Pesquisa Operacional (2014) 34(3): 621-645

Printed version ISSN 0101-7438 / Online version ISSN 1678-5142
www.scielo.br/pope
doi: 10.1590/0101-7438.2014.034.03.0621

ON AUTOMATIC DIFFERENTIATION AND ALGORITHMIC LINEARIZATION

Andreas Griewank

Received January 11, 2014 / Accepted March 4, 2014

ABSTRACT. We review the methods and applications of *automatic differentiation*, a research and development activity, which has evolved in various computational fields since the mid 1950's. Starting from very simple basic principles that are familiar from school, one arrives at various theoretical and practical challenges. The resulting activity encompasses mathematical research and software development; it is now often referred to as *algorithmic differentiation*. From a geometrical and algebraic point of view, differentiation amounts to linearization, a concept that naturally extends to infinite dimensional spaces. In contract to other surveys, we will emphasize this interpretation as it has become more important recently and also facilitates the treatment of nonsmooth problems by piecewise linearization.

Keywords: Jacobians, Taylor expansions, piecewise linearization.

1 INTRODUCTION AND MOTIVATION

1.1 Functions in their prime

Mathematicians are fond of writing statements like $F \in C^{1,1}(\Omega)$. This short hand means that the mapping or function $F : \Omega \subseteq X \mapsto Y$ has a locally Lipschitz-continuous derivative on the domain $\Omega \subseteq X$, with X and the range Y being linear spaces. Moreover, the *derivative* $F'(x)$ at some fixed point x is then interpreted as a linear mapping between X and Y. This abstract notion seems to be very far removed from the differentiation taught in school, where one learns for example that

$$f(x) = x^p \implies f'(x) = p\, x^{p-1}.$$

Here we have a very constructive way of computing the derivative function $f'(x)$, which can then be easily evaluated at any given point x. In contrast, attaching a prime as superscript to F in order to obtain $F'(x)$ is little more than the declaration that such a so-called Fréchet derivative exists. Only in rather rare circumstances it can be written down in terms of a procedure for evaluating it. Before discussing a more constructive approach let us first reconcile the two notions in terms of linearization.

Mathematics, Humboldt-University, Unter den Linden 6, 10099 Berlin, Germany. E-mail: griewank@math.hu-berlin.de

Throughout the paper we will denote by $y = f(x)$ real valued functions so that $Y = \mathbb{R}$, whereas $y = F(x)$ denotes vector valued functions usually with range $Y = \mathbb{R}^m$, the $m-$ dimensional Euclidean space. The domain will always be $X = \mathbb{R}^n$, i.e., the independent variables form a vector $x = (x_j)_{j=1...n}$ with $x_j \in \mathbb{R}$.

1.2 Drawing the line

Suppose for the given $\mathring{x}$ we wish to estimate the value of $F(\mathring{x}+\Delta x)$ for small increments $\Delta x \in X$ that belong to the same linear space as $\mathring{x}$. Then local Lipschitz-continuous differentiability of F at x is equivalent to the first order Taylor-expansion

$$F(\mathring{x} + \Delta x) \;=\; F(\mathring{x}) + F'(\mathring{x})\,\Delta x + O(\|\Delta x\|^2). \tag{1.1}$$

Here $\|\Delta x\|$ is a norm that measures the size of elements $\Delta x \in X$, and $O(\rho^p)$ a function of $0 \le \rho \in \mathbb{R}$ that is bounded by $c\rho^p$ for some constant $c \ge 0$. With $\|\cdot\|$ simultaneously denoting a norm on the range space Y we can write more precisely

$$\left\| F(\mathring{x} + \Delta x) \;-\; [F(\mathring{x}) + F'(\mathring{x})\,\Delta x] \right\| \;\le\; c\,\|\Delta x\|^2. \tag{1.2}$$

In other words, the original function $F(\mathring{x} + \Delta x)$ is approximated by its linearization $F(x) + F'(\mathring{x})\Delta x$ up to the quadratic error term $c\|\Delta x\|^2$. For the case when $X = \mathbb{R} = Y$ so that x, Δx and their values are real numbers we may have the situation depicted in Figure 1.

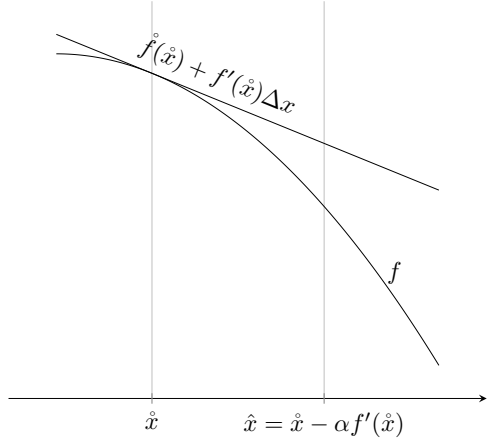

Figure 1 – Approximation of $f(x)$ by tangent line and downhill step.

Here $f(\mathring{x}) + f'(\mathring{x}) \cdot \Delta x$ represents a straight line with the slope $f'(\mathring{x}) \in R$ and the fixed reference value $f(\mathring{x})$. Obviously the tangent line provides useful information about the behavior of the function $f(x)$ for x near $\mathring{x}$. For example, it is well understood that $\mathring{x}$ can only be a minimum of $f(x)$ if $f'(\mathring{x}) = 0$. Here the derivative serves the role of verifying whether a certain optimality criterion is satisfied. If it is not, one can go downhill by stepping from a current point $\mathring{x}$ to a next point $\hat{x}$ given by

$$\hat{x} \;=\; \mathring{x} - \alpha \cdot f'(\mathring{x}) \quad \text{for} \quad 0 < \alpha \approx 0. \tag{1.3}$$

Here the step multiplier α needs to be chosen positive and sufficiently small to ensure progress towards a minimum. This so-called descent approach is the core method of optimization and also works when $n > 1$ with $f'(\mathring{x})$ the gradient vector as discussed below.

1.3 Newton's method and Jacobians

When $n = 1 = m$, the slope $f'(\mathring{x})$ can also be used in Newton's method where, given a current point $\mathring{x}$, a new point $\hat{x}$ is computed as

$$\hat{x} \;=\; \mathring{x} - f(\mathring{x})/f'(\mathring{x}) \quad \text{provided} \quad f'(\mathring{x}) \neq 0. \tag{1.4}$$

For vector functions F with $n = m$ one uses instead the formulation

$$F'(\mathring{x})(\mathring{x} - \hat{x}) \;=\; F(\mathring{x}) \quad \text{provided} \quad \det(F'(\mathring{x})) \neq 0, \tag{1.5}$$

which is equivalent to (1.4) in the scalar case $m = 1 = n$.

For general $m = n > 1$ the computation of the new iterate $\hat{x}$ requires the solution of a linear system of equations, which can be done for example by Gaussian elimination with pivoting. The key assumption for that is that the first derivative of $y = F(x)$ is available as the so-called Jacobian matrix

$$F'(\mathring{x}) \;=\; \left(\partial F_i \,\big/\, \partial x_j\right)_{j=1\ldots n}^{i=1\ldots m} \in \mathbb{R}^{m\times n}, \tag{1.6}$$

where the partial derivatives $\partial F_i \,\big/ \partial x_j \;=\; \partial y_i \,\big/ \partial x_j$ are evaluated at the current point $\mathring{x}$. In the scalar case, $m = 1 = n$ and for rather simple vector functions F one may sometimes be able to derive $F' = f'$ by hand. However, in general that is a very tedious and error prone task best left to a computer.

Evaluating Jacobians of general dimensions $m \times n$ with minimal or at least reasonable computational effort is the core task of automatic differentiation. Trying to absolutely minimize the number of arithmetic operations is impractical because it leads to a combinatorial problem that is very difficult, more specifically, NP hard.

Before we go on let us offer a few apologetic remarks on true mathematicians, who wish to have things defined precisely. Throughout the paper we will restrict our consideration to finite dimensional spaces $X = \mathbb{R}^n$, $Y = \mathbb{R}^m$, and correspondingly identify the Fréchet derivative $F'(x)$ with its matrix representation in terms of the Cartesian basis of X and Y. Moreover, we will interpret differentiability always in the sense of local Lipschitz-continuous differentiability, so that there are no order $o(\rho)$ terms at all. Thus, we can get by without any explicit limit process, but propagate first and higher order Taylor expansions forward and backward.

In any case, there is a sometimes confusing range of ways to denote derivatives, starting with the controversy between the Newton followers and the Leibniz camp in the early 18th century. Apart from Newton's prime superscript we will also use the Leibniz expression $\frac{\partial v}{\partial x}$ to denote the derivative of some scalar or vector quantity v with respect to the variable vector $x \in \mathbb{R}^n$ and $\dot{v}$ to denote its directional derivative with respect to some given direction $\dot{x} \in \mathbb{R}^n$.

1.4 Differentiation rules

Automatic differentiation is based on the classical rules expressing the derivatives of the composition of two functions F and G in terms of their individual derivatives, namely:

(i) $\dfrac{\partial}{\partial x} F(\alpha x) = \alpha F'(x)$ for $\alpha \in \mathbb{R}$;

(ii) $\dfrac{\partial}{\partial x}\binom{F(x)}{G(x)} = \binom{F'(x)}{G'(x)}$ where $\binom{G}{F} : X \to \mathbb{R}^{\tilde{m}\times m}$;

(iii) $\dfrac{\partial}{\partial x}(G \circ F)(x) = G'(F(x))\, F'(x)$ where $G : \mathbb{R}^m \mapsto \mathbb{R}^{\tilde{m}}$;

(iv) $\dfrac{\partial}{\partial x}(F(x) \pm G(x)) = F'(x) \pm G'(x)$;

(v) $\dfrac{\partial}{\partial x}(F(x)^\top G(x)) = G(x)^\top F'(x) + F(x)^\top G'(x)$.

