Automatic differentiation

In mathematics and computer algebra, **automatic differentiation** (**AD**), also called **algorithmic differentiation** or **computational differentiation**,[1][2] is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. AD exploits the fact that every computer program, no matter how complicated, executes a sequence of elementary arithmetic operations (addition, subtraction, multiplication, division, etc.) and elementary functions (exp, log, sin, cos, etc.). By applying the chain rule repeatedly to these operations, derivatives of arbitrary order can be computed automatically, accurately to working precision, and using at most a small constant factor more arithmetic operations than the original program.

Automatic differentiation is not:

- Symbolic differentiation, nor
- Numerical differentiation (the method of finite differences).

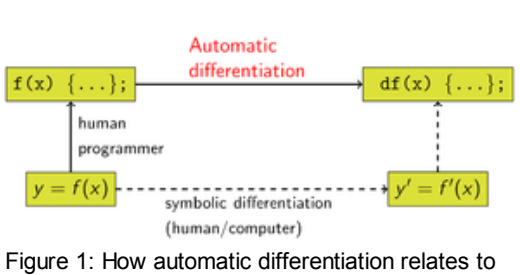

Figure 1: How automatic differentiation relates to symbolic differentiation

These classical methods run into problems: symbolic differentiation leads to inefficient code (unless done carefully) and faces the difficulty of converting a computer program into a single expression, while numerical differentiation can introduce round-off errors in the discretization process and cancellation. Both classical methods have problems with calculating higher derivatives, where the complexity and errors increase. Finally, both classical methods are slow at computing the partial derivatives of a function with respect to *many* inputs, as is needed for gradient-based optimization algorithms. Automatic differentiation solves all of these problems, at the expense of introducing more software dependencies.

Contents

The chain rule, forward and reverse accumulation

Fundamental to AD is the decomposition of differentials provided by the chain rule. For the simple composition $y = f(g(h(x))) = f(g(h(w_0))) = f(g(w_1)) = f(w_2) = w_3$ the chain rule gives

$$\frac{dy}{dx} = \frac{dy}{dw_2}\frac{dw_2}{dw_1}\frac{dw_1}{dx}$$

Usually, two distinct modes of AD are presented, **forward accumulation** (or **forward mode**) and **reverse accumulation** (or **reverse mode**). Forward accumulation specifies that one traverses the chain rule from inside to outside (that is, first compute dw_1/dx and then dw_2/dw_1 and at last dy/dw_2), while reverse accumulation has the traversal from outside to inside (first compute dy/dw_2 and then dw_2/dw_1 and at last dw_1/dx). More succinctly,

1. **forward accumulation** computes the recursive relation: $\frac{dw_i}{dx} = \frac{dw_i}{dw_{i-1}}\frac{dw_{i-1}}{dx}$ with $w_3 = y$, and,
2. **reverse accumulation** computes the recursive relation: $\frac{dy}{dw_i} = \frac{dy}{dw_{i+1}}\frac{dw_{i+1}}{dw_i}$ with $w_0 = x$.

Generally, both forward and reverse accumulation are specific manifestations of applying the operator of program composition, with the appropriate one of the two mappings (w, y) being fixed.

Forward accumulation

In forward accumulation AD, one first fixes the *independent variable* to which differentiation is performed and computes the derivative of each sub-expression recursively. In a pen-and-paper calculation, one can do so by repeatedly substituting the derivative of the *inner* functions in the chain rule:

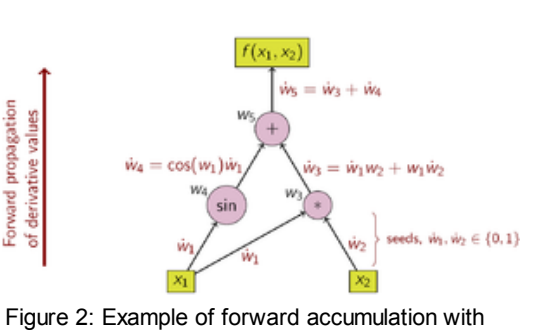

Figure 2: Example of forward accumulation with computational graph

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_{n-1}}\frac{\partial w_{n-1}}{\partial x} = \frac{\partial y}{\partial w_{n-1}}\left(\frac{\partial w_{n-1}}{\partial w_{n-2}}\frac{\partial w_{n-2}}{\partial x}\right) = \frac{\partial y}{\partial w_{n-1}}\left(\frac{\partial w_{n-1}}{\partial w_{n-2}}\left(\frac{\partial w_{n-2}}{\partial w_{n-3}}\frac{\partial w_{n-3}}{\partial x}\right)\right) = \cdots$$

This can be generalized to multiple variables as a matrix product of Jacobians.

Compared to reverse accumulation, forward accumulation is very natural and easy to implement as the flow of derivative information coincides with the order of evaluation. One simply augments each variable w with its derivative $\dot{w}$ (stored as a numerical value, not a symbolic expression),

$$\dot{w} = \frac{\partial w}{\partial x}$$

as denoted by the dot. The derivatives are then computed in sync with the evaluation steps and combined with other derivatives via the chain rule.

As an example, consider the function:

$$\begin{aligned} z &= f(x_1, x_2) \\ &= x_1 x_2 + \sin x_1 \\ &= w_1 w_2 + \sin w_1 \\ &= w_3 + w_4 \\ &= w_5 \end{aligned}$$

For clarity, the individual sub-expressions have been labeled with the variables w_i.

The choice of the independent variable to which differentiation is performed affects the *seed* values $\dot{w}_1$ and $\dot{w}_2$. Suppose one is interested in the derivative of this function with respect to x_1. In this case, the seed values should be set to:

$$\begin{aligned} \dot{w}_1 &= \frac{\partial x_1}{\partial x_1} = 1 \\ \dot{w}_2 &= \frac{\partial x_2}{\partial x_1} = 0 \end{aligned}$$

With the seed values set, one may then propagate the values using the chain rule as shown in the table below. Figure 2 shows a pictorial depiction of this process as a computational graph.

Operations to compute value	Operations to compute derivative
$w_1 = x_1$	$\dot{w}_1 = 1$ (seed)
$w_2 = x_2$	$\dot{w}_2 = 0$ (seed)
$w_3 = w_1 \cdot w_2$	$\dot{w}_3 = w_2 \cdot \dot{w}_1 + w_1 \cdot \dot{w}_2$
$w_4 = \sin w_1$	$\dot{w}_4 = \cos w_1 \cdot \dot{w}_1$
$w_5 = w_3 + w_4$	$\dot{w}_5 = \dot{w}_3 + \dot{w}_4$

To compute the gradient of this example function, which requires the derivatives of f with respect to not only x_1 but also x_2, one must perform an *additional* sweep over the computational graph using the seed values $\dot{w}_1 = 0; \dot{w}_2 = 1$.

The computational complexity of one sweep of forward accumulation is proportional to the complexity of the original code.

Forward accumulation is more efficient than reverse accumulation for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ with $m \gg n$ as only n sweeps are necessary, compared to m sweeps for reverse accumulation.

Reverse accumulation

In reverse accumulation AD, one first fixes the *dependent variable* to be differentiated and computes the derivative *with respect to* each sub-expression recursively. In a pen-and-paper calculation, one can perform the equivalent by repeatedly substituting the derivative of the *outer* functions in the chain rule:

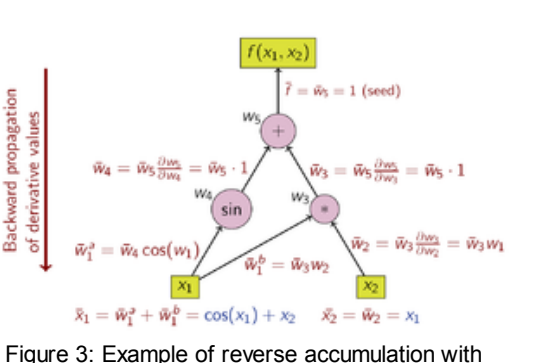

Figure 3: Example of reverse accumulation with computational graph

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1}\frac{\partial w_1}{\partial x} = \left(\frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} = \left(\left(\frac{\partial y}{\partial w_3}\frac{\partial w_3}{\partial w_2}\right)\frac{\partial w_2}{\partial w_1}\right)\frac{\partial w_1}{\partial x} = \cdots$$

In reverse accumulation, the quantity of interest is the *adjoint*, denoted with a bar ($\bar{w}$); it is a derivative of a chosen dependent variable with respect to a subexpression w:

$$\bar{w} = \frac{\partial y}{\partial w}$$

Reverse accumulation traverses the chain rule from outside to inside, or in the case of the computational graph in Figure 3, from top to bottom. The example function is scalar-valued, and thus there is only one seed for the derivative computation, and only one sweep of the computational graph is needed in order to calculate the (two-component) gradient. This is only half the work when compared to forward accumulation, but reverse accumulation requires the storage of the intermediate variables w_i as well as the instructions that produced them in a data structure known as a Wengert list (or "tape"),[3][4] which may represent a significant memory issue if the computational graph is large. This can be mitigated to some extent by storing only a subset of the intermediate variables and then reconstructing the necessary work variables by repeating the evaluations, a technique known as checkpointing.

The operations to compute the derivative using reverse accumulation are shown in the table below (note the reversed order):

Operations to compute derivative
$\bar{w}_5 = 1$ (seed)
$\bar{w}_4 = \bar{w}_5$
$\bar{w}_3 = \bar{w}_5$
$\bar{w}_2 = \bar{w}_3 \cdot w_1$
$\bar{w}_1 = \bar{w}_3 \cdot w_2 + \bar{w}_4 \cdot \cos w_1$

The data flow graph of a computation can be manipulated to calculate the gradient of its original calculation. This is done by adding an adjoint node for each primal node, connected by adjoint edges which parallel the primal edges but flow in the opposite direction. The nodes in the adjoint graph represent multiplication by the derivatives of the functions calculated by the nodes in the primal. For instance, addition in the primal causes fanout in the adjoint; fanout in the primal causes addition in the adjoint; a unary function $y = f(x)$ in the primal causes $\bar{x} = \bar{y} f'(x)$ in the adjoint; etc.

Reverse accumulation is more efficient than forward accumulation for functions $f : \mathbb{R}^n \to \mathbb{R}^m$ with $m \ll n$ as only m sweeps are necessary, compared to n sweeps for forward accumulation.

Reverse mode AD was first published in 1970 by Seppo Linnainmaa in his master thesis.[5][6][7]

Backpropagation of errors in multilayer perceptrons, a technique used in machine learning, is a special case of reverse mode AD.[2]

Beyond forward and reverse accumulation

Forward and reverse accumulation are just two (extreme) ways of traversing the chain rule. The problem of computing a full Jacobian of $f : \mathbb{R}^n \to \mathbb{R}^m$ with a minimum number of arithmetic operations is known as the *optimal Jacobian accumulation* (OJA) problem, which is NP-complete.[8] Central to this proof is the idea that there may exist algebraic dependencies between the local partials that label the edges of the graph. In particular, two or more edge labels may be recognized as equal. The complexity of the problem is still open if it is assumed that all edge labels are unique and algebraically independent.

Automatic differentiation using dual numbers

Forward mode automatic differentiation is accomplished by augmenting the algebra of real numbers and obtaining a new arithmetic. An additional component is added to every number which will represent the derivative of a function at the number, and all arithmetic operators are extended for the augmented algebra. The augmented algebra is the algebra of dual numbers. This approach was generalized by the theory of operational calculus on programming spaces (see Analytic programming space), through tensor algebra of the dual space.

Replace every number x with the number $x + x'\varepsilon$, where x' is a real number, but ε is an abstract number with the property $\varepsilon^2 = 0$ (an infinitesimal; see *Smooth infinitesimal analysis*). Using only this, we get for the regular arithmetic

$$\begin{aligned} (x + x'\varepsilon) + (y + y'\varepsilon) &= x + y + (x' + y')\varepsilon \\ (x + x'\varepsilon) \cdot (y + y'\varepsilon) &= xy + xy'\varepsilon + yx'\varepsilon + x'y'\varepsilon^2 = xy + (xy' + yx')\varepsilon \end{aligned}$$

and likewise for subtraction and division.

Now, we may calculate polynomials in this augmented arithmetic. If $P(x) = p_0 + p_1 x + p_2 x^2 + \cdots + p_n x^n$, then

$$\begin{aligned} P(x + x'\varepsilon) &= p_0 + p_1(x + x'\varepsilon) + \cdots + p_n(x + x'\varepsilon)^n \\ &= p_0 + p_1 x + \cdots + p_n x^n + p_1 x'\varepsilon + 2p_2 x x'\varepsilon + \cdots + n p_n x^{n-1} x'\varepsilon \\ &= P(x) + P^{(1)}(x) x'\varepsilon \end{aligned}$$

where $P^{(1)}$ denotes the derivative of P with respect to its first argument, and x', called a *seed*, can be chosen arbitrarily.

The new arithmetic consists of ordered pairs, elements written $\langle x, x' \rangle$, with ordinary arithmetics on the first component, and first order differentiation arithmetic on the second component, as described above. Extending the above results on polynomials to analytic functions we obtain a list of the basic arithmetic and some standard functions for the new arithmetic:

$$\begin{aligned} \langle u,u'\rangle + \langle v,v'\rangle &= \langle u+v, u'+v'\rangle \\ \langle u,u'\rangle - \langle v,v'\rangle &= \langle u-v, u'-v'\rangle \\ \langle u,u'\rangle * \langle v,v'\rangle &= \langle uv, u'v+uv'\rangle \\ \langle u,u'\rangle / \langle v,v'\rangle &= \left\langle \frac{u}{v}, \frac{u'v-uv'}{v^2} \right\rangle \quad (v \ne 0) \\ \sin\langle u,u'\rangle &= \langle \sin(u), u'\cos(u)\rangle \\ \cos\langle u,u'\rangle &= \langle \cos(u), -u'\sin(u)\rangle \\ \exp\langle u,u'\rangle &= \langle \exp u, u'\exp u\rangle \\ \log\langle u,u'\rangle &= \langle \log(u), u'/u\rangle \quad (u > 0) \\ \langle u,u'\rangle^k &= \langle u^k, ku^{k-1}u'\rangle \quad (u \ne 0) \\ \left|\langle u,u'\rangle\right| &= \langle |u|, u'\operatorname{sign} u\rangle \quad (u \ne 0) \end{aligned}$$

and in general for the primitive function g,

$$g(\langle u,u'\rangle, \langle v,v'\rangle) = \langle g(u,v), g_u(u,v)u' + g_v(u,v)v'\rangle$$

where g_u and g_v are the derivatives of g with respect to its first and second arguments, respectively.

When a binary basic arithmetic operation is applied to mixed arguments—the pair $\langle u, u'\rangle$ and the real number c—the real number is first lifted to $\langle c, 0\rangle$. The derivative of a function $f : \mathbb{R} \to \mathbb{R}$ at the point x_0 is now found by calculating $f(\langle x_0, 1\rangle)$ using the above arithmetic, which gives $\langle f(x_0), f'(x_0)\rangle$ as the result.

Vector arguments and functions

Multivariate functions can be handled with the same efficiency and mechanisms as univariate functions by adopting a directional derivative operator. That is, if it is sufficient to compute $y' = \nabla f(x) \cdot x'$, the directional derivative $y' \in \mathbb{R}^m$ of $f : \mathbb{R}^n \to \mathbb{R}^m$ at $x \in \mathbb{R}^n$ in the direction $x' \in \mathbb{R}^n$, this may be calculated as $(\langle y_1, y'_1\rangle, \ldots, \langle y_m, y'_m\rangle) = f(\langle x_1, x'_1\rangle, \ldots, \langle x_n, x'_n\rangle)$ using the same arithmetic as above. If all the elements of ∇f are desired, then n function evaluations are required. Note that in many optimization applications, the directional derivative is indeed sufficient.

High order and many variables

The above arithmetic can be generalized to calculate second order and higher derivatives of multivariate functions. However, the arithmetic rules quickly grow very complicated: complexity will be quadratic in the highest derivative degree. Instead, truncated Taylor polynomial algebra can be used. The resulting arithmetic, defined on generalized dual numbers, allows to efficiently compute using functions as if they were a new data type. Once the Taylor polynomial of a function is known, the derivatives are easily extracted. A rigorous, general formulation is achieved through the tensor series expansion using operational calculus on programming spaces.

Operational calculus on programming spaces

Operational calculus on programming spaces [9] provides differentiable programming with formal semantics through an algebra of higher-order constructs. It can thus be used to express the concepts underlying automatic differentiation.

Differentiable programming space

A differentiable programming space $\mathcal{P}_0$ is any subspace of $\mathcal{F}_0 : \mathcal{V} \to \mathcal{V}$ such that

$$\partial\mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*),$$

where $T(\mathcal{V}^*)$ is the tensor algebra of the dual space $\mathcal{V}^*$. When all elements of $\mathcal{P}_0$ are analytic, we call $\mathcal{P}_0$ an analytic programming space.

> **Theorem**. Any differentiable programming space $\mathcal{P}_0$ is an infinitely differentiable programming space, meaning that
> $$\partial^k\mathcal{P}_0 \subset \mathcