1, y2;
...
v1 = x1+x2;
v2 = exp(v1);
v3 = sin(v1);
v4 = x3*x3;
v5 = v2 + v3;
v6 = v4 * v1;
y1 = v5;
y2 = v6;
```

FIGURE 1 Evaluation procedure and real C code for the function in Example 2.1

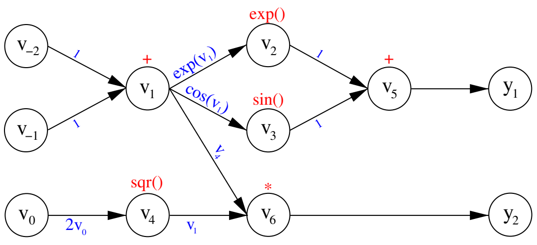

FIGURE 2 Computational graph for Example 2.1. The vertices v_1 through v_6 corresponding to intermediate variables have been annotated by elemental functions (operations). The edges are annotated with weights indicating local partial derivatives (i.e., for an edge (v_j,v_i), the edge weight is $\frac{\partial v_i}{\partial v_j}$)

Example 2.1 *Throughout the paper, we will use the function*

$$F:\mathbb{R}^3 \to \mathbb{R}^2, F(x) = \begin{pmatrix} \exp(x_1+x_2)+\sin(x_1+x_2) \\ x_3^2 \cdot (x_1+x_2) \end{pmatrix}$$

as a running example to illustrate the principles of AD. The evaluation procedure for this function is given on the left panel in Figure 1. *A real C code for evaluating the function is displayed in the right panel in the same figure.*

Furthermore, a *computational graph* is often used as an alternative representation of a function evaluation. Here, the roots of the directed acyclic graph represent the independent variables, the leaves represent the dependent variables and all other vertices between the two correspond to intermediate variables. The edges describe the dependency relationship between the vertices. For the function in Example 2.1, the computational graph is shown in Figure 2. Sometimes, the edges of the computational graph are weighted so that they show the local partial derivatives, that is, the partial derivative of the target vertex with respect to the source vertex. This is illustrated in Figure 2 by the blue weights annotating the edges. The edge weight information is for example used in an implementation of AD using a trace as discussed in Section 4.2.

3 | FORWARD MODE, REVERSE MODE, AND THEIR COMBINATION

The basic approaches of AD are the forward mode and the reverse mode. We describe in detail in Section 3.1 and 3.2 the use of these modes for computing first-order derivatives and briefly discuss the use of the modes for higher order derivatives in Section 3.3. There have also been attempts to develop a "cross-country" mode for more efficient evaluation of Jacobians (discussed in Section 3.4). A variant of this mode is used by some AD tools in the form of statement-level preaccumulation to improve the efficiency of the derivative calculation, but the mode has not yet been well established for the efficient calculation of general-purpose Jacobians.

3.1 | Forward mode of AD

As mentioned in the Introduction, the main idea of AD is the application of the chain rule. In a very simple approach, derivatives can be propagated together with the evaluation of the function during the execution of a given code segment. From a mathematical point of view, this can be interpreted as the computation of tangent information in the following way. Assume for a moment that the argument vector x represents the value of a time-dependent path $x(t)$ at $t = 0$. Then, this time-dependent path defines a tangent $\dot{x} = \partial x(t)/\partial t\,|_{t=0}$. Using the forward mode of AD, one computes the resulting tangent $\dot{y}$ obtained for the time-dependent path $F(x(t))$ that is well defined if the function F is sufficiently smooth, that is,

$$\dot{y} = \frac{\partial}{\partial t} F(x(t))|_{t=0} = F'(x(t))\frac{\partial x(t)}{\partial t}|_{t=0} = F'(x)\dot{x} \equiv \dot{F}(x,\dot{x}).$$

This interpretation also explains the alternative names sometimes used for the forward mode of AD, namely tangent mode or tangent-linear mode. The underlying idea is illustrated (geometrically) in Figure 3.

When this idea is applied to an elemental function $\varphi_i : \mathbb{R}^{n_i} \to \mathbb{R}$, $v_i = \varphi_i(u_i)$, we have

$$\dot{\varphi}_i : \mathbb{R}^{2n_i} \to \mathbb{R}, \quad \dot{v}_i = \nabla\varphi_i(u_i)\dot{u}_i \equiv \dot{\varphi}_i(u_i,\dot{u}_i), \quad \text{that is,}$$

$$\dot{v}_i = \sum_{j \prec i} \frac{\partial}{\partial v_j}\varphi_i(u_i) * \dot{v}_j,$$

where we use the abbreviations $u_i = (v_j)_{j \prec i}$ and $\dot{u}_i = \left(\dot{v}_j\right)_{j \prec i}$. For a multiplication $v = u * w$, for example, the tangent operation is given by

$$v = \varphi(u,w) = u * w, \qquad \dot{v} = \dot{\varphi}(u,w,\dot{u},\dot{w}) = \dot{u} * w + u * \dot{w}.$$

Combining the resulting derivative information with the other differentiated elemental functions using the chain rule, one obtains the tangent for a given code segment, that is, the overall derivative $F'(x)\dot{x}$.

For an efficient implementation of a function evaluation, it is essential to reuse memory. Let us consider as example again a multiplication to illustrate the challenges that arise in the case of forward mode AD. Assume that the result of the multiplication $v = u * w$, is stored in the memory location of one of the arguments, say u. Then, the corresponding derivative calculation $\dot{v} = \dot{u} * w + u * \dot{w}$ obviously would yield the wrong value if it is executed after the evaluation of the product since the original value of u is no longer available. However, the derivative calculation is exactly the same if performed before the computation of v then giving the correct value $\dot{v}$. In other cases, values computed during the evaluation of an elemental function can be reused for the derivative calculation as for example if $\varphi(u) = \exp(u)$. For these reasons, it is advantageous to evaluate φ_i and $\dot{\varphi}_i$ simultaneously, sharing intermediate results. We denote this by

$$[v_i, \dot{v}_i] = [\varphi_i(u_i), \dot{\varphi}_i(u_i,\dot{u}_i)].$$

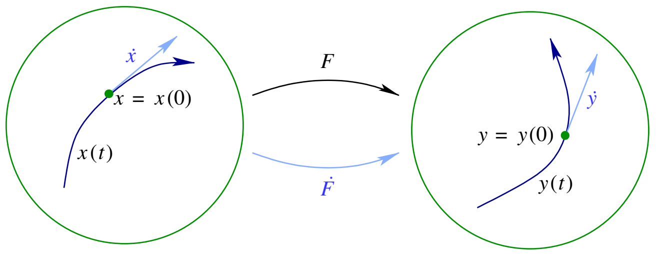

FIGURE 3 Idea of forward mode algorithmic differentiation: Mapping of variable x to variable y by function F and tangent $\dot{x}$ to tangent $\dot{y}$ by function $\dot{F}$

Based on these tangent operations, we obtain a tangent procedure for the computation of $\dot{y} = F'(x)\dot{x}$ as illustrated in Table 2.

Instead of propagating just one vector of derivatives in the tangent procedure as illustrated in Table 2, one could also apply the same idea to an entire matrix $\dot{X} \in \mathbb{R}^{n\times p}$. This approach, which is called *vector forward mode of AD*, yields the matrix $\dot{Y} \in \mathbb{R}^{m\times p}$ as a result. The vector forward mode of AD allows the reuse of the intermediate values calculated for the evaluation of y, a fact reflected in the complexity result stated in Theorem 3.1. For the derivation of this result, the number of operations required to compute $\dot{y}$ and $\dot{Y}$, respectively, are counted. The stated memory requirement follows immediately from Table 2.

Theorem 3.1 *(Complexity Results for Forward Mode AD). For a function $F:\mathbb{R}^n \to \mathbb{R}^m$ and given vectors x, $\dot{x} \in \mathbb{R}^n$, the number of operations required to compute the Jacobian-vector product $F'(x)\dot{x}$ is bounded by*

$$\mathrm{OPS}(F'(x)\dot{x}) \le 3\cdot\mathrm{OPS}(F(x)). \tag{1}$$

The memory requirement is given by

$$\mathrm{RMEM}(F'(x)\dot{x}) = 2\cdot\mathrm{RMEM}(F(x)), \tag{2}$$

where RMEM denotes the amount of random access memory.

For a given vector $x \in \mathbb{R}^n$ and a matrix $\dot{X} \in \mathbb{R}^{n\times p}$, the number of operations required to compute the Jacobian-matrix product $F'(x)\dot{X}$ is bounded by

$$\mathrm{OPS}\left(F'(x)\dot{X}\right) \le (1+2p)\cdot\mathrm{OPS}(F(x)) \tag{3}$$

and the memory requirement is given by

$$\mathrm{RMEM}\left(F'(x)\dot{X}\right) = (1+p)\cdot\mathrm{RMEM}(F(x)). \tag{4}$$

For sharper statements on the bounds and for proofs, see Griewank and Walther (2008, Chapter 3). Using an upper bound on the time needed to execute one operation, a similar upper bound can be obtained for the time needed to compute $F'(x)\dot{x}$ and $F'(x)\dot{X}$, respectively. Note that the "1" on the right-hand side of Equation (3) stems from the fact that the intermediate results computed during the function evaluation (just one) can be used for the derivative computations for all of the p directions. This exploitation of common results reduces the upper bound on the number of operations for computing the product $F'(x)\dot{X}$ from $3p$ OPS($F(x)$) to $(1+2p)$ OPS($F(x)$). This considerable reduction is especially important when p is large, for example, for the computation of whole Jacobian matrices using the forward mode of AD.

Example 3.2 (Forward mode applied to our basic example). *Applying the forward mode AD to the function defined in Example* 2.1, *the resulting tangent procedure to evaluate $\dot{y} = F'(x)\dot{x} \in \mathbb{R}^2$ for given vectors $x, \dot{x} \in \mathbb{R}^3$ is shown in Table* 3. *There, the evaluation of v_i is performed together with the evaluation of $\dot{v}_i$. Hence, the procedure is executed line by line.*

3.2 | Reverse mode of AD

Instead of being propagated together with the function evaluation, the derivatives can also be computed backwards starting from the function value y toward the independent variables x after a complete function evaluation. Once more, a corresponding

$[v_{i-n}, \dot{v}_{i-n}] = [x_i, \dot{x}_i]$	$i = 1,\ldots,n$
$[v_i, \dot{v}_i] = [\varphi_i(u_i), \dot{\varphi}_i(u_i, \dot{u}_i)]$	$i = 1,\ldots,l$
$[y_{m-i}, \dot{y}_{m-i}] = [v_{l-i}, \dot{v}_{l-i}]$	$i = 0,\ldots,m-1$

TABLE 2 The tangent procedure for computing the first derivative of a function $F:\mathbb{R}^n \to \mathbb{R}^m$

TABLE 3 The tangent procedure for Example 2.1

$v_{-2} = x_1$	$\dot{v}_{-2} = \dot{x}_1$
$v_{-1} = x_2$	$\dot{v}_{-1} = \dot{x}_2$
$v_0 = x_3$	$\dot{v}_0 = \dot{x}_3$
$v_1 = v_{-2} + v_{-1}$	$\dot{v}_1 = \dot{v}_{-2} + \dot{v}_{-1}$
$v_2 = \exp(v_1)$	$\dot{v}_2 = \exp(v_1)*\dot{v}_1$
$v_3 = \sin(v_1)$	$\dot{v}_3 = \cos(v_1)*\dot{v}_1$
$v_4 = v_0^2$	$\dot{v}_4 = 2*v_0*\dot{v}_0$
$v_5 = v_2 + v_3$	$\dot{v}_5 = \dot{v}_2 + \dot{v}_3$
$v_6 = v_4 * v_1$	$\dot{v}_6 = \dot{v}_4*v_1 + v_4*\dot{v}_1$
$y_1 = v_5$	$\dot{y}_1 = \dot{v}_5$
$y_2 = v_6$	$\dot{y}_2 = \dot{v}_6$

interpretation from a mathematical point of view is possible. Consider for a given vector $\bar{y} \in \mathbb{R}^m$ and a given value $c \in \mathbb{R}$ the hyperplane $\{y \in \mathbb{R}^m | \bar{y}^\top y = c\}$ in the range of F. The hyperplane has the inverse image $\{x \in \mathbb{R}^n | \bar{y}^\top F(x) = c\}$. By the implicit function theorem, this set is a smooth hypersurface with the normal $\bar{x}^\top = \bar{y}^\top F'(x) \equiv \bar{F}(x,\bar{y})$ at x, provided $\bar{x}^\top$ does not vanish. Geometrically, $\bar{y}$ and $\bar{x}$ can be seen as normals or cotangents as illustrated in Figure 4. It is important to note that the vector-Jacobian product $\bar{y}^\top F'(x)$ cannot be approximated directly by finite differences.

To implement the reverse mode of AD, one obtains for each elemental function $\varphi_i : \mathbb{R}^{n_i} \to \mathbb{R}$, the adjoint function

$$\bar{u}_i + = \bar{v}_i * \nabla\varphi_i(u_i) \quad \text{for } \bar{u}_i \equiv \left(\bar{v}_j\right)_{j \prec i} \in \mathbb{R}^{n_i} . \tag{5}$$

In the forward mode, one can take care of the reuse of memory by rearranging the order of evaluating φ_i and $\dot{\varphi}_i$ correspondingly, as described earlier for a multiplication in more detail. However, since v_i is computed during the function evaluation and $\bar{u}_i$ on the way backwards, such rearrangements no longer help in the case of the reverse mode. Therefore, one very simple strategy for maintaining all values needed for the derivative calculation is to store and restore them on a strictly sequentially accessed data structure. To do so, the evaluation of the elemental function and its adjoint is modified to

$$\begin{array}{ll} \text{push}(v) & v \leftarrow \text{pop}() \\ v = \varphi(u) & \bar{u} + = \bar{v} * \nabla\varphi(u) \\ & \bar{v} = 0 \\ \text{elemental function} & \text{adjoint function} \end{array} . \tag{6}$$

Here, the push(v) operation stores the value of v onto a stack-like data structure, whereas the $v \leftarrow$ pop() operation retrieves a value from the same data structure and stores this value in the variable v. This notation is borrowed from the AD tool Tapenade, where one finds exactly these operations in the generated source code. Note that the popped value of v may be part of the vector u required to evaluate the correct adjoint value $\bar{u}$. The nullifying of the adjoint value $\bar{v}$ is necessary for potential later updates, in the case of overwrites. This yields the adjoint procedure illustrated in Table 4 for the computation of $\bar{x}^\top = \bar{y}^\top F'(x)$.

Once more, one could also propagate a matrix $\bar{Y} \in \mathbb{R}^{m \times q}$ instead of just a vector $\bar{y} \in \mathbb{R}^m$ backwards yielding the *vector reverse mode of AD* to compute $\bar{X}^\top = \bar{Y}^\top F'(x) \in \mathbb{R}^{n \times q}$. Using again a complexity analysis based on a count of the additional

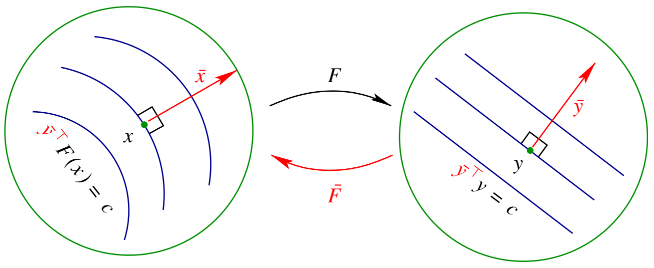

FIGURE 4 Idea of reverse mode of algorithmic differentiation: Mapping of variable x to variable y by function F and normal $\bar{y}$ to normal $\bar{x}$ by function $\bar{F}$

$\bar{v}_i$	$=$	0	$i = 1,\ldots,l$
$[v_{i-1}, \bar{v}_{i-1}]$	$=$	$[x_i, \bar{x}_i]$	$i = 1,\ldots,n$
push(v_i)			
v_i	$=$	$\varphi_i(u_i)$	$i = 1,\ldots,l$
$y_{m\text{-}1}$	$=$	$v_{l\text{-}i}$	$i = 0,\ldots,m-1$
$\bar{v}_{l-i}$	$=$	$\bar{y}_{m-i}$	$i = 0,\ldots,m-1$
$v_i \leftarrow$ pop()			
$\bar{u}_i$	$+=$	$\bar{v}_i * \nabla\varphi_i(u_i)$	$i = l,\ldots,1$
v_i	$=$	0	
$\bar{x}_i$	$=$	$\bar{v}_{i-n}$	$i = 1,\ldots,n$

TABLE 4 The adjoint procedure for computing the first derivative of a function $F: \mathbb{R}^n \to \mathbb{R}^m$

operations introduced by the adjoint operations, one obtains the following complexity results for the (vector) reverse mode of AD.

Theorem 3.3 *(Complexity result for reverse mode AD). For a function $F: \mathbb{R}^n \to \mathbb{R}^m$, given vectors $x \in \mathbb{R}^n$ and $\bar{y} \in \mathbb{R}^m$, the number of operations required to compute the vector-Jacobian product $\bar{y}^\top F'(x)$ is bounded by*

$$\mathrm{OPS}\left(\bar{y}^\top F'(x)\right) \le 4 \cdot \mathrm{OPS}(F(x)). \tag{7}$$

The memory requirement is given by

$$\mathrm{RMEM}\left(\bar{y}^\top F'(x)\right) = 2 \cdot \mathrm{RMEM}(F(x)), \quad \mathrm{SMEM}\left(\bar{y}^\top F'(x)\right) \approx \mathrm{OPS}(F(x)), \tag{8}$$

where SMEM denotes the amount of sequentially accessed memory.

For a given vector $x \in \mathbb{R}^n$ and a matrix $\bar{Y} \in \mathbb{R}^{m \times q}$, the number of operations required to compute the matrix-Jacobian product $\bar{Y}^\top F'(x)$ is bounded by

$$\mathrm{OPS}\left(\bar{Y}^\top F'(x)\right) \le \left(\frac{3}{2} + \frac{5}{2}q\right) \cdot \mathrm{OPS}(F(x)) \tag{9}$$

and the memory requirement is given by

$$\mathrm{RMEM}\left(\bar{Y}^\top F'(x)\right) = (1+q) \cdot \mathrm{RMEM}(F(x)), \quad \mathrm{SMEM}\left(\bar{Y}^\top F'(x)\right) \approx \mathrm{OPS}(F(x)). \tag{10}$$

For sharper statements on the bounds and for proofs, see Griewank and Walther (2008, Chapter 3). Once more, using an upper bound on the time needed to execute one operation, one obtains a similar upper bound also for the time needed to compute $\bar{y}^\top F'(x)$ and $\bar{Y}^\top F'(x)$, respectively. Furthermore, now the constant "$\frac{3}{2}$" on the right-hand side of Equation (9) reflects the fact that some results computed during the derivative evaluation can be used for adjoint computations for all q directions. This exploitation of common results reduces the upper bound on the number of operations for computing the product $\bar{Y}^\top F'(x)$ from $4q$ OPS($F(x)$) to ($3/2 + 5q/2$) OPS($F(x)$), which is especially important when q is large, for example, when computing the whole Jacobian matrix using the reverse mode of AD.

Example 3.4 (Reverse mode AD for our basic example). *Applying the scalar reverse mode of AD to the function defined in Example 2.1 one obtains the adjoint procedure listed in Table 5 to evaluate $\bar{x}^\top = \bar{y}^\top F'(x) \in \mathbb{R}^3$ for given vectors*

TABLE 5 The adjoint procedure for Example 2.1

$v_{-2} = x_1$	$\bar{v}_5 = \bar{y}_1$
$v_{-1} = x_2$	$\bar{v}_6 = \bar{y}_2$
$v_0 = x_3$	$\bar{v}_1 \mathrel{+}= v_4 * \bar{v}_6$
$v_1 = v_{-2} + v_{-1}$	$\bar{v}_4 \mathrel{+}= v_1 * \bar{v}_6$
$v_2 = \exp(v_1)$	$\bar{v}_2 \mathrel{+}= \bar{v}_5$
$v_3 = \sin(v_1)$	$\bar{v}_3 \mathrel{+}= \bar{v}_5$
$v_4 = v_0^2$	$\bar{v}_0 \mathrel{+}= 2 * v_0 * \bar{v}_4$
$v_5 = v_2 + v_3$	$\bar{v}_1 \mathrel{+}= \cos(v_1) * \bar{v}_3$
$v_6 = v_4 * v_1$	$\bar{v}_1 \mathrel{+}= \exp(v_1) * \bar{v}_2$
$y_1 = v_5$	$\bar{v}_{-1} \mathrel{+}= \bar{v}_1$
$y_2 = v_6$	$\bar{v}_{-2} \mathrel{+}= \bar{v}_1$
	$\bar{x}_1 = \bar{v}_{-2}$
	$\bar{x}_2 = \bar{v}_{-1}$
	$\bar{x}_3 = \bar{v}_0$

$x \in \mathbb{R}^3, \bar{y} \in \mathbb{R}^2$. *We have omitted the zeroing out of all* $\bar{v}_i$ *at the beginning and during the evaluation for brevity. Furthermore, we deleted the push and pop operations since we did not use overwrites in this small example such that the storing and retrieving of the intermediate results is not necessary.*

To compute the adjoint values $\bar{x}_i$*, one first evaluates the left column to compute the intermediate values. We assume that all of them are kept in memory. Otherwise, push operations would be needed. Subsequently, the available intermediate results like* v_4 *and* v_1 *are used during the evaluation of the right column to obtain the adjoint values.*

3.3 | Higher-order and other derivative information

As detailed in the previous two subsections, the forward mode and the reverse mode of AD yield first-order derivatives. Based on these ideas, it is also possible to compute higher order derivatives. As a first alternative, one may apply a forward mode differentiation to an already differentiated code in reverse mode. This yields for the input variables $x, \dot{x} \in \mathbb{R}^n$ and $\bar{y}, \dot{\bar{y}} \in \mathbb{R}^m$ the results

$$\dot{y} = F'(x)\dot{x} \quad \text{and} \quad \dot{\bar{x}}^\top = \bar{y}^\top F''(x)\dot{x} + \dot{\bar{y}}^\top F'(x) \ ,$$

and therefore also second-order derivatives. For example, for $m = 1$, $\bar{y} = 1$, $\dot{\bar{y}}^\top = 1$, and $\dot{x} = e_i$ (the ith unit vector in $\mathbb{R}^n$), the vector $\dot{\bar{x}}^\top$ represents the ith column of the Hessian $F''(x)$. For sparse Hessian matrices, the *edge-pushing* algorithm proposed in Gower and Mello (2012) and analyzed further in Wang, Gebremedhin, and Pothen (2016) represents a different approach to compute this second-order information.

As another alternative, one could also propagate Taylor polynomials through the tangent procedure given in Table 2 using appropriate Taylor arithmetic. This allows computing derivatives of arbitrary order, see, for example, Bischof, Corliss, and Griewank (1993) and Griewank, Utke, and Walther (2000).

The principle of propagating information forward and backward through the function evaluation can also be used to obtain other information like sparsity patterns of Jacobians and Hessians (as described in Section 6), convex under estimators (Beckers, Mosenkis, & Naumann, 2012), and generalized derivatives (Griewank, 2013).

3.4 | Combined modes

For the computation of first-order derivatives using the forward mode or the reverse mode of AD, rigorous error analysis that show that the computed derivatives are exact up to working accuracy have been developed (Griewank, Kulshreshtha, & Walther, 2012). Similar analysis cannot however be done for other modes of AD, since it is possible to construct examples where an error blow up may occur.

Nevertheless, it might be useful to combine the forward and the reverse mode at an intermediate level for efficiency purposes (Volin & Ostrovskii, 1985). This was done, for example, in the AD tool ADIFOR, where the developers used what they call *statement-level reverse mode* (Hovland, Bischof, Spiegelman, & Casella, 1997), which we cited earlier as one variant of the cross-country mode. Here, the reverse mode is applied at the statement level but the forward mode is used to compute the overall derivative information. At a somehow higher level, a similar approach is taken in what is called *interface contraction* (Bücker & Rasch, 2003), where the local structure of a given code is exploited to temporarily reduce the global number of derivatives propagated through the code.

An alternative structure exploitation for the computation of complete Jacobian matrices is the use of a *cross-country* approach (see (Naumann, 2000) for a formal definition). Applying the forward mode to a function evaluation corresponds to the propagation of derivatives through the computational graph from the roots to the leaves, that is, from left to right in Figure 2. Using the reverse mode, one propagates derivatives from the leaves to the roots, that is, from right to left in Figure 2. Instead of this "one-directional" approach, one might also compute derivatives in a kind of cross-country mode. For example, in Figure 2, for a given vector of independents, one could first perform the derivative calculation for the vertices v_2 and v_4 before doing something else. This would influence the number of operations required to compute the complete Jacobian $F'(x)$. It was shown in Naumann (2006) that finding a way to compute $F'(x)$ with a minimal number of multiplications—a task that is generally referred to as the optimal Jacobian accumulation (OJA) problem—is NP-hard. Therefore, several heuristics have been developed to approximate an optimal strategy. However, these approaches are yet to be realized in AD tools in a general purpose fashion. The statement-level preaccumulation and the basic-block preaccumulations performed by OpenAD/F and ADIC 2.0 (which additionally incorporate some heuristics) represent some first steps in this direction.

4 | IMPLEMENTATION PARADIGMS

The two major paradigms for implementing stand-alone AD tools are *source transformation* and *operator overloading*. We discuss the basic principles for these two techniques and tools around them in Section 4.1 to Section 4.3. Due to various programming language features, such as dynamic memory allocation and pointers, AD packages have largely been based on source transformation for Fortran 77 codes and operator overloading for C/C++ codes. However, with the development of Fortran 95, the increased use of Matlab and scripting languages like Python, such a clear distinction is no longer apparent. In addition to stand-alone AD tools, nowadays numerous software packages provide a built-in AD facility. This holds true for modeling languages such as AMPL or GAMS, finite element tools like FEniCS, or, especially important in the context of this paper, machine learning libraries. We discuss some aspects of such high-level AD implementations in Section 4.4 and Section 5.

4.1 | Source transformation

When using source transformation to implement AD, a source code that evaluates the function to be differentiated is given to the AD tool together with information as to which variables are the independents and which variables are the dependents. The AD tool then generates a new source code to compute the function as well as the required derivative information. To achieve this, one needs in essence a complete compiler. As a first step, this compiler parses the program to be differentiated to perform analysis, including syntactic and semantic analysis. In a second step, a transformation is performed to include the statements for the derivative calculations. In a third step, optimization of the resulting code is carried out to increase performance. Finally, in a fourth step, the extended program is written out in the target programming language. As can be inferred from this outline, the first and fourth steps (and to a large extent the third step) are independent of AD and belong to standard compiler technology. Therefore, it is a rather complex task to develop and maintain an AD tool based on source transformation.

Example 4.1 (Source transformation applied to Example 2.1). *To the C code given in Figure* 1 *that evaluates our basic example, we applied the web interface of the AD tool Tapenade (Hascoët & Pascual*, 2013*; Tapenade*, 2018*) to obtain a code that evaluates the derivatives. The left panel in Figure* 5 *lists the code obtained when the forward mode is used. The code generated using the reverse mode is displayed in the right panel of the same figure. As can be seen, the code generated using the forward mode agrees almost line-by-line with the tangent procedure given in Table* 3*. The same holds true for the code generated using the reverse mode. This is because Tapenade exploits a lot of structure for the reverse mode. For example, it recognizes that there are no overwrites such that the storing and retrieving of the intermediate values is not necessary in this case. Therefore, the standard push and pop operations introduced in Equation (6) do not appear. Also, note that not the complete*

FIGURE 5 Differentiated code generated by Tapenade for the function in Example 2.1. Left: Forward mode, right: Reverse mode

```
double x1, x1d, x2, x2d, x3, x3d;
double v1, v2, v3, v4, v5, v6;
double v1d, v2d, v3d, v4d, v5d, v6d;
double y1, y1d, y2, y2d;
...
v1d = x1d + x2d;
v1 = x1 + x2;
v2d = v1d*exp(v1);
v2 = exp(v1);
v3d = v1d*cos(v1);
v3 = sin(v1);
v4d = x3d*x3 + x3*x3d;
v4 = x3*x3;
v5d = v2d + v3d;
v5 = v2 + v3;
v6d = v4d*v1 + v4*v1d;
v6 = v4*v1;
y1d = v5d;
y1 = v5;
y2d = v6d;
y2 = v6;
```

```
double x1, x1b, x2, x2b, x3, x3b;
double v1, v2, v3, v4, v5, v6;
double v1b, v2b, v3b, v4b, v5b, v6b;
double y1, y1b, y2, y2b;
...
v1 = x1 + x2;
v4 = x3*x3;
v6b = y2b;
y2b = 0.0;
v5b = y1b;
y1b = 0.0;
v4b = v1*v6b;
v2b = v5b;
v3b = v5b;
v1b = cos(v1)*v3b+exp(v1)*v2b+v4*v6b;
x3b = 2*x3*v4b;
x1b = v1b;
x2b = v1b;
```

function is evaluated but just those statements that are needed to compute the derivatives. Such optimizations for efficiency are not taken into account in the theoretical version of the reverse mode discussed in Section 3.2.

4.2 | Operator overloading

The technique of operator overloading exploits the capability of programming languages like C++, Fortran 95, and the like to define a new class of variables and to *overload* operators like +,−,* and functions like sin(). Using operator overloading, it is rather a matter of patience to implement a tool for the forward mode AD. For example, one could define a new class *adoublet* like

```
class adoublet
  { double value;
    double deriv; }
```

such that each variable of type *adoublet* does not carry only the value but also a corresponding derivative value. Then, one would overload an addition using

```
adoublet operator + (adoublet& u, adoublet& w)
  { adoublet v;
    v.value=u.value+w.value;
    v.deriv=u.deriv+w.deriv;
    return v; }
```

and the sin() function using

```
adoublet sin (adoublet u)
  { adoublet v;
    v.value=sin(u.value);
    v.deriv=cos(u.value)*u.deriv;
    return v; }
```

As can be seen, all other operators and functions can be overloaded in a similar fashion using the basic rules for differentiation.

In contrast to the forward mode, designing a tool for the reverse mode of AD is difficult. This is primarily due to the fact that the computational graph has to be traversed in the backward direction using values that were generated during the traversal in forward direction. Hence, one idea could be to define the new class such that instances of this class can be used to represent the computational graph. This would require a corresponding pointer structure. However, for function evaluations of reasonable size, it turns out that the maintenance of such a graph structure can be too expensive. Therefore, almost all current AD tools based on operator overloading use a new class of variables to generate an internal representation of the computational graph in an appropriate data structure. This internal function representation is frequently called *trace*.

For the actual implementation, each variable of the new class gets an identifier and each operation stores information about the computational graph. Such a class may look like

```
class adouble
  { double value;
    int   index;}
```

and the + operator could be overloaded by

```
adouble operator + (adouble u, adouble w)
  { adouble v;
    v.value=u.value+w.value;
    v.index=newindex();
    store_op("plus"); store_ind(u.index, w.index, v.index);
    return v;
  }
```

Such an implementation would yield a data structure representing the computational graph that can be accessed in a strictly sequential manner. Subsequently, this computational graph can be used to propagate derivatives forward and backward, where, in the second case, a preparatory forward sweep is needed to store the required intermediate values using something similar to the push() operation. Based on this approach, the resulting AD tool can use the computational graph to also propagate higher-order derivatives, sparsity patterns, and derivative evaluations at other values of the argument x. Thus, this strategy offers a great flexibility.

Alternatively, one could store not the structure of the computational graph but partial derivatives of the involved variables. This corresponds to the exploitation of the weights for the edges in the computational graph as illustrated in Figure 2. One could then use an implementation like

```
adouble operator + (adouble u, adouble w)
  { adouble v; v.value=u.value+w.value;
    v.index=newindex();
    store(u.index;1);  // partial derivative of v w.r.t. u is 1
    store(w.index;1);  // partial derivative of v w.r.t. w is 1
    return v;
  }
```

This allows a more efficient derivative computation because one only has to update intermediate values instead of analyzing the operation performed, computing the corresponding partial derivatives, and updating the adjoints appropriately while traversing the computational graph. However, one loses flexibility with respect to other tasks like the computation of sparsity patterns. Both approaches are well established and used by mature AD tools (Walther & Griewank, 2012) for the storage of the computational graph and (Sagebaum, Albring, & Gauger, 2017) for the storage of partial derivatives.

FIGURE 6 Calculation of derivatives of the function in Example 2.1 using ADOL-C

```
double xp1, xp2, xp3;                      % passive variant of independent variables, required for initialization
adouble x1, x2, x3;                        % active variant of independent variables using new data type
double y1, y2;                             % dependent variables;
adouble v1, v2, v3, v4, v5, v6;            % intermediate variables using new data type
int tag = 1;                               % identifier of trace
trace_on(tag);                             % begin of trace, tag = trace identifier
   x1 <<= xp1; x2 <<= xp2; x3 <<= xp3;     % set and mark independent variables
   v1 = x1 + x2;
   ...                                     % function evaluation unchanged
   v6 = v4 * v1;
   v5 >>= y1; v6 >>= y2;                   % set and mark dependent variables;
trace_off();                               % end of trace
...                                        % initializations: x, xd, yb
jac_vec(tag, 2, 3, x, xd, yd);             % compute yd = F'(x)*xd at argument x for trace with number tag
                                           % with 2 dependent variables and 3 independent variables
vec_jac(tag, 2, 3, 0, x, yb, xb);          % compute xb = yb*F'(x) at argument x for trace with number tsg
                                           % with 2 dependent variables and 3 independent variables
```

Example 4.2 (Illustration of use of ADOL-C to evaluate derivatives for the code in Example 2.1). *Figure* 6 *illustrates the derivative calculation for Example* 2.1 *when using the operator overloading AD tool ADOL-C. One can clearly identify the generation of the internal function representation called trace. It is important to note that the function evaluation itself remains to a large extend unchanged. Only the declaration, the commands to initiate and stop the trace generation and the drivers of ADOL-C to calculate the desired derivative information are modified and added, respectively. These are shown in blue in the code snippet.*

4.3 | Trade-off and sample list of tools

An obvious advantage of the approach based on source transformation is that compiler optimization can be used to make the evaluation of the code generated for the derivative computation more efficient. However, it is now well accepted that operator overloading is indispensable to cover language features that come with C++ and Fortran 95 like classes, templates, and the like. Thus, one has to live with the drawback that just using operator overloading already hinders compiler optimization independent of the specific approach chosen to implement AD.

Table 6 lists some representative examples of currently publicly available stand-alone AD tools. The website www.autodiff.org has a more comprehensive list of available AD tools.

4.4 | High-level implementations of AD

Several software packages primarily aimed at other purposes have over time been enlarged to provide their own AD facility. For example, this is true of the modeling languages AMPL (Gay, 1991) and GAMS (Brooke, Kendrick, & Meeraus, 1988), although it is often unrecognized by users since the coupling to an optimizer exploits the derivatives provided by AD in an automated fashion. A more recent development (that is better recognized by the community) is the extension *dolfin-adjoint* (Farrell, Ham, Funke, & Rognes, 2013) of the finite element package FEniCS. The tool provides adjoints and hence the reverse mode of AD using a very high-level functional representation of optimization problems with partial differential equation constraints.

There is also a burgeoning research activity in AD for domain-specific and emerging languages. Examples include *autodiff* for Halide (Ragan-Kelley et al., 2013), *JuliaDiff* for Julia (Bezanson, Edelman, Karpinski, & Shah, 2017), and *Clad* as AD tool using Clang and LLVM (Vassilev, Vassilev, Penev, Moneta, & Ilieva, 2015). *Tangent* (van Merriënboer, Wiltschko, & Moldovan, 2017) can be used for the AD of code generated with the scripting language Python. The developers of Tangent state that Tangent takes the source code of the Python function, converts it into an abstract syntax tree, and then performs reverse mode AD on this version of the computational graph to generate new Python code for the evaluation of gradients. Therefore, the underlying approach is very similar to source-transformation-based AD tools. In contrast, the approach taken by *Swift for TensorFlow* (Swift, 2019) is very similar to an operator-overloading-based approach. Thus, one finds that the

TABLE 6 Sample list of stand-alone AD tools

Language	Tool	Paradigm	Mode
C/C++	ADOL-C	OOL	FM, RM, Taylor arithmetic
	ADIC	ST	FM
	CasADi	OOL	FM, RM
	CoDiPack	OOL	FM, RM, Taylor arithmetic
	CppAD	OOL	FM, RM, Taylor arithmetic
	dco/c++	OOL	FM, RM
	OpenAD	ST	FM, RM
	Sacado	ST	FM, RM
	TAPENADE	ST	FM, RM
Fortran	ADIFOR	ST	FM
	OpenAD	ST	FM, RM
	TAF	ST	FM, RM
	TAPENADE	ST	FM, RM
Python	ADOL-C	OOL	FM, RM, Taylor arithmetic
	CasADi	OOL	FM, RM
Julia	JuliaDiff	OOL	FM, RM, Taylor arithmetic
MATLAB	AdiMat	ST + OOL	FM, RM
	CasADi	OOL	FM, RM
R	ADOL-C	OOL	FM, RM, Taylor arithmetic
	Madness	OOL	FM, Taylor arithmetic

Abbreviations: FM, forward mode; OOL, operator overloading; RM, reverse mode; ST, source transformation.

implementation strategies described earlier for the stand-alone AD tools also apply for these high-level tools with AD capabilities.

5 | AD AND MACHINE LEARNING

Many methods in machine learning require the evaluation of derivatives, and particularly, gradients and Hessian-vector products. Although, as discussed thus far, AD can furnish these quantities accurately and efficiently and has been successfully used in a wide range of areas, AD has generally been used quite often covertly within the machine learning community. This is due to the fact that prominent tools for machine learning like *TensorFlow* (Ten, 2019) have a built-in AD facility that can be used without noticing that AD is applied. The awareness with respect to AD has begun to gradually change however, in part thanks to the success of deep learning (Goodfellow, Bengio, & Courville, 2016). Projects such as *PyTorch* (Paszke et al., 2017) and *autograd* (Maclaurin, 2016) are introducing AD to mainstream machine learning. Table 7 gives an overview of commonly used machine learning libraries that have built-in AD capabilities.

Besides the general use of AD in machine learning, an especially interesting—but not widely known—link between AD and machine learning is the connection between the reverse mode of AD and backpropagation. In particular, backpropagation is simply a specialized variant of the reverse mode. The recent article (Baydin, Pearlmutter, Radul, & Siskind, 2018) surveys the intersection of AD and machine learning, and gives an interesting account of the repeated times backpropagation and the reverse mode had been reinvented in each of the two domains.

One very interesting aspect from an AD point of view is the use of nonsmooth activation functions in neural networks. A well-known example is the *relu* function $f(x) = \max\{0, x\}$. Even for such nonsmooth functions, AD is often applied in a black-box fashion. Here, the argument is that the points, where the considered function is nondifferentiable, form a set with measure zero. Therefore, the probability to hit such a point in floating point arithmetic is zero. However, exploiting the non-smooth structure may provide additional information that could speed up the optimization process. For this reason, the AD tools ADOL-C (Walther & Griewank, 2012), CppAD (Bell, 2019), and Tapenade (Hascoët & Pascual, 2013) have been

TABLE 7 Sample list of machine learning tools with AD capability. RM stands for reverse mode AD

Tool	Mode	Comment
Arrayfire	RM	Implementation based on the package autograd
MXNet	RM	Implementation based on autograd
PyTorch	RM	Implementation based on autograd
TensorFlow	RM	Trace-based implementation, higher derivatives via repreated application

```
procedure SPARSECOMPUTE($F : R^n \rightarrow R^m$)
    Step 1.  Determine the sparsity structure of the derivative matrix $A$ of $F$.
    Step 2.  Using a coloring on an appropriate graph of $A$, obtain an $n \times p$ seed matrix $S$.
    Step 3.  Compute the numerical values of the entries of the compressed matrix $B \equiv AS$.
    Step 4.  Recover the numerical values of the entries of $A$ from $B$.
```

FIGURE 7 A framework for computing a sparse derivative matrix. Step 1 and Step 3 need to be done by an algorithmic differentiation tool; Steps 2 and 4 can be done outside an AD tool

TABLE 8 Overview of graph coloring models in computation of sparse derivative matrices. The Jacobian is represented by its bipartite graph $G_b = (V_1, V_2, E)$, where V_1 corresponds to rows and V_2 to columns. The Hessian is represented by its adjacency graph $G = (V, E)$

Matrix	Direct recovery	Recovery via substitution
Jacobian	*Distance-2 coloring on* V_2	NA
Hessian	*Star coloring*	*Acyclic coloring*

extended to provide correspondingly generalized derivatives (in earlier effort ADIFOR's exception handling supported generalized gradients). An optimization algorithm based exactly on this derivative information is proposed in Fiege, Walther, and Griewank (2018).

6 | SPARSITY EXPLOITATION

For large-scale problems, the underlying derivative matrices—namely, Jacobians and Hessians—are quite often very sparse. That is, they contain numerous zero entries. Instead of computing all the zero entries with the vector modes of AD as described in Section 3, one can explicitly exploit the sparsity to make the derivative calculation much more efficient. Figure 7 outlines a scheme—based on *compression*—that has been found to be an effective framework for computing sparse Jacobian as well as sparse Hessian matrices (Coleman & Cai, 1986; Coleman & Moré, 1983, 1984; Gebremedhin, Pothen, Tarafdar, & Walther, 2009; Gebremedhin, Pothen, & Walther, 2008).

The input to the framework is a function F whose derivative matrix A is sparse. The matrix A could be the Jacobian or the Hessian. The seed matrix $S \in \{0,1\}^{n \times p}$ is a matrix that encodes a *partitioning* of the columns of A into p groups. In particular, the (j,k)th entry of the matrix S is one if the jth column of the matrix A belongs to group k and zero otherwise. Based on the sparsity structure of the matrix A (obtained in Step 1), the goal in the determination of the seed matrix S (in Step 2) is to make the number of groups p as low as possible.

6.1 | Seed matrix computation

The set of criteria used to define the seed matrix S—the partitioning problem—depends on whether the derivative matrix A to be computed is a Jacobian (nonsymmetric) or a Hessian (symmetric). It also depends on whether the entries of the matrix A are to be recovered from the compressed representation B *directly* (without requiring any further arithmetic) or *indirectly* (for example, by solving for unknowns via successive substitutions). Graph coloring problems have proven effective in modeling the matrix partitioning problems in the various computational scenarios and in designing algorithms for solving the problems, as the pioneering works in (Coleman & Cai, 1986; Coleman & Moré, 1983, 1984) showed. Table 8 gives an overview of the coloring models used. The table does not include "bicoloring models," coloring models used in the computation of the Jacobian via both row-wise and column-wise compression (Coleman & Verma, 1998; Hossain & Steihaug, 1998). A comprehensive review and synthesis of the use of graph coloring in derivative computation is available in (Gebremedhin, Manne, &

Pothen, 2005). Efficient star and acyclic coloring algorithms for Hessian computation have been presented in Gebremedhin, Tarafdar, Manne, and Pothen (2007). Hessian recovery algorithms that take advantage of the structures of star and acyclic coloring have been presented in Gebremedhin et al. (2009). A suite of implementations of various coloring algorithms, recovery methods, and graph construction routines (needed in Steps 2 and 4 of procedure sparseCompute in Figure 7) is provided in the software package ColPack (Gebremedhin, Nguyen, Patwary, & Pothen, 2013). ColPack is integrated with ADOL-C and has been in distribution since ADOL-C version 2.1.0.

6.2 | Sparsity pattern detection

Several techniques for sparsity pattern detection (Step 1 in sparseCompute) have been suggested for both of the major AD implementation paradigms source transformation and operator overloading. The techniques could be classified as *static* and *dynamic*, depending on whether analysis is performed at compile time or run time. For a determination at runtime, two major approaches could be identified in the literature: *sparse vector*-based and *bit vector*-based. In the first one, sparse vectors are propagated through the tangent and adjoint procedure discussed in Section 3. Early examples of works employing sparse vector-based approaches include Bartholomew-Biggs, Bartholomew-Biggs, and Christianson (1994) and Bischof, Khademi, Buaricha, and Alan (1996).

Bit vector-based approaches avoid this need for dynamic memory management, at the cost of increased memory requirement. An early example of a bit vector approach in the context of source transformation is the work in Giering and Kaminski (2006). When bit vectors are used, say, in the forward mode, the Jacobian is multiplied from the left by n bit vectors, where n is the number of independent variables; each arithmetic operation in the forward sweeps then corresponds to a logical **OR**, yielding the overall sparsity pattern of the Jacobian. Since one Jacobian-vector product needs to be performed for each independent variable, the complexity of this approach is $O(n \cdot OPS(F))$, where $OPS(F)$ is the number of operations involved in the evaluation of the function F. This time complexity can be reduced via Bayesian probing (Griewank & Mitev, 2002). Alternatively, index sets can be propagated through the tangent and adjoint procedure to obtain the sparsity pattern (Gebremedhin et al., 2008).

7 | CHECKPOINTING

As we have seen in Section 3, the reverse mode of AD allows the computation of gradient information with runtime that is only a very small multiple of the time needed to evaluate the underlying function. However, the memory requirement to compute this adjoint information is in principle proportional to the operation count of the underlying function as stated in Theorem 3.3. In Chapter 12 of Griewank and Walther (2008), several checkpointing alternatives to reduce this high memory complexity are discussed. Checkpointing strategies use a small number of memory units (checkpoints) to store the system state at distinct times. Subsequently, the recomputation of information that is needed for the adjoint computation but is not available is performed using these checkpoints in an appropriate way. Several checkpointing techniques have been developed, all of which seek an acceptable compromise between memory requirement and runtime increase. Here, the obvious question is where to place the checkpoints during the forward integration to minimize the overall amount of required recomputations.

If nothing about the function to be differentiated is known, a subroutine-oriented checkpointing is the only viable approach that can be used to reduce the memory requirement. Such an approach is for example implemented in the AD tools OpenAD (ope, 2012) and Tapenade (Hascoët & Pascual, 2013). However, it was proven in Naumann (2008) that the determination of an optimal checkpointing strategy, that is, one that yields for a given amount of memory the minimal runtime, is NP complete.

The situation changes completely if one assumes that the evaluation of the function to be differentiated has a time-step-like structure given by the evaluation of s steps as

$$w_i = F_i(w_{i-1}, x_{i-1}), \quad i = 1, \ldots, s \ ,$$

for a given state w_0, where $w_i \in \mathbb{R}^r$, $i = 0,\ldots,s$, denote the state of the considered system and $x_i \in \mathbb{R}^n$ denote, for example, a possible control. That is, x represents again the optimization variable. The operator $F_i : \mathbb{R}^r \times \mathbb{R}^n \to \mathbb{R}^r$ defines the step to compute the state w_i. The process to compute w_s for a given initial state w_0 and a control x is also called *forward integration*. Note, that the s steps may be time steps such that the mapping from w_0 to w_s represents a real transient process. However, also pseudo-time steps or even scenarios completely independent of time like fixed point iterations are covered by this scenario.

To optimize a specific criterion or to obtain a desired state, a cost functional

$$C(w(x),x) = C(w,x)$$

measures the quality of $w(x) = (w_0,\ldots,w_s)$ and $x = (x_0, \ldots, x_{s-1})$, where $w(x)$ depends on the control x. Applying reverse mode AD to compute a gradient of the cost function C with respect to the control x yields an adjoint procedure of the form

$$\bar{x}_s = 0\ , \quad \bar{w}_s \text{ given}$$

$$(\bar{w}_{i-1},\bar{x}_{i-1}) = \bar{F}_i(\bar{w}_i,\bar{x}_i,w_{i-1},x_{i-1}), i = s,\ldots,1,$$

where the operator $\bar{F}_i$ denotes the adjoint step. Subsequently or concurrently to the adjoint procedure, the desired derivative information $C_x(w(x),x)$ is reconstructed from $\bar{w}$ depending on the specific format of the target function.

To develop optimal checkpointing strategies, one has to take into account the specific setting of the application. A fixed number of steps to perform and a constant computational cost of all steps to be evaluated is the simplest situation. It was shown in Griewank and Walther (2000) that, for this case, a checkpointing scheme based on binomial coefficients yields, for a given number of checkpoints, the minimal number of steps to be recomputed.

An obvious extension of this approach would be to include flexibility with respect to the computational cost of the steps. For example, if one uses an implicit time-stepping method based on the solution of a nonlinear system, the number of iterations needed to solve the nonlinear system may vary from time step to time step, yielding nonuniform step costs. In this situation, it is no longer possible to derive an optimal checkpointing strategy beforehand. Some heuristics were developed to tackle this situation (Sternberg & Hinze, 2010). However, extensive testing showed that, even in the case of nonuniform step costs, binomial checkpointing is quite competitive.

Another important extension is the coverage of adaptive time stepping. In this case, the number of time steps to be performed is not known beforehand. Therefore, so-called *online checkpointing* strategies were developed (Stumm & Walther, 2010; Wang, Moin, & Iaccarino, 2009).

Finally, one has to take into account where the checkpoints are stored. Checkpoints stored in memory can be lost on failure. For the sake of resilience or because future supercomputers may be memory constrained, checkpoints may have to necessarily be stored to disk. Therefore, the access time to read or write a checkpoint is not negligible in contrast to the assumption frequently made for the development of checkpointing approaches. There are a few contributions that extend the available checkpointing techniques to a hierarchical checkpointing (Aupy, Herrmann, Hovland, & Robert, 2016; Schanen, Marin, Anitescu, & Z, 2016; Stumm & Walther, 2009).

8 | AD AND PARALLEL COMPUTING

We distinguish between two different scenarios in the context of AD and parallel computing.

The first scenario is the case where the program to be differentiated using AD is evaluated in a sequential fashion. In this case, it is quite easy to evaluate the several directions propagated in the vector forward mode or in the vector reverse mode in parallel, which is one possible application of parallel computing to AD. Such a functionality is currently incorporated in the AD tool ADOL-C.

In the second, much more important, scenario the function evaluation itself is done in parallel. In this case, applying the forward mode of AD to the given code is usually quite simple since the derivatives are just propagated forward when a shared memory programming model such as OpenMP is used. When the parallelization is based on a message passing programming model such as MPI, more care has to be taken to steer the derivative calculation in parallel (Carle & Fagan, 2002; Hovland, 1997; Utke et al., 2009).

When a parallel program is differentiated in reverse mode, the updating rule given in Equation (6) could give rise to potential race conditions one has to worry about and take care of (Bischof, Guertler, Kowarz, & Walther, 2008; Naumann et al., 2008). An obvious but not optimal solution is to use *blocking* commands. Blocking however slows down overall computation. Consequently, appropriate differentiation of parallel function evaluations using the reverse mode is still an active research topic.

Additionally, the use of different parallel programming models brings about another dimension for exploration. For shared memory programming models, the corresponding adaptations have to be made in each tool separately. Some of the AD tools listed in Table 6 provide such extensions. The situation changes considerably when a message passing programming model is used. Here, the difficulties can be handled by a separate tool that serves as an interface between the parallel simulation code to be differentiated and the AD tool. That is, it sets up the correct handling of the reverse communication required by the AD tool. Examples of such interfacing-based approaches are AMPI (AMP, 2014) and MeDiPack (med, 2019). Of the tools listed in Table 6, ADOL-C and CoDiPack support OpenMP and MPI and Tapenade can handle a useful subset of the MPI primitives.

9 | OPEN PROBLEMS

From a developer's point of view, the realization of an AD tool based on source transformation for C/C++ codes is sometimes referred to as the "Holy Grail" of AD. As explained in earlier sections, source transformation allows for compiler optimization. For this reason, it would be very advantageous to have such tools. However, due to the language features of C and especially C++, the analysis required at compile time to generate a differentiated piece of code for a full-fledged C++ application is still out of reach for the foreseeable future. Meanwhile, there is an active on-going research in this direction. For example, AD tools for C/C++ that are based on source transformation have a growing range of language coverage. As mentioned in Section 8, the differentiation of parallel function evaluations in reverse mode is still a largely unsolved problem.

For well-defined situations, that is, structured function evaluations, a variety of checkpointing approaches have been proposed. However, there are still several open questions. For example, when the function itself is evaluated in parallel, the actual implementation of parallel checkpointing approaches and also the coverage of resilience issues are still unclear. The same is true if in a time stepping procedure the size of the checkpoints varies. First results in extending the binomial checkpointing in this direction can be found in Siskind and Pearlmutter (2018).

From a mathematical point of view, when differentiating iterative processes, it is necessary to analyze whether the derivatives also converge. For fixed point iterations, there is already a fully developed theory (Griewank & Walther, 2008, Chapter 15) for an overview. However, the situation is very different for Krylov-space methods. Here, the application of AD in a black-box fashion does not yield converging derivatives (Gratton, Titley-Peloquin, Toint, & Ilunga, 2014). First theoretical results for a suitable application of AD can be found in Christianson (2018).

ACKNOWLEDGMENTS

We thank the three anonymous referees for their insightful comments and suggestions which helped improve the paper greatly. This article is funded by National Science Foundation, IIS-1553528.

CONFLICT OF INTEREST

The authors have declared no conflicts of interest for this article.

ORCID

Assefaw H. Gebremedhin https://orcid.org/0000-0001-5383-8032
Andrea Walther https://orcid.org/0000-0002-3516-4641

RELATED WIREs ARTICLES

Clustering high dimensional data

REFERENCES

Abraham, F., Behr, M., & Heinkenschloss, M. (2004). The effect of stabilization in finite element methods for the optimal boundary control of the Oseen equations. *Finite Elements in Analysis and Design*, *41*, 229–251.

Adjoinable MPI. (2014). Retrieved from https://trac.mcs.anl.gov/projects/AdjoinableMPI

Aupy, G., Herrmann, J., Hovland, P., & Robert, Y. (2016). Optimal multi-stage algorithm for adjoint computation. *SIAM Journal on Scientific Computing*, *38*, C232–C255.

Bartholomew-Biggs, M., Bartholomew-Biggs, L., & Christianson, B. (1994). Optimization & automatic differentiation in Ada: Some practical experience. *Optimization Methods and Software*, *4*, 47–73. https://doi.org/10.1080/10556789408805577

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: A survey. *Journal of Marchine Learning Research*, *18*, 1–43.

Beckers, M., Mosenkis, V., & Naumann, U. (2012). Adjoint mode computation of subgradients for McCormick relaxations. In S. Forth, P. Hovland, E. Phipps, J. Utke, & A. Walther (Eds.), *Recent advances in algorithmic differentiation Lecture notes in computational science and engineering* (Vol. 87, pp. 103–113). Berlin, Germany: Springer.

Bell, B. (2019). *CppAD*. Retrieved from https://www.coin-or.org/CppAD/

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*, 65–98.

Bischof, C., Corliss, G., & Griewank, A. (1993). Structured second- and higher-order derivatives through univariate Taylor series. *Optimization Methods and Software*, *2*, 211–232.

Bischof, C., Guertler, N., Kowarz, A., & Walther, A. (2008). Parallel reverse mode automatic differentiation for OpenMP programs with ADOL-C. In C. H. Bischof, H. M. Bücker, P. D. Hovland, U. Naumann, & J. Utke (Eds.), *Advances in automatic differentiation* (pp. 163–173). Berlin, Heidelberg: Springer.

Bischof, C. H., Khademi, P. M., Buaricha, A., & Alan, C. (1996). Efficient computation of gradients and Jacobians by dynamic exploitation of sparsity in automatic differentiation. *Optimization Methods and Software*, *7*, 1–39.

Brooke, A., Kendrick, D., & Meeraus, A. (1988). *GAMS: A user's guide*. Redwood City, California: The Scientific Press.

Bücker, M., & Rasch, A. (2003). Modeling the performance of interface contraction. *ACM Transactions on Mathematical Software*, *29*, 440–457.

Carle, A., & Fagan, M. (2002). Chapter 25: Automatically differentiating MPI-1 datatypes: The complete story. In G. Corliss, C. Faure, A. Griewank, L. Hascoët, & U. Naumann (Eds.), *Automatic differentiation of algorithms: From simulation to optimization Computer and Information Science* (pp. 215–222). New York, NY: Springer.

Christianson, B. (2018). Differentiating through conjugate gradient. *Optimization Methods and Software*, *33*, 988–994.

Coleman, T., & Moré, J. (1983). Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM Journal on Numerical Analysis*, *20*, 187–209. https://doi.org/10.1137/0720013

Coleman, T. F., & Cai, J.-Y. (1986). The cyclic coloring problem and estimation of sparse hessian matrices. *SIAM Journal on Algebraic Discrete Methods*, *7*, 221–235.

Coleman, T. F., & Moré, J. J. (1984). Estimation of sparse Hessian matrices and graph coloring problems. *Mathematical Programming*, *28*, 243–270.

Coleman, T. F., & Verma, A. (1998). The efficient computation of sparse Jacobian matrices using automatic differentiation. *SIAM Journal on Scientific Computing*, *19*, 1210–1233.

Farrell, P. E., Ham, D. A., Funke, S. W., & Rognes, M. E. (2013). Automated derivation of the adjoint of high-level transient finite element programs. *SIAM Journal on Scientific Computing*, *35*, C369–C393.

Fiege, S., Walther, A., & Griewank, A. (2018). An algorithm for nonsmooth optimization by successive piecewise linearization. *Mathematical Programming, Series A*, *177*, 343–370. https://doi.org/10.1007/s10107-018-1273-5

Gay, D. M. (1991). Automatic differentiation of nonlinear AMPL models. In A. Griewank & G. F. Corliss (Eds.), *Automatic differentiation of algorithms: Theory, implementation, and application* (pp. 61–73). Philadelphia, PA: SIAM.

Gebremedhin, A., Manne, F., & Pothen, A. (2005). What color is your Jacobian? Graph coloring for computing derivatives. *SIAM Review*, *47*, 629–705.

Gebremedhin, A., Nguyen, D., Patwary, M., & Pothen, A. (2013). ColPack: Software for graph coloring and related problems in scientific computing. *ACM Transactions on Mathematical Software*, *40*, 1–31.

Gebremedhin, A., Pothen, A., Tarafdar, A., & Walther, A. (2009). Efficient computation of sparse hessians using coloring and automatic differentiation. *INFORMS Journal on Computing*, *21*, 209–223.

Gebremedhin, A., Pothen, A., & Walther, A. (2008). Exploiting sparsity in Jacobian computation via coloring and automatic differentiation: A case study in a simulated moving bed process. In C. Bischof, M. Bücker, P. Hovland, U. Naumann, & J. Utke (Eds.), *Advances in automatic differentiation* (pp. 339–349). Berlin, Heidelberg: Springer.

Gebremedhin, A., Tarafdar, A., Manne, F., & Pothen, A. (2007). New acyclic and star coloring algorithms with application to computing Hessians. *SIAM Journal on Scientific Computing*, *29*, 1042–1072.

Giering, R., & Kaminski, T. (2006). Automatic sparsity detection implemented as a source-to-source transformation. In *International conference on computational science* (pp. 591–598). Heidelberg: Springer.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. Cambridge, MA: MIT Press. http://www.deeplearningbook.org

Gower, R. M., & Mello, M. P. (2012). A new framework for the computation of hessians. *Optimization Methods and Software*, *27*, 251–273.

Gratton, S., Titley-Peloquin, D., Toint, P., & Ilunga, J. (2014). Differentiating the method of conjugate gradients. *SIAM Journal on Matrix Analysis and Applications*, *35*, 110–126.

Griewank, A. (2012). Who invented the reverse mode of differentiation? *Documenta Mathematica*, Extra Volume ISMP, 389–400.

Griewank, A. (2013). On stable piecewise linearization and generalized algorithmic differentiation. *Optimization Methods and Software*, *28*, 1139–1178.

Griewank, A., Kulshreshtha, K., & Walther, A. (2012). On the numerical stability of algorithmic differentiation. *Computing*, *94*, 125–149.

Griewank, A., & Mitev, C. (2002). Detecting Jacobian sparsity patterns by Bayesian probing. *Mathematical Programming*, *93*, 1–25.

Griewank, A., Utke, J., & Walther, A. (2000). Evaluating higher derivative tensors by forward propagation of univariate Taylor series. *Mathematics of Computation*, *69*, 1117–1130.

Griewank, A., & Walther, A. (2000). Algorithm 799: Revolve: An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, *26*, 19–45.

Griewank, A., & Walther, A. (2008). *Evaluating derivatives: Principles and techniques of algorithmic differentiation*. Philadelphia, PA: SIAM, 2nd Retrieved from http://www.ec-securehost.com/SIAM/OT105.html

Gunzburger, M. (2003). *Perspectives in flow control and optimization*. Philadelphia, PA: SIAM.

Hascoët, L., & Pascual, V. (2013). The tapenade automatic differentiation tool, principles, model, and specification. *ACM Transactions on Mathematical Software*, *39*, 43.

Hossain, A. S., & Steihaug, T. (1998). Computing a sparse Jacobian matrix by rows and columns. *Optimization Methods and Software*, *10*, 33–48.

Hovland, P., Bischof, C., Spiegelman, D., & Casella, M. (1997). Efficient derivative codes through automatic differentiation and interface contraction: An application in biostatistics. *SIAM Journal on Scientific Computing*, *18*, 1056–1066.

Hovland, P. D. (1997). *Automatic differentiation of parallel programs*. (Ph.D. thesis). University of Illinois at Urbana-Champaign, Urbana, IL.

Maclaurin, D. (2016). *Modeling, inference and optimization with composable differentiable procedures*. (PhD thesis). School of Engineering and Applied Sciences, Harvard University.

MeDiPack—Message Differentiation Package. (2019). *SciComp, TU Kaiserslautern*. Retrieved from https://github.com/scicompkl/medipack

Naumann, U. (2000). Optimized Jacobian accumulation techniques. In M. Mastorakis (Ed.), *Problems in modern applied mathematics* (pp. 163–168). WSES Press.

Naumann, U. (2006). Optimal Jacobian accumulation is NP-complete. *Mathematical Programming*, *112*, 427–441.

Naumann, U. (2008). Call tree reversal is NP-complete. In C. Bischof, M. Bücker, P. Hovland, U. Naumann, & J. Utke (Eds.), *Advances in automatic differentiation* (pp. 13–22). Berlin, Heidelberg: Springer.

Naumann, U. (2012). The art of differentiating computer programs: An introduction to algorithmic differentiation. In *24 in software, environments, and tools*. Philadelphia, PA: SIAM Retrieved from http://www.ec-securehost.com/SIAM/SE24.html

Naumann, U., Hascoët, L., Hill, C., Hovland, P., Riehme, J., & Utke, J. (2008). A framework for proving correctness of adjoint message-passing programs. In *Proceedings of the 15th European PVM/MPI users' group meeting on recent advances in parallel virtual machine and message passing Interface* (pp. 316–321). Berlin and Heidelberg, Germany: Springer-Verlag.

OpenAD. (2012). Retrieved from https://www.mcs.anl.gov/OpenAD/

Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., … Lerer, A. (2017). Automatic differentiation in PyTorch. In *NIPS 2017 autodiff workshop: The future of gradient-based machine learning software and techniques*. Retrieved from https://openreview.net/forum?id=BJJsrmfCZ

Quarteroni, A., Sacco, R., & Saleri, F. (2000). *Numerical mathematics. Texts in applied mathematics*. Berlin, Heidelberg: Springer Retrieved from http://books.google.ca/books?id=YVpyyi1M7vUC

Ragan-Kelley, J., Barnes, C., Adams, A., Paris, S., Durand, F., & Amarasinghe, S. (2013). Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Notices*, *48*, 519–530.

Sagebaum, M., Albring, T., & Gauger, N. R. (2017). High-performance derivative computations using CoDiPack. *arXiv preprint arXiv:1709.07229*. Retrieved from https://arxiv.org/abs/1709.07229

Schanen, M., Marin, O., Anitescu, a. M., & Z, H. (2016). Asynchronous two-level checkpointing scheme for large-scale adjoints in the spectral-element solver Nek5000. *Procedia Computer Science*, *80*, 1147–1158.

Siskind, J., & Pearlmutter, B. (2018). Divide-and-conquer checkpointing for arbitrary programs with no user annotation. *Optimization Methods and Software*, *33*, 1288–1330.

Squire, W., & Trapp, G. (1998). Using complex variables to estimate derivatives of real functions. *SIAM Review*, *40*, 110–112.

Sternberg, J., & Hinze, M. (2010). A memory-reduced implementation of the Newton-CG method in optimal control of nonlinear time-dependent PDEs. *Optimization Methods and Software*, *25*, 553–571.

Stumm, P., & Walther, A. (2009). Multi-stage approaches for optimal offline checkpointing. *SIAM Journal of Scientific Computing*, *31*, 1946–1967.

Stumm, P., & Walther, A. (2010). New algorithms for optimal online checkpointing. *SIAM Journal on Scientific Computing*, *32*, 836–854.

Swift. (2019). *Swift for tensorflow*. Retrieved from https://www.tensorflow.org/community/swift

Tapenade. (2018). *Web interface*. Retrieved from http://www-tapenade.inria.fr:8080/tapenade/index.jsp

TensorFlow. (2019). Retrieved from https://www.tensorflow.org/

Tröltzsch, F. (2010). *Optimal control of partial differential equations. Theory, methods and applications* (Vol. 112). Providence, Rhode Island: AMS.

Utke, J., Hascoët, L., Heimbach, P., Hill, C., Hovland, P., & Naumann, U. (2009). Toward adjoinable MPI. In: *Proceedings of the 10th IEEE international workshop on parallel and distributed scientific and engineering computing, PDSEC-09*.

van Merriënboer, B., Wiltschko, A. and Moldovan, D. (2017) Tangent: Automatic differentiation using sourcecode transformation in Python. In *31st conference on neural information processing system*. Retrieved from https://arxiv.org/abs/1711.02712

Vassilev, V., Vassilev, M., Penev, A., Moneta, L., & Ilieva, V. (2015). Clad—Automatic differentiation using Clang and LLVM. *Journal of Physics: Conference Series*, *608*, 012055.

Volin, Y., & Ostrovskii, G. (1985). Automatic computation of derivatives with the use of the multilevel differentiating technique—I: Algorithmic basis. *Computers and Mathematics with Applications*, *11*, 1099–1114.

Walther, A., & Griewank, A. (2012). Chapter 7: Getting started with ADOL-C. In U. Naumann & O. Schenk (Eds.), *Combinatorial scientific computing* (pp. 181–202). New York: Chapman-Hall CRC Computational Science.

Wang, M., Gebremedhin, A., & Pothen, A. (2016). Capitalizing on live variables: New algorithms for efficient Hessian computation via automatic differentiation. *Mathematical Programming Computation*, *8*, 393–433.

Wang, Q., Moin, P., & Iaccarino, G. (2009). Minimal repetition dynamic checkpointing algorithm for unsteady adjoint calculation. *SIAM Journal on Scientific Computing*, *31*, 2549–2567.

How to cite this article: Gebremedhin AH, Walther A. An introduction to algorithmic differentiation. *WIREs Data Mining Knowl Discov*. 2019;e1334. https://doi.org/10.1002/widm.1334