Here it is assumed that F and G are once locally Lipschitz-continuously differentiable on their respective domains, which coincide except for the concatenation (ii) and the chain rule (iii). Then their composites will have the same differentiability properties. In the final section we will consider generalizations of these rules to the piecewise smooth scenario.

1.5 Composite function model

Every realistic computational model from the sciences, engineering, and economics is built up from simple building blocks. The specification typically involves several layers of abstractions, but we may think of it as a single computer program in an imperative language like Fortran and C, or systems like Mathematica and Maple. By applying the above simple rules recursively we must arrive at the derivatives of elementary operations and functions that are part of the programming environment. In other words, we will assume that the function in question is evaluated by a sequence of $\ell \gg 1$ instructions

$$v_i = v_j \circ v_k \quad \text{or} \quad v_i = \varphi_i(v_j) \quad \text{for } i = 1 \ldots \ell. \tag{1.7}$$

Here $\max(j, k) < i$, each $\circ$ is an addition $+$ or a multiplication $*$, and

$$\varphi_i \in \Phi \equiv \{c, \text{rec}, \text{sqrt}, \sin, \cos, \exp, \log, \ldots\} \tag{1.8}$$

is an elemental function from a given library. Φ must include in particular a constant setting $v = c$, the reciprocal $v = \text{rec}(u) = 1/u$ and typically the square root $v = \text{sqrt}(u) = \sqrt{u}$. Note that subtractions $u - w = u + (-1) * w$ and divisions $u/w = u * \text{rec}(w)$ can be performed as an addition or multiplication after multiplication of the second argument by -1 or computing its reciprocal, respectively. Identifying the first $v_k \equiv x_{k+n}$ for $k = 1 - n \ldots 0$ with the independent variables and the last $v_k \equiv y_{k-l+m}$ with the dependents, we are left with the vector $z = (v_k)_{k=1\ldots l-m}$ of intermediate quantities. The total set of variables is therefore given by

$$(x, z, y) = (v_0, \ldots, v_{n-1}, v_n, \ldots, v_{\ell-m}, v_{\ell-m+1}, \ldots, v_l) \in \mathbb{R}^{n+\ell}.$$

As an example, we may specify a vector function $y = F(x) : \mathbb{R}^3 \mapsto \mathbb{R}^2$ first by the formula

$$F(x) = \left[\log(x_1) \cdot (x_2 + x_3)/\sin(x_1),\ \sqrt{x_3} - \exp(x_2 + x_3)\right] .$$

Then we may decompose the formula into the following sequence of elemental operations:

Table 1 – Small example procedure.

$v_{-2} = x_1,\ v_{-1} = x_2,\ v_0 = x_3$	
$v_1 = \log(v_{-2})$	$\varphi_1 = \log$
$v_2 = \sin(v_{-2})$	$\varphi_2 = \sin$
$v_3 = v_{-1} + v_0$	$\varphi_3 = +$
$v_4 = v_1 \cdot v_3$	$\varphi_4 = \cdot$
$v_5 = \sqrt{v_0}$	$\varphi_5 = \sqrt{}$
$v_6 = 1/v_2$	$\varphi_6 = \text{rec}$
$v_7 = \exp(v_3)$	$\varphi_7 = \exp$
$v_8 = v_4 \cdot v_6$	$\varphi_8 = \cdot$
$v_9 = v_5 - v_7$	$\varphi_9 = -$
$y_1 = v_8,\ y_2 = v_9$	

More generally, we will consider three-part function evaluation procures of the form given in Table 2. Here the precedence relation $j \prec i$ indicates that the variable v_i depends directly on the variable v_j. It must be acyclic so that each intermediate is unambiguously defined by known quantities and we may order the v_i such that $j \prec i \implies j < i$. In actual computer programs some of the intermediate quantities v_i will share the same memory location because they are not needed at the same time. For example, in the simple example above v_4 and v_5 may overwrite v_1 or v_3.

Table 2 – General evaluation procedure.

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

That makes no difference for the forward mode of differentiation, but poses a challenge to the reverse mode as we will see. For the time being we will stay with our single assignment assumption that each variable v_i has its own memory location and occurs exactly once on the left-hand side of an instruction. For details see the standard reference [15].

1.6 Cost of elemental derivatives

Our key assumption is that all elemental functions $\varphi_i()$ are at least once Lipschitz continuously differentiable on some neighborhood $\mathcal{D}_i$ of their current argument

$$u_i \equiv (v_j)_{j \prec i} \in \mathcal{D}_i \subseteq \mathbb{R}^{n_i} .$$

Here n_i, the number of arguments, is equal to 1 for unary nonlinear functions and to 2 for binary arithmetic operations. Other elementary functions like **atan2**$(\cdot)$ or, for example, basic linear algebra subroutines can be included. The gradient of the φ_i will be denoted by

$$\varphi_i'(u_i) \;=\; (c_{ij}(u_i))_{j\prec i} \;\equiv\; \left(\frac{\partial}{\partial v_j}\varphi_i(u_i)\right)_{j\prec i} .$$

Within the forward and reverse mode to be discussed below we will need to compute not so much the gradient φ_i' itself, but its inner product with a given vector $\dot{u}_i \in \mathbb{R}^{n_i}$ or the incremental addition of $\bar{v}_i \in \mathbb{R}$ times φ_i' to a given vector $\bar{u}_i \in R^{n_i}$. In other words, we need to compute

$$\varphi_i'(u_i)\cdot\dot{u}_i \;=\; \sum_{j\prec i} c_{ij}(u_i)\cdot\dot{v}_j \quad \text{and} \quad \bar{u}_i + \bar{v}_i\cdot\varphi_i'(u_i) \;=\; (\bar{v}_j + \bar{v}_i\cdot c_{ij})_{j\prec i} \;\; .$$

Given any finite library of elementary functions Φ and any reasonable measure of computational effort OPS we will assume the existence of a constant ω such that

$$\texttt{OPS}(\varphi_i'(u_i)\cdot\dot{u}_i) \;\le\; \omega\,\texttt{OPS}(\varphi_i(u_i)) \;\ge\; \texttt{OPS}(+= \bar{v}_i\cdot\varphi_i'(u_i)) \;\; . \tag{1.9}$$

More specifically, we assume that the cost of providing the gradient $\varphi_i'(u_i)$ at the current u_i and multiplying it by the vector $\dot{u}_i \in \mathbb{R}^{n_i}$ from the right or the scalar $\bar{v}_i \in \mathbb{R}$ from the left is at most ω times as expensive as evaluating $\varphi_i(u_i)$ by itself. For example, in case of a multiplication

$$v_i = \varphi_i(v_j, v_k) = v_j \cdot v_k$$

we have $\varphi_i' = (v_k, v_j)$, $\dot{u}_i = (\dot{v}_j, \dot{v}_k)$ and the corresponding operations are

$$\dot{v}_i \;=\; v_k\cdot\dot{v}_j + v_j\cdot\dot{v}_k \quad \text{and} \quad \bar{u}_i \mathrel{+}= (\bar{v}_i\cdot v_k, \bar{v}_i\cdot v_j).$$

Thus we have in both cases two extra multiplications and either one or two extra additions. Hence, a reasonable value for ω would be 4, since multiplications and additions take about the same time on modern processors. The complexity growth factor 4 also covers the extra number of data movements, namely 4 fetches and one store compared to 2 fetches and 1 store for the original operation $v_i = \varphi_i(v_j, v_k)$. It just so happens that the multiplication operation is in some sense the worst case in that $\omega = 4$ is in fact a rather conservative estimates for the derivative complexities of the other elemental functions from the typical library Φ.

2 THEORY OF FORWARD AND REVERSE

After the preparation we can now develop the basic modes of algorithmic differentiation.

2.1 The forward mode

Let us look at a straight line $x(t) = x + \dot{x}\cdot t$ in the domain $\mathbb{R}^n$. Then each corresponding intermediate value v_i has a linearization

$$v_i(t) \;=\; v_i(x(t)) \;=\; v_i + \dot{v}_i\cdot t + O(t^2) \quad \text{with} \quad \dot{v}_i \;=\; \left.\frac{d}{dt}v_i(x(t))\right|_{t=0} .$$

The local Lipschitz continuous differentiability follows from the chain rule, and the dot quantities of the v_i can be obtained from the $\dot{x}_j = \dot{v}_{j-n}$ by the following procedure.

Table 3 – Tangent recursion for general evaluation procedure.

v_{i-n}	$\equiv$ x_i	$\dot{v}_{i-n}$	$\equiv$ $\dot{x}_i$	$i = 1 \ldots n$
v_i	$\equiv$ $\varphi_i(u_i)$	$\dot{v}_i$	$\equiv$ $\varphi_i'(u_i) \cdot \dot{u}_i$	$i = 1 \ldots \ell$
y_{m-i}	$\equiv$ $v_{\ell-i}$	$\dot{y}_{m-i}$	$\equiv$ $\dot{v}_{\ell-i}$	$i = m-1 \ldots 0$

In the little example above we obtain the extended procedure depicted in Table 4.

Table 4 – Forward differentiation on example.

$v_{-2} = x_1,\ v_{-1} = x_2,\ v_0 = x_3$	$\dot{v}_{-2} = \dot{x}_1,\ \dot{v}_{-1} = \dot{x}_2,\ \dot{v}_0 = \dot{x}_3$
$v_1 = \log(v_{-2})$	$\dot{v}_1 = \dot{v}_{-2}/v_{-2}$
$v_2 = \sin(v_{-2})$	$\dot{v}_2 = \cos(v_{-2}) \cdot \dot{v}_{-2}$
$v_3 = v_{-1} + v_0$	$\dot{v}_3 = \dot{v}_{-1} + \dot{v}_0$
$v_4 = v_1 \cdot v_3$	$\dot{v}_3 = v_1 \cdot \dot{v}_3 + v_3 \cdot \dot{v}_1$
$v_5 = \sqrt{v_0}$	$\dot{v}_5 = 0.5 \cdot \dot{v}_0/v_5$
$v_6 = 1/v_2$	$\dot{v}_6 = -v_6 \cdot v_6 \cdot \dot{v}_2$
$v_7 = \exp(v_3)$	$\dot{v}_7 = v_7 \cdot \dot{v}_3$
$v_8 = v_4 \cdot v_6$	$\dot{v}_8 = v_4 \cdot \dot{v}_6 + v_6 \cdot \dot{v}_4$
$v_9 = v_5 - v_7$	$\dot{v}_9 = \dot{v}_5 - \dot{v}_7$
$y_1 = v_8,\ y_9 = v_8$	$\dot{y}_1 = \dot{v}_8,\ \dot{y}_2 = \dot{v}_9$

Notice that in differentiating the square root $v_5 = \sqrt{v_0}$, the reciprocal $v_6 = 1/v_2$, and the exponential $v_7 = \exp(v_3)$ we have reused the function values themselves to simplify the derivative calculation, avoiding a second division, a second root, and a second exponential, respectively. It is quite clear that the computational effort for propagating the directional derivatives $\dot{v}_i$ on top of the v_i is just a small multiple of propagating the v_i by themselves. More precisely, in terms of the complexity measure OPS we obtain by (1.9) for the cost of evaluating $y = F(x)$ and $\dot{y} = F'(x)\,\dot{x}$ the bound

$$\texttt{OPS}(y, \dot{y}) \;=\; \sum_{i=1}^{\ell} \texttt{OPS}(v_i, \dot{v}_i) \;\le\; \sum_{i=1}^{\ell} (1+\omega) \cdot \texttt{OPS}(v_i) \;=\; (1+\omega) \cdot \texttt{OPS}(\texttt{y}).$$

In this estimate we have assumed that computational cost is essentially additive, which ignores for example delays due to the scheduling between the various subtasks and gains that might be made on a multicore machine by parallel executions of several threads. The upper bound of $1 + \omega \approx 5$ is certainly rather pessimistic.

Moreover, it is also significantly worse than the penalty factor 2, which one obtains if one approximates $\dot{y}$ by divided differences, i.e., utilizes the linearization

$$\dot{y} \;=\; [F(x + \varepsilon \dot{x}) - F(x)]/\varepsilon + O(\varepsilon)\ .$$

Here one only needs two values of F at neighboring points in order to estimate $\dot{y} = F'(x)\,\dot{x}$, but it is well understood that even for the optimal step multiplier ε half the number of significant digits is lost so that the accuracy is degraded and hard to predict.

If one wishes to compute the whole Jacobian, one may employ the procedure above with $\dot{x}$ ranging over the n Cartesian basis vectors in $\mathbb{R}^n$. The resulting complexity estimate is

$$\texttt{OPS}(F'(x)) \leq n\,(1+\omega)\cdot \texttt{OPS}(F(x)).$$

This bound can be reduced in theory and practice if one executes Table 3 in vector mode, i.e., with the scalars $\dot{v}_i$ replaced by a vector of directional derivatives with respect to several directions $\dot{x}$, possibly also the whole gradient $\partial v_i/\partial x \in \mathbb{R}^n$. Then common intermediate quantities can be reused and the vector operations are likely to run quite fast for a moderate number n of independent variables. Moreover, if $F'(x)$ never has more than $\hat{n} \leq n$ nonzeros in any one of its rows, one can define a seed matrix $S \in \mathbb{R}^{n\times\hat{n}}$ such that the Jacobian $F'(x)S \in \mathbb{R}^{m\times\hat{n}}$ of the reduced function $F(x+Sz)$ with respect to $z \in \mathbb{R}^{\hat{n}}$ contains enough information to reconstruct $F'(x)$ from B. This technique of *row compression* is described in some detail in [15] and reduces the complexity growth to a multiple of $\hat{n}$ rather than n.

2.2 The reverse mode

Recently, there has been a steadily growing interest in a process called *adjoining* scientific and industrial codes. The concept of adjoints applies originally to algebraic and differential equations on function spaces. Its discrete analog is what is called the reverse mode of differentiation. Instead of propagating the *dot* quantities $\dot{v}_i$ that represent sensitivities of intermediates with respect to independent variables forwards, the reverse mode propagates backwards the *bar* quantities

$$\bar{v}_i \;\equiv\; \frac{\partial}{\partial v_i}\left(\bar{y}^\top y\right).$$

In other words, with $\bar{y} \in \mathbb{R}^m$ a given weight vector we consider the sensitivities of the inner product $\bar{y}^\top y$ with respect to variations in the intermediate v_i, which may be due to round-off. For $v_{j-n} = x_j$ the adjoint quantities $\bar{v}_{j-n} = \bar{x}_j$ for $j = 1\ldots n$ form the gradient

$$\bar{x}^\top \;\equiv\; (\bar{y}^\top F(x))' \;\equiv\; \bar{y}^\top F'(x) \in \mathbb{R}^n.$$

When $F = f$ is scalar valued so that $m = 1$, we may set $\bar{y} = 1 \in \mathbb{R}$ and obtain directly the gradient $\bar{x}^\top = f'(x)$. What at first may just seem some notational manipulation turns out to be an exciting and fundamental result. Namely, one discovers that the transposed Jacobian vector product $\bar{x} = F'(x)^\top \bar{y}$ can be computed just like $\dot{y} = F'(x)\,\dot{x}$ at a cost that is a small multiple of that for evaluating F itself, which is certainly impossible by differencing. More specifically, the adjoint quantities $\bar{v}_j$ and their combinations $\bar{u}_i = (\bar{v}_j)_{j\prec i} \in \mathbb{R}^{n_i}$ can be propagated backwards by the reverse procedure listed in Table 5.

Here we use the implicit assumption that all v_i for $-n < i \leq \ell - m$ have the values they were assigned in Table 2 and that the $\bar{v}_i$ for $-n < i \leq \ell - m$ are initialized to zero.

Table 5 – Adjoint recursion for general evaluation procedure.

$\bar{v}_{\ell-i}$	$\equiv$	$\bar{y}_{m-i}$	$i = 0 \ldots m-1$
$\bar{u}_i$	$+=$	$\bar{v}_i \cdot \varphi_i'(u_i)$	$i = \ell \ldots 1$
$\bar{x}_j$	$\equiv$	$\bar{v}_{j-n}$	$j = n \ldots 1$

As one can see, adjoint statements $+= \bar{v}_i \cdot \varphi_i'(u_i)$ are executed in the opposite order in which the underlying statements $v_i = \varphi_i(u_i)$ occurred in the original program. The same is true for the initialization of the $\bar{y}_{l-i}$, which are input variables, and the deinitialization of the $\bar{x}_j$, which are output variables. Apparently the first author to write down this reverse procedure was Seppo Linnainmaa, who listed it in Fortran at the end of his Master Thesis [23], which is otherwise written in Finnish. He interpreted and used the quantities $\bar{v}_i$ to estimate the propagation of errors in complicated programs. For more information on the history of the reverse mode see [16].

The validity of Table 5 can be derived from the classical rules of differentiation using either directed acyclic graphs or matrix products [15]. It should be noted that Table 5 is, in contrast to Table 3, not a single assignment code. Starting from their initial value $\bar{v}_i = 0$ for $i \leq l - m$ the adjoint quantities may be incremented repeatedly until they reach their final value and then occur once on the right-hand side as pre-factor of φ_i'. This can be seen in the adjoint procedure Table 6, which needs to follow our little example program Table 1. Here we see that $\bar{v}_i$ for $i = 3, 0-2$ are each incremented twice before they reach their final value and can then occur on the right hand side.

Table 6 – Reverse mode on small example.

$\bar{v}_9$	$=$	$\bar{y}_2,$	$\bar{v}_8$	$=$	$\bar{y}_8,$
$\bar{v}_5$	$+=$	$\bar{v}_9$	$\bar{v}_7$	$-=$	$\bar{v}_9$
$\bar{v}_4$	$+=$	$\bar{v}_8 \cdot v_6$	$\bar{v}_6$	$+=$	$\bar{v}_8 \cdot v_4$
$\bar{v}_3$	$+=$	$\bar{v}_7 \cdot v_7$			
$\bar{v}_2$	$-=$	$\bar{v}_6 \cdot v_6 \cdot v_6$			
$\bar{v}_0$	$=$	$\bar{v}_5 \cdot 0.5/v_5$			
$\bar{v}_1$	$+=$	$\bar{v}_4 \cdot v_3$	$\bar{v}_3$	$+=$	$\bar{v}_4 \cdot v_1$
$\bar{v}_{-1}$	$+=$	$\bar{v}_3,$	$\bar{v}_0$	$+=$	$\bar{v}_3$
$\bar{v}_{-2}$	$+=$	$\bar{v}_2 \cdot \cos(v_{-2})$			
$\bar{v}_{-2}$	$+=$	$\bar{v}_1/v_{-2}$			
$\bar{x}_3$	$=$	$\bar{v}_0$ $\bar{x}_2 = \bar{v}_{-1}$	$\bar{x}_1$	$=$	$\bar{v}_{-2}$

With respect to the computational cost we find by (1.9) in analogy to the forward mode for $\bar{x} = F'(x)^\top \bar{y}$ that

$$\mathtt{OPS}(y, \bar{x}) \;=\; \sum_{i=1}^{l} \mathtt{OPS}(v_i, += \bar{v}_i \cdot \varphi_i'(u_i)) \;\leq\; \sum_{i=1}^{l} (1+\omega) \cdot \mathtt{OPS}(v_i) \;=\; (1+\omega) \cdot \mathtt{OPS}(\mathrm{Y}).$$

Here we account for executing the forward sweep Table 2 and the reverse sweep Table 5 after one another. With $\bar{y}$ ranging over all m Cartesian basis vectors of the range $\mathbb{R}^m$ we obtain the complete Jacobian at the cost

$$\texttt{OPS}(F'(x)) \leq m\,(1+\omega)\,\texttt{OPS}(F(x)).$$

Obviously this operations count is advantageous compared to the forward mode if $m < n$ and we may also employ a corresponding vector mode with $\bar{v}_i$ a vector of adjoints for several weightings $\bar{y}$. In the scalar case $m = 1$ with $f(x) = F(x)$, we have the striking result that

$$\texttt{OPS}\{f'(x)\} \;\leq\; (1+\omega)\;\texttt{OPS}\{f(x)\}.$$

In other words, as Wolfe [36] observed in 1982, gradients can 'always' be computed at a small multiple of the cost of computing the underlying function, irrespective of n, the number of independent variables, which may be huge. Since $m = 1$, we may also interpret the scalars $\bar{v}_i$ as Lagrange multipliers of the defining relations $v_i - \varphi_i(u_i) = 0$ with respect to the single dependent $y = v_\ell$ viewed as objective function. This interpretation was used amongst others by the oceanographer Thacker in [32]. It might be used to identify critical and calm parts of an evaluation process, possibly suggesting certain simplifications, e.g. the local coarsening of meshes.

2.3 The Sedgewick-Speelpenning example

To highlight the properties of the reverse mode, let us consider a very simple example of variable dimension that was originally suggested by the late Arthur Sedgewick, the PhD supervisor of Bert Speelpenning at the University of Urbana Champaign. They considered a simple product and its gradient

$$f(x) \;=\; \prod_{i=1}^{n} x_i \quad \Rightarrow \quad g_j(x) \;\equiv\; \prod_{i\neq j}^{n} x_i \;\equiv\; \frac{\partial f(x)}{\partial x_j} \;=\; \frac{f(x)}{x_j}\,.$$

Computing each gradient component independently would require $n^2 - O(n)$ multiplications and there are cheaper alternatives. Firstly, once f has been evaluated at the cost of $n-1$ multiplications, the whole gradient $g(x) = f'(x)$ can be obtained using n divisions. However, this approach is not entirely convincing since divisions are much more expensive than multiplications and may lead to a `NaN` if some component x_j is zero. Now let us apply the reverse mode. Starting from the natural, forward evaluation loop

$$v_0 = 1;\; v_i = v_{i-1} \cdot x_i \quad \text{for} \quad i = 1 \ldots n\;; \quad y = v_n$$

we obtain the combination of forward and reverse sweep listed in Table 7.

Here we have eliminated the assignments from the x_j to the v_{j-n} and vice versa from $\bar{v}_{j-n}$ to $\bar{x}_j$ for the sake of readability.

We could also have replaced the incremental assignments by direct assignments since no intermediate occurs more than once as an argument. That would save n zero initializations and the same number of additions.

Table 7 – Reverse mode on Speelpenning's example.

v_0	$=$	1	
v_i	$=$	$v_{i-1} \cdot x_i$	$i = 1 \ldots n$
y	$=$	v_n	
$\bar{v}_n$	$=$	1	
$\bar{x}_i$	$+=$	$\bar{v}_i \cdot v_{i-1}$	$i = n \ldots 1$
$\bar{v}_{i-1}$	$+=$	$\bar{v}_i \cdot x_i$	

Of course, the key observation is that this gradient procedure requires at most 4 times as many arithmetic operations as the evaluation of the function itself and involves no tests and branching. This effect was observed by Arthur S. who suggested to Bert S. rather daringly that something like that should be possible for much more general functions. That suggestion led Bert S. indeed to the reverse mode, which can be implemented in a completely automatic line by line fashion and yields gradients with optimal complexity, at least up to a small constant. The AD community only learnt in 2012 at a conference in Fort Collins about Arthur S. and usually simply refers to Speelpenning's example.

2.4 Storing the trajectory

When m and n are similar, the forward mode is still preferable because of the following memory issue. The forward mode as specified in Table 3 can be performed more or less in-place with the storage essentially doubled as the scalars $\dot{v}_i \in \mathbb{R}$ must be stored in addition to the v_i. The dot quantities $\dot{v}_i$ and $\dot{v}_j$ can and should share the same memory location exactly when that is true for v_i and v_j. The results will still be consistent in that $\dot{v}_i$ is the directional derivative of v_i, even if there were some unintended overwriting, for example through the aliasing of calling parameters.

The situation is radically different in the reverse mode. Here the quantity v_j may be used last in calculating some v_i with $j \prec i$ and then overwritten by some v_k with $k > i$, all that still during the forward sweep Table 2. Then the old value of v_j will no longer be available when we have to perform the calculation $+= \bar{v}_i \cdot \varphi_i'(u_i)$ with v_j one of the components of $u_i \in \mathbb{R}^{n_i}$. If φ_i is nonlinear, the partials $c_{ij}(u_i)$ cannot be evaluated. Hence, the old value of v_j must be written on a stack just before it is overwritten and then recuperated on the reverse sweep.

Alternatively, some AD implementations prefer to store the partials c_{ij}. For either strategy and even if linear operations are identified, we must expect that on average about one floating point number needs to be stored per elemental function. Consequently, the maximal memory requirement for the combined forward and reverse sweep will be much larger than that for the original evaluation of F in that

$$\texttt{MEM}(\bar{x}) \sim \texttt{OPS}(y) \gtrsim \texttt{MEM}(y).$$

Here $\texttt{MEM}(\bar{x})$ represents the memory requirement for the combined forward and reverse sweep, which can mostly be stored onto a stack.

In case of the Sedgewick-Speelpenning example the function itself can be evaluated using just $O(1)$ memory locations apart from the independents x_j for $j = 1 \ldots n$. More specifically, all partial products v_i can be stored in the same memory cell as y. However, for the sake of the reverse sweep, the v_i must be kept around either an array or some stack on a remote storage device.

Note that the memory estimate above applies to the vector and scalar cases $m > 1$ and $m = 1$ alike. Hence, from a memory point of view it is advantageous to propagate several adjoints simultaneously backward, for example in an optimization calculation with a handful of active constraints. Originally, the memory usage was a big concern because memory size was severely limited. Today the issue is more the delay caused by large data movements from and to external storage devices, whose size seems almost unlimited. As already suggested by Benett [1] and Ostrowski [29] et al., the memory can be reduced by orders of magnitude through an appropriate compromise between storage and recomputation of intermediates, described as checkpointing in [15].

3 OTHER ASPECTS OF THE BASIC MODES

3.1 Gradients and adjoint dynamics

The cheapness of gradients is of great importance for nonlinear optimization, but still not widely understood, except in the time dependent context. There we may have, on the unit time interval $0 \leq t \leq 1$, the primal dual pair of evolutions

$$\begin{aligned} \dot{v}(t) \equiv \partial v(t)/\partial t &= F(v(t)) && \text{with} \quad v(0) = x, \\ \dot{\bar{v}}(t) \equiv \partial \bar{v}(t)/\partial t &= F'(v(t))^\top \bar{v}(t) && \text{with} \quad \bar{v}(1) = f'(v(1)). \end{aligned}$$

Here the state v belongs to some Euclidean or Banach space and $\bar{v}$ to its topological dual. Correspondingly, the right-hand side $F(v)$ and its dual $F'(v)^\top \bar{v}$ may be strictly algebraic or involve differential operators.

Then it has been well understood since Pontryagin that the gradient of a scalar function $y = f(v(1))$ with respect to the initial point x is given by $\bar{v}(0)$. It can be computed at maximally $\omega = 2$ times the computational effort of the forward calculation of $v(t)$ by additionally integrating the second, linear evolution equation backward. In the simplest mode without checkpointing this requires the storage of the full trajectory $v(t)$ unless the right-hand side F is largely linear. Also for each t the adjoint states $\bar{v}(t)$ represent the sensitivity of the final value $y = f$ with respect to perturbations of the primal state $v(t)$.

Of course, the same observations apply to appropriate discretizations, which implies again the proportionality between the operations count of the forward sweep and memory need of the reverse sweep for the gradient calculation. To avoid the full trajectory storage one may keep only selected checkpoints during the forward sweep and then recuperate the primal trajectory in pieces on the way back, when the primal states are actually needed.

In some sense the reverse mode is just a discrete analog of the maximum principle going back to Pontryagin. Naturally, the discretizations of dynamical systems have more structure than our general evaluation loop, but the key characteristics of the reverse mode are the same.

3.2 Numerical stability

We have demonstrated that and how the forward and reverse mode can be used to compute first order derivatives of functions given by evaluation procedures. Moroever, the computational complexity was found to be quite reasonable. Now the natural question is how numerically accurate the results are, especially in view of the fact that numerical analysts strongly believe differentiation to be an ill-conditioned process. However, this conviction is based on the notion that all we know about the function $f : \mathbb{R}^n \mapsto \mathbb{R}$ is an oracle that evaluates it with a certain absolute accuracy.

Under our assumptions we know a lot more about $f(x)$ in that we have its decomposition into a sequence of elementary function evaluations $v_i = \varphi_i(u_i)$. The key question is how exact we can evaluate the φ_i and their derivatives on a given computing platform. The celebrated IEE 754 standard prescribes in detail the computation of arithmetic operations, but only makes recommendations regarding the accuracy of special function evaluations.

As discussed in [33] we can reasonably assume that the values $\tilde{v}_i$ and $\dot{\tilde{v}}_i$ obtained from $\tilde{u}_i = (\tilde{v}_j)_{j \prec i}$ and $\dot{\tilde{u}}_i = (\dot{\tilde{v}}_j)_{j \prec i}$ by the forward differentiation procedure listed in Table 3 satisfy the relations

$$\begin{aligned} \tilde{v}_i &= (1 + \check{\varepsilon}_i) \cdot \varphi_i((1 + \varepsilon_i) \cdot \tilde{u}_i) \\ \dot{\tilde{v}}_i &= (1 + \hat{\varepsilon}_i) \cdot \varphi_i'((1 + \varepsilon_i) \cdot \tilde{u}_i) \cdot \dot{\tilde{u}}, \end{aligned}$$

where the three perturbations $\check{\varepsilon}_i, \hat{\varepsilon}_i$ and ε_i are all of the order of the machine precision $\texttt{eps}$. While a forward error analysis would be, as usual, quite complicated, we can apply a backward error analysis that shows that algorithmic differentiation yields the exact derivative values for a slightly perturbed problem.

To obtain the pair $(\tilde{v}_i, \dot{\tilde{v}}_i)$ as the exact result of $(\tilde{u}_i, \dot{\tilde{u}}_i)$ we define for each univariate elemental function $\varphi_i(u_i)$ a modification given by

$$\begin{aligned} \tilde{\varphi}_i(u_i) &\equiv (1 + \check{\varepsilon}_i) \cdot \varphi_i\left(\frac{1 + \hat{\varepsilon}_i}{1 + \check{\varepsilon}_i} \cdot u_i + \left(1 + \varepsilon_i - \frac{1 + \hat{\varepsilon}_i}{1 + \check{\varepsilon}_i}\right) \cdot \tilde{u}_i\right) \\ &= (1 + \mathcal{O}(\texttt{eps})) \cdot \varphi_i(u_i \cdot (1 + \mathcal{O}(\texttt{eps})) + \mathcal{O}(\texttt{eps} \cdot \tilde{u}_i)) \ . \end{aligned} \tag{3.1}$$

We contend that this artificially constructed $\tilde{\varphi}_i$ may be considered a *small perturbation* of φ_i given the machine precision $\texttt{eps}$. It is easy to check that it satisfies indeed exactly the relations

$$\tilde{v}_i = \tilde{\varphi}_i(\tilde{u}_i) \quad \text{and} \quad \dot{\tilde{v}}_i = \tilde{\varphi}_i'(\tilde{u}_i) \cdot \dot{\tilde{u}}_i.$$

A very similar small perturbation can be applied to the binary operations and other elementary functions with several variables.

Hence, we have shown that the forward mode of algorithmic differentiation is indeed backward stable in the sense of Wilkinson [35]. In the same way we can also establish the backward stability of the reverse mode, but then there may be some $O(\tilde{m}\,\mathtt{eps})$ terms where $\tilde{m}$ is the maximal number of times that any intermediate v_j belongs to some u_i, i.e., occurs as argument of some φ_i as $j \prec i$. Nevertheless, this is a very satisfactory backward stability result and in our by now twenty five years of experience with AD we have never heard a user complain about poor accuracy derivative values, once the logical bugs had been overcome.

3.3 Software and application

So far we have not really explained why AD became known as automatic differentiation. We have seen how the original evaluation procedure needs to be augmented with additional instructions to calculate the correct derivative values. In principle, this program transformation can be done by hand, but that is very time-consuming and error prone. It is also hard to manage; if the underlying program is subject to continual change, then keeping track of these changes in order to maintain the integrity of the transformed version is also time consuming. Therefore, it is usually desirable to automate at least partially the process of transformation. For a computer science oriented introduction to AD consult the excellent book [27]. Here we will just try to convey the basic ideas.

Conceptually, we may think of the AD process as a transformation of the form

$$\begin{array}{c} \downarrow\ \downarrow \\ \mathtt{eval(x,\,y,\,z)} \\ \downarrow \end{array} \quad \Longrightarrow \quad \begin{array}{c} \downarrow\ \downarrow \qquad \downarrow \\ \mathtt{deval(x,\,dx,\,y,\,dy,\,z)} \\ \downarrow\ \downarrow \end{array} \quad \text{or} \quad \begin{array}{c} \downarrow\ \downarrow\ \downarrow \\ \mathtt{beval(bx,\,x,\,by,\,y,\,z)} \\ \downarrow \qquad \downarrow \end{array} . \tag{3.2}$$

Here we assume that the original subprogram `eval(x, y, z)` evaluates the output vector `y` as a function of the input vectors `x` and `z`. The inputs and outputs are marked by small arrows on top and bottom, respectively. Whereas `z` is considered a constant parameter, `x` is nominated as a vector of independent variables with respect to which `y` is to be differentiated. Consequently, a dot derivative object `dx` is associated with `x` as an additional input, and a corresponding output `dy` associated with the dependent variables `y` is also included as a parameter in the tangent procedure listed `deval` corresponding to Table 3. In the simplest case `dx` is the directional derivative $\dot{x}$ and `dy` is the directional derivative $\dot{y}$ described in the section on the forward mode. Then the format of the arrays `x` and `y` would be exactly the same as that of the underlying independent and dependent variables `x` and `y`, respectively. In vector mode `dx` and `dy` could contain several directional derivatives. The reverse mode function `beval` corresponding to Table 5 has the additional input parameter `by` and the new output parameter `bx`, which means that there is a reversal of the information flow. Again `by` and `bx` may correspond to the adjoint vectors $\bar{y}$ and $\bar{x}$ or be larger derivative objects of compatible format.

The program extension can be performed essentially line-by-line, although the reverse mode requires the trajectory transfer between the forward and reverse sweep. Also, at least on a subroutine level it might be worthwhile to perform some code analysis in order to improve the derivative code or even the original procedure. Traditionally, AD was developed for codes written in the classical procedural languages Fortran and C++. This was done using two computer

science concepts employed in one guise or another, namely source transformation and operator overloading.

Source transformation requires an elaborate transformation tool that takes a certain computer code, analyzes it like a compiler and then generates a new extended source code. While the development and maintenance of such a tool is a significant effort for the developer the user needs to do very little other than making his input code truly conform to the language standard. He or she also has to nominate the independent and dependent variables and select the mode of differentiation as well as the order of the desired derivatives. The only drawback for the user may be a certain lack of flexibility in terms of what order and modes of differentiation the tool provides.

There may also be restrictions regarding recently added language features like for example in the continuous evolution of Fortran. Generally, there is the reasonable expectation that source transformation yields faster derivative codes, if only because of the application of standard compiler optimization. However, this advantage applies mainly to clearly structured numerical codes that also parallelize well. Recursions, pointers and indirect addressing can render static analysis at compile time rather ineffective.

Operator overloading tools consist mainly of a header file and a runtime library, which are both completely problem independent. The user has to retype all variables that are involved in the differentiation process to a new tool-specific type, called for example `adouble` in ADOL-C. The standard language compiler will then look up in the header file what needs to be done for all elemental operations and functions. So the extra derivative related instructions will only be included in the object file, but not in the source files. The user must also declare and initialize the independent and dependent variables in some tool-dependent way. When the code compiles without diagnostics and there are no runtime errors, one can be pretty sure that the derivative results are correct. For tools that provide the forward and reverse mode one can check the Lagrange identity

$$\bar{y}^\top [F'(x)\,\dot{x}] \;\equiv\; [F'(x)^\top \bar{y}]\,\dot{x}$$

for random vectors $\dot{x}$ and $\bar{y}$ to see if the results are consistent up to numerical round-off. If they are not, something has gone wrong in the transformation process.

Table 8 lists some of the tools in the order of their first appearance. Some of them are still available, some have metamorphosed into successor systems, and others have fallen by the wayside altogether. Of those still existent all except `TAF/TAC` are available under the GNU public license. The oldest one still under development with its original name and basic design is the overloading tool `ADOL-C`.

There is some indication that more recent projects like Adept and the NAG-compiler integrated AD utilities can achieve a significant speed-up using expression templates and other modern language constructs. Unfortunately, no large commercial software vendor has ever invested significant resources to develop a professional computing environment with AD capabilities. The modeling languages AMPL, GAMS and also the optimization test environment CUTEr have provided AD capabilities under the hood for a long time.

Table 8 – Source transformation (C) and operator overloading (O) AD tools. The modes are forward: $\rightarrow$, reverse: $\leftarrow$, both: $\leftrightarrow$, and $\Leftrightarrow$ with checkpointing.

Name	Year	Source	Type	Order	Mode	Developer
PROSE	75	F66	C	1	$\rightarrow$	J.M. Thames
Augment	80	F66	O	1	$\rightarrow$	G. Kedem
JAKEF	80	F77	C	1	$\leftarrow$	B. Speelpenning
GRESS	82	F77	C	1	$\rightarrow$	J. Horwedel
ADGEN	86	F77	C	1	$\leftarrow$	J. Horwedel
DAFOR	87	F77	C	∞	$\rightarrow$	M. Berz
ADOL-C	90	C++	O	∞	$\Leftrightarrow$	A. Walther
ADIFOR	92	F77	C	1	$\rightarrow$	A. Carle
Odyssee	92	F77	C	1	$\leftrightarrow$	Ch. Faure
ADIC	94	C	C	1	$\rightarrow$	P. Hovland
PADRE2	95	F77++	C	2	$\leftrightarrow$	K. Kubota
FADBAD	96	C++	O	∞	$\leftrightarrow$	C. Bendtsen
TAF/TAC	98	F77/C++	C	1	$\Leftrightarrow$	R. Giering
Tapenade	01	F77	C	1	$\Leftrightarrow$	L. Hascoet
CppAD	02	C++	O	∞	$\Leftrightarrow$	B. Bell
Rapsodia	07	F77/C++	C	∞	$\rightarrow$	J. Utke
Sacado	08	C++	O	2	$\leftrightarrow$	D. Gay
ADEPT	13	C++	O	1	$\leftarrow$	R. Hogan

There has also been a string of Matlab implementations, in particular ADiMat [4]. For these developments and the current state of AD tools in general one should consult the community website `www.autodif.org`.

Of course, there is also a close connection to the fully symbolic differentiation capabilities of computer algebra packages. A very exciting recent development has been the integration of AD capabilities into the problem solving environment `FEnICS` [24] for PDEs based on finite elements. See also the discussion by Korelc in [21].

Application studies with the major tools can be found in the proceedings of the workshops in Breckenridge 91 [12], Santa Fe 96 [2], Nice 00 [9], Chicago 04 [5], Bonn 08 [3], Fort Collins 12 [10]. The sheer size of a computer model is no longer an objection to the successful application of AD. However, problems arise with multi-layer codes in different languages and possibly involving precompiled proprietary software. Unfortunately, there are, as yet, no agreed upon interfaces for the transfer of sensitivities from one part of a computer model to another. In particular the function signatures of derived subprograms like `deval(x, dx, y, dy, z)` suggested above will vary from tool to tool. Fortunately, terminology and notation in AD have been unified to a large extent, at least compared to the early days.

4 FURTHER DEVELOPMENTS

Many well established results, techniques and aspects of AD could not be elaborated in this survey paper. We have covered here only first derivative calculations in the forward and reverse mode. This is of central importance, but certainly not the whole game.

4.1 Higher derivatives

By combining the forward and reverse mode one obtains a method for evaluating second order adjoints of the form

$$\bar{y}^\top F''(x)\,\dot{x} \;=\; \frac{d}{dt}\;\bar{y}^\top F'(x + t\,\dot{x})\Big|_{t=0} \;\in \mathbb{R}^n.$$

Their cost is about $\omega^2 \approx 16$ times as much as that of evaluating F by itself. When $\dot{x}$ ranges over the Cartesian basis vectors in $\mathbb{R}^n$, one obtains the complete Hessian $(\bar{y}^\top F(x))'' \in \mathbb{R}^{n\times n}$ at a cost of about $n\,\omega^2 \approx 16\,n$ times that of F. Such second order information is particularly useful in optimization.

One of the earliest application of AD was the solution of ODEs by Taylor series methods [34]. Up to 30-50 terms are really no problem and may yield very accurate solution trajectories (see [6] and [30]). There have also been some preliminary studies concerning Differential Algebraic Equations (DAEs), where AD has in my view still a large untapped potential.

Using the fact that exp(), sin(), cos() and all the other intrinsic elemental functions are solutions of linear ODEs one can cheaply compute univariate Taylor expansions

$$y(t) \;=\; \sum_{k=0}^{d-1} y_k t^k + O(t^d) \quad \text{for given} \quad x(t) \;=\; \sum_{k=0}^{d-1} x_k t^k \in \mathbb{R}^n.$$

Cheap means here that the effort only grows like $O(d^2)$ with respect to the degree d, and it could even be lowered to $O(d\,\log(d))$ using FFT based polynomial arithmetic.

Apparently that possibility has not been seriously utilized since the cross-over happens too late. Based on the coefficients of a carefully selected family of univariate Taylor expansions one may then also interpolate the entries of derivative tensors of arbitrary order [13]. All this happens in the forward mode and one may then also compute cheaply the gradients of these high order mixed partials with respect to a different set of parameters, e.g. in design optimization.

4.2 Combinatorial aspects

There is a natural tendency to consider differentiation as a mechanical process, and so far the only truly surprising thing that we have encountered is the reverse mode. In fact, there are many combinatorial aspects of AD that arise if one really wishes to minimize the computational effort.

Firstly, the forward and reverse mode are only the extreme variants of a general elimination approach on linearized computational graphs. Here one considers the variables v_i and equivalently their indices i for $i = 1 \ldots \ell$ as the vertices of a directed acyclic graph (DAG) with (j, i) an edge iff $j \prec i$. Then the minimal and maximal vertices with respect to that partial ordering are

exactly the independents $j - n$ for $j = 1 \ldots n$ and the dependents $v_{\ell-i}$ for $i = 0 \ldots m - 1$. The edges can be labeled with the elementary partials $c_{ij}(u_i)$ evaluated at the fixed current point x. When the graph is bipartite, i.e., $\ell = m$, the edge values c_{ij} represent exactly the nonzero entries of the Jacobian $F'(x)$. Otherwise the $\ell - m$ intermediate vertices can be successively eliminated according to the chain rule in the style of sparse Gaussian elimination.

More specifically, as explained in [14], the DAG can be successively simplified by vertex, edge, or face eliminations until it becomes the bipartite representation of the Jacobian. The result of this accumulation or elimination process is up to round-off independent of the order in which the individual vertices, edges, or faces are eliminated, but the operations count and memory requirement can vary drastically. Not surprising, efforts to minimize either the temporal or spatial complexity lead to NP-hard combinatorial meta-problems [26]. We call them meta-problems because the underlying task of accumulating Jacobians from the elementary partials c_{ij} can be achieved in strongly polynomial time with respect to the problem size $\ell \geq \max(n, m)$.

A restricted version of the accumulation problem is that of finding, for a given sparsity pattern of $F'(x)$, a seed matrix $S \in \mathbb{R}^{n \times p}$ with a minimal number of columns $p \leq n$ such that the compressed Jacobian $F'(x)S$ allows the identification of all nonzero entries of $F'(x)$. It has been known for a long time that the minimal p is the chromatic number of an associated column-coincidence graph [7]. Using the reverse mode one may look for so-called adjoint seeds $W \in \mathbb{R}^{q \times m}$ with minimal $q \leq m$ such that the column compressed Jacobian $WF'(x)$ reveals all nonzero entries of $F'(x)$. Moreover, the two compressions can be combined to further lower the number of directional derivatives or adjoint vectors needed for computing $F'(x)$ by a so-called Coleman-Verma partition [8]. Finally, there are special considerations and heuristic algorithms for the symmetric case when $F'(x)$ is in fact a Hessian. Suitable colorings are provided by the fairly comprehensive package ColPack [11]. An alternative approach called Newsam-Ramsdell seeding in [15], which absolutely minimizes p and or q, is described in [19].

The full storage of the forward value trajectory can be avoided by checkpointing and similar store-recompute trade-offs at a finer grain. All these tasks are combinatorial in nature and of course AD would also benefit from a host of other optimizations like expression simplification, instruction scheduling, and register allocation. It is well understood that these compiler tasks are NP-hard and can thus only be attacked by suitable heuristics. A particularly nice example is the pebble game, where one wishes to propagate adjoint values backward through the computational graph using only a fixed number $\tilde{\ell} < \ell$ of vertices in memory and minimizing the recomputations.

4.3 Generalized differentiation

In the first section we have noted that the evaluation of gradients and Jacobians is essential for the efficient numerical solution of unconstrained optimization and nonlinear equation problems. There are many other algorithms in nonlinear scientific computing that rely on successive linearizations via first order Taylor expansions. On the other hand, many realistic computer models are nondifferentiable in that the functional relation between input and output variables is not everywhere smooth. Also, from a theoretical point of view it has been realized that the solution operators of many computational problems are often nonsmooth, and may even have jumps like

bang-bang controls. By now there is a very significant body of nonsmooth analysis, based on concepts of generalized differentiation (see e.g. [25] and [22]). This development was largely ignored by the AD community, mostly because the generalized differentiation rules did not seem amenable to an automatic implementation.

The key concepts of nonsmooth analysis on finite dimensional spaces are the following: If $F : \mathbb{R}^n \to \mathbb{R}^m$ is locally Lipschitz continuous, it follows from Rademacher's theorem that it has a Fréchet derivative $F'(x)$ at all $x \in \mathbb{R}^n \setminus S$ with S a set of measure zero such that

$$\lim_{\|s\| \to 0} \|F(x+s) - F(x) - F'(x)\, s\| / \|s\| = 0 \,.$$

Moreover, the induced norm $\|F'(x)\|$ is bounded above by any local Lipschitz constant L with respect to suitable vector norms. Then we may define, for all $x \in \mathbb{R}^n$, the set

$$\partial^L F(x) \equiv \left\{ \lim_{k \to \infty} F'(x_k) : x_k \to x \text{ and } x_k \notin S \right\},$$

which we call the *limiting Jacobian*. It is never empty due to the density of the Fréchet differentiable points and the uniform boundedness of the nearby Jacobians.

As an example we obtain for the mapping $F(x) = |x|$

$$F'(x) = \begin{cases} 1 & \text{if } x > 0 \\ \varnothing & \text{if } x = 0 \\ -1 & \text{if } x < 0 \end{cases} \quad \text{and} \quad \partial^L F(x) = \begin{cases} \{1\} & \text{if } x > 0 \\ \{-1, 1\} & \text{if } x = 0 \\ \{-1\} & \text{if } x < 0 \end{cases}.$$

The convex hull

$$\partial F(x) \equiv \operatorname{conv}(\partial^L F(x))$$

is called the Clarke generalized derivative. For the example above $\partial^L F(x)$ and $\partial F(x)$ differ only at $x = 0$, with the latter expanding to $\partial F(0) = [-1, 1]$.

For the limiting Jacobian we obtain the following generalized differentiation rules, which immediately imply corresponding set inclusions for the Clarke Jacobian.

(i) $\partial^L(\alpha F) = \alpha\, \partial^L(F)$ for $\alpha \in \mathbb{R}$

(ii) $\partial^L \binom{F}{G} \subseteq \partial^L F \times \partial^L G \equiv \left\{ \binom{A}{B} : A \in \partial^L(F), B \in \partial^L(G) \right\}$

(iii) $\partial^L(G \circ F) = \partial^L G(F) \cdot \partial^L F \quad$ if $\quad F \in C^1(R^n)$ or $G \in C^1(R^m)$

(iv) $\partial^L(F \pm G) \subseteq \partial^L F \pm \partial^L G = \{A \pm B : A \in \partial^L F, B \in \partial^L G\}$

(v) $\partial^L(f \cdot g) \subseteq g \cdot \partial^L f + f \cdot \partial^L g$

(vi) $\partial^L |f| \quad \begin{cases} = \partial^L f & \text{when } f > 0 \\ \subseteq -\partial^L \cup \{0\} \cup \partial^L f & \text{when } f = 0 \\ = -\partial^L f & \text{when } f < 0 \end{cases}.$

In contrast to the list of corresponding smooth differentiation rules listed in Section 1.4, we have added the last rule applying to the absolute value function $\mathbf{abs}(x) = |x|$. Of course, in general all of the set inclusions above are proper, i.e., not satisfied as equalities even if we restrict our attention to functions defined by evaluation procedures with elementals $\varphi_i \in \Phi \cup \{\mathbf{abs}\}$. In other words, the only nonsmooth elemental function that we allow is the absolute value function, which yields immediately max and min as

$$\max(u, w) = \tfrac{1}{2}(u + w + |u - w|) \quad \text{and} \quad \min(u, w) = \tfrac{1}{2}(u + w - |u - w|) \,.$$

The resulting functions are called *composite piecewise smooth* in [20] and will be studied in the remainder of this survey paper. The more general concept of piecewise smoothness has been examined for example in [31].

4.4 Between the lines

Clearly, composite piecewise smooth functions are locally Lipschitz continuous and thus differentiable at almost all points $\mathring{x} \in \mathbb{R}$. However, the resulting linearizations (1.1) will only be valid on a small ball about $\mathring{x}$ whose radius does not exceed the distance to the next point where the Fréchet derivative $F'(x)$ is undefined. Instead, we believe that generalized algorithmic differentiation should provide an approximating function $\Delta F(\mathring{x}, \Delta x)$ at all $\mathring{x}$ such that

$$F(\mathring{x} + \Delta x) \;=\; F(\mathring{x}) + \Delta F(\mathring{x}; \Delta x) + O(\|\Delta x\|^2) \tag{4.1}$$

or, again more precisely, for a constant c

$$\|F(\mathring{x} + \Delta x) \;-\; [F(\mathring{x}) + \Delta F(\mathring{x}; \Delta x)]\| \;\leq\; c\,\|\Delta x\|^2. \tag{4.2}$$

In other words, we are looking for a generalized Taylor expansion with uniform quadratic error term. To get an idea how this is constructively possible, let us consider a univariate function of the form $F(x) = \max(F_1(x), F_2(x))$ as depicted in Figure 2.

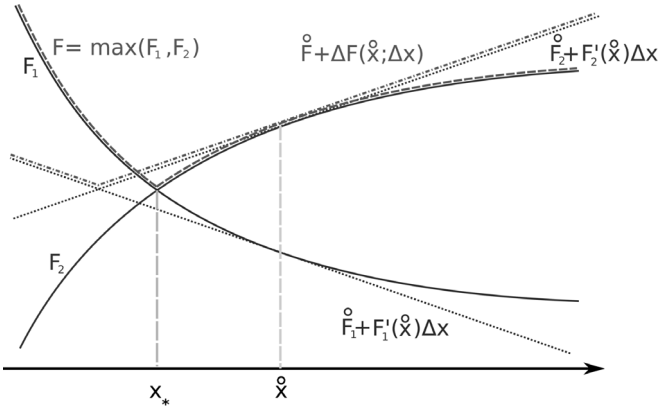

Figure 2 – Piecewise linear approximation $F(\mathring{x}) + \Delta F(\mathring{x}; \Delta x)$ of piecewise smooth $F(x)$

Since F_1 and F_2 are assumed to be smooth, the function F is everywhere differentiable except at the kink point x_*, where the two values tie. There the generalized gradient $\nabla^C F(x_*) = [F_1'(x_*), F_2'(x_*)]$ in the sense of Clarke is the interval spanned by the negative slope $F_1'(x_*)$ of the red branch and the positive slope $F_2'(x_*)$ of the blue branch. This reflects the fact that the set-valued Clarke derivative is just the convex and outer semicontinuous hull of the classical derivative $F'(x)$, which is undefined at $x = x_*$ itself.

At any $x \neq x_*$ Clarke's and all the other derivative concepts reduce simply to the slope, which gives no indication of the nearby kink whatsoever. If the F is repeatedly evaluated as part of a larger interactive computation in floating point arithmetic, the kink will almost certainly never be hit exactly, and modeling F by the tangent line of either F_1 or F_2 may of course yield rather poor results. All we are suggesting is to model F by the dashed green function $F(\mathring{x}) + \Delta F(\mathring{x}; \Delta x)$. As we will see later, this piecewise linear function is obtained by approximating $F_1(x)$ as well as $F_2(x)$ by their tangent line at the base point $\mathring{x}$ and then taking the maximum afterwards.

It is intuitively clear that this approximating function varies continuously with $\mathring{x}$ and yields a rather good approximation to the original function F on both sides of its kink. In general, the second order approximation of a composite piecewise smooth function by a piecewise linear function can be achieved according to the following extremely simple recipe.

> **Replace all smooth elemental functions by their tangent line or plane, and the piecewise linear elementals abs, max and min by themselves.**

That means we set for smooth elementals

$$\Delta\varphi_i(\tilde{u}_i; \Delta u_i) \equiv \varphi_i'(\tilde{u}_i) \cdot \Delta u_i \quad \text{with} \quad \Delta u_i \equiv (\Delta v_j)_{j \prec i}$$

and for the absolute value function

$$\Delta\mathbf{abs}(\mathring{v}_j; \Delta v_j) \equiv \mathbf{abs}(\mathring{v}_j + \Delta v_j) - \mathring{v}_i \quad \text{with} \quad \mathring{v}_j = \mathbf{abs}(\mathring{v}_j).$$

Then we can evaluate the composite incremental function $\Delta y = \Delta F(\mathring{x}; \Delta x)$ by the modification of Table 3 listed in Table 9.

Table 9 – Piecewise linearization procedure.

v_{i-n}	$\equiv x_i$	Δv_{i-n}	$\equiv \Delta x_i$	$i = 1 \ldots n$
v_i	$\equiv \varphi_i(u_i)$	Δv_i	$\equiv \Delta\varphi_i(u_i; \Delta u_i)$	$i = 1 \ldots \ell$
y_{m-i}	$\equiv v_{\ell-i}$	Δy_{m-i}	$\equiv \Delta v_{\ell-i}$	$i = m-1 \ldots 0$

Obviously, very little has changed and the computational complexity for evaluating Δy given Δx is again at most $\omega \approx 4$ times that of evaluating the composite piecewise smooth function F itself. When there are no nonsmooth elementals at all, Table 9 is equivalent to Table 3. So far we know of no straightforward generalization of the reverse mode listed for the smooth case in Table 5.

It should be noted that $\Delta F(\mathring{x};\cdot)$ is not just a matrix, but a continuous piecewise linear map from $\mathbb{R}^n$ to $\mathbb{R}^m$. That situation might take a little getting used to, since one tends to expect that the end product of any differentiation process is some collection of derivative vectors or matrices. In fact, one can describe $\Delta F(\mathring{x};\cdot)$ in terms of matrices and vectors by the so-called abs-normal form described in [18]. It can be generated directly by a minor extension of standard AD tools, which has for example been provided for ADOL-C.

In sharp contrast to the standard concepts of generalized differentiation, which are very volatile as functions of the development point $\mathring{x}$, the piecewise linearization $\Delta F(\mathring{x};\cdot)$ varies locally Lipschitz continuously with respect to $\mathring{x}$. It also satisfies exact propagation rules, namely, we have for F and G with compatible domains and dimensions

$$\begin{aligned}
\Delta[\alpha F + \beta G](x;\Delta x) &= \alpha\Delta F(x;\Delta x) + \beta\Delta G(x;\Delta x)\\
\Delta[F^\top G](x;\Delta x) &= G(x)^\top \Delta F(x;\Delta x) + F(x)^\top \Delta G(x;\Delta x)\\
\Delta[G\circ F](x;\Delta x) &= \Delta G(F(x);\Delta F(x;\Delta x)).
\end{aligned}$$

Now, what can we do using the piecewise linear models of composite piecewise smooth functions described above? Well, mostly all the things that one does with linearizations of smooth functions, such as solving equations, (un)constrained minimization and the solution of discretized ordinary and partial differential equations. The uniform second order approximation property ensures rapid convergence of such successive piecewise linearization procedures from within sizable domains of attraction.

One instant benefit of piecewise linearization is the capability to compute one or more elements of the limiting Jacobian set

$$\varnothing \neq \partial^L_{\Delta x}\Delta F(\mathring{x};\Delta x)\Big|_{\Delta x=0} \subseteq \partial^L_x F(x)\Big|_{x=\hat{x}}\ .$$

More precisely, we can compute limiting Jacobians of $\Delta F(\mathring{x};\cdot)$ by lexicographic differentiation [28] and is has been shown in [20] and [17] that these are then also limiting Jacobians of the underlying CPS function F. Furthermore, they have a desirable feature called conic activity at $\mathring{x}$, which is a little stronger than the concept of essential activity used by Scholtes [31].

5 SUMMARY

In this survey we have considered the problem of locally approximating a function $F:\mathbb{R}^n\mapsto\mathbb{R}^m$ defined as composite of elementary functions φ_i as described in Section 1. If all these elementals are smooth, i.e., Lipschitz continuously differentiable, we obtain a classical Taylor expansion in terms of the Jacobian $F'(\mathring{x}):\mathbb{R}^n\mapsto\mathbb{R}^m$ evaluated at the current point $\mathring{x}$.

In Section 2 it is described how, rather than evaluating all entries of $F'(\mathring{x})$ simultaneously, one prefers to compute matrix-vector products $F'(\mathring{x})\,\dot{x}$ or $F'(\mathring{x})^\top\bar{y}$ in the forward and reverse mode of algorithmic differentiation, respectively.

The procedures for doing that are listed in the Tables 3 and 5 with the I/O characteristics indicated in (3.2). The temporal complexity of both modes is essentially the same, namely just about

3-4 times as many operations are needed as for evaluating the underlying function F. However, there is a large gap in the spatial complexity since the reverse mode requires the recuperation of the forward value trajectory on the way back, in one way or another. At the beginning of Section 3 we consider this memory aspect and answer the question of numerical accuracy quite satisfactorily in the sense of Wilkinson backward stability.

The two basic modes have been implemented in almost a hundred different software tools adapted to the most important languages and computing environments. To give a flavor of this evolution we have listed some of them in Table 8 without any ambition of being representative. The continual tool development can be tracked on the website `www.autodif.org`. There is a lot more to AD than the basic modes, as we have indicated in Section 4. That applies in particular to the evaluation of second or higher derivatives and also certain combinatorial tasks that arise in the handling of problems that are sparse or otherwise structured. Much remains to be done in this respect, also in view of parallel and other modern computer architectures.

In the final Subsection 4.4 we have sketched how piecewise smoothness arising from the Lipschitz elementals abs, min, and max can be handled very naturally by piecewise linearization. In effect one obtains a generalized Taylor approximation by a piecewise linear model with a uniform error of order 2 in terms of the distance to the reference point $\mathring{x}$. This model varies Lipschitz continuously w.r.t $\mathring{x}$ and can be used for successive piecewise linearization techniques, to solve piecewise smooth equations and other fundamental tasks of scientific computing. So far the model is computed exclusively in the forward mode and it is not at all clear how the concept of adjoints should be generalized and implemented for nonsmooth F. That scenario is of particular interest in design optimization and optimal control, where good piecewise linear approximations may need to be discontinuous.

REFERENCES

[1] BENNETT CH. 1973. Logical reversibility of computation. *IBM J. Research and Development*, **17**: 525–532.

[2] BERZ M, BISCHOF CH, CORLISS GF & GRIEWANK A. (Eds). 1996. Computational Differentiation: Techniques, Applications and Tools. Proceedings of the SIAM Workshop on the Automatic Differentiation of Algorithms in Santa Fe, New Mexico, SIAM, Philadelphia.

[3] BISCHOF CH H, BÜCKER HM, HOVLAND PD, NAUMANN U & UTKE J. (Eds). 2008. Advances in Automatic Differentiation. *Lecture Notes in Computational Science and Engineering*, **64**, Springer Berlin.

[4] BISCHOF CH H, BÜCKER HM, LANG B, RASCH A & VEHRESCHILD A. 2002. *Combining Source Transformation and Operator Overloading Techniques to Compute Derivatives for MATLAB Programs*. Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2002), IEEE Computer Society.

[5] BÜCKER HM, CORLISS GF, HOVLAND PD, NAUMANN U & NORRIS B. (Eds). 2005. Automatic Differentiation: Applications, Theory, and Implementations. *Lecture Notes in Computational Science and Engineering*, **50**, Springer New York.

[6] CHANG YF & CORLISS GF. 1994. ATOMFT: Solving ODEs and DAEs using Taylor series. *Comput. Math. Appl.*, **28**: 209–233.

[7] COLEMAN TF & MORÉ JJ. 1983. Estimation of sparse Jacobian matrices and graph coloring problems. *SIAM J. Numer. Anal.*, **20**: 187–209.

[8] COLEMAN TF & VERMA A. 1996. *Structure and efficient Jacobian calculation*, in [2], 149–159.

[9] CORLISS GF, FAURE CH, GRIEWANK A, HASCOËT L & NAUMANN U. (Eds.) 2002. Automatic Differentiation of Algorithms: From Simulation to Optimization. Conference proceedings, Nice 2000, *Computer and Information Science*, Springer New York.

[10] FORTH SH, HOVLAND PD, PHIPPS E, UTKE J & WALTHER A. (Eds.) 2012. Recent Advances in Algorithmic Differentiation. *Lecture Notes in Computational Science and Engineering*, **87**, Springer Berlin.

[11] GEBREMEDHIN AH, NGUYEN D, PATWARY MMA & POTHEN A. 2013. ColPack: Software for Graph Coloring and Related Problems in Scientific Computing. *ACM Transactions on Mathematical Software*, **40**(1): Art. 1, 31.

[12] GRIEWANK A & CORLISS GF. (Eds.) 1991. Automatic Differentiation of Algorithms: Theory, Implementation, and Application. Proceedings of the SIAM Workshop on the Automatic Differentiation of Algorithms, Breckenridge, Colorado, SIAM, Philadelphia.

[13] GRIEWANK A, UTKE J & WALTHER A. 2000. Evaluating higher derivative tensors by forward propagation of univariate Taylor series. *Mathematics of Computation*, **69**(231): 1117–1130.

[14] GRIEWANK A & NAUMANN U. 2002. Accumulating Jacobians by Vertex, Edge or Face Elimination. *Proceedings of CARI'02*, 375–383.

[15] GRIEWANK A & WALTHER A. 2008. *Principles and Techniques of Algorithmic Differentiation*, Second Edition, SIAM.

[16] GRIEWANK A. 2012. Who invented the Reverse Mode of Differentiation? In: MARTIN GRÖTSCHEL, (Ed.). *Optimization Stories*, Documenta Matematica, Extra Volume ISMP (2012): 389–400.

[17] GRIEWANK A. 2013. On Stable Picewise Linearization and Generalized Differentiation. *Optimization Methods and Software*, **28**(6): 1139–1178.

[18] GRIEWANK A, BERNT J-U, RADONS M & STREUBEL T. 2014. Solving piecewise lineare equations in abs-normal form. Sybmitted to *Linear Algebra and Applications.*

[19] GRIEWANK A & STROGIES N. 2014. Efficient Evaluation of Sparse Jacobians by Matrix Compression, Part I: Motivation and Theory. *Proceeding of the 2nd International Conference on Mathematics, Computational and Statistical Sciences*, pp. 33–40.

[20] KHAN KA & BARTON PI. 2013. Evaluating an element of the Clarke generalized Jacobian of a composite piecewise differentiable function. *ACM Transactions on Mathematical Software (TOMS)*, **39**(4): 23:1–23:28.

[21] KORELC J. 1997. Automatic generation of finite-element code by simultaneous optimization of expressions. *Theoretical Computer Science*, **187**: 231–248.

[22] KLATTE D & KUMMER B. 2002. *Nonsmooth equations in optimization*, volume 60 of *Nonconvex Optimization and its Applications*. Kluwer Academic Publishers, Dordrecht, 2002. Regularity, calculus, methods and applications.

[23] LINNAINMAA S. 1970. *The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding error*, Master's thesis (in Finnish), Department of Computer Science, University of Helsinki.

[24] LOGG A, MARDAL K-A & WELLS GN ET AL. 2012. Automated Solution of Differential Equations by the Finite Element Method. Springer.

[25] MORDUKHOVICH BS. 2006. *Variational analysis and generalized differentiation. I*, volume 330 of *Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences]*. Springer-Verlag, Berlin, 2006. Basic theory.

[26] NAUMANN U. 2008. Optimal Jacobian accumulation is NP-complete. *Math. Program.*, **112**: 427–441.

[27] NAUMANN U. 2012. *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*, SIAM, *Software, Environments, and Tools*, Philadelphia.

[28] NESTEROV YU. 2005. Lexicographic differentiation of nonsmooth functions. *Mathematical programming*, **104**(2-3): 669–700, Springer.

[29] OSTROVSKII GM, VOLIN YU M & BORISOV WW. 1971. Über die Berechnung von Ableitungen. *Wiss. Z. Tech. Hochschule für Chemie*, **13**: 382–384.

[30] PHIPPS E, CASEY R & GUCKENHEIMER J. 2005. Periodic Orbits of Hybrid Systems and Parameter Estimation via AD. In: [5], 211–223.

[31] SCHOLTES ST. 2012. Introduction to piecewise differentiable equations. *Springer Briefs in Optimization*, Springer Heidelberg.

[32] THACKER WC. 1991. *Automatic differentiation from an oceanographer's perspective*, in [12], pp. 191–201.

[33] WALTHER A, KULSHRESHTHA K & GRIEWANK A. 2012. On the Numerical Stability of Algorithmic Differentiation. *Computing*, **94**: 125–149.

[34] WANNER G. 1969. *Integration gewöhnlicher Differentialgleichnugen, Lie Reihen, Runge-Kutta-Methoden* **XI**, B.I-Hochschulskripten, no. 831/831a, Bibliogr. Inst., Mannheim-Zürich, Germany.

[35] WILKINSON J. 1971. Modern error analysis. *SIAM Rev.*, **13**: 548–568.

[36] WOLFE P. 1982. Checking the calculation of gradients. *ACM Trans. Math. Softw.*, **8**: 337–343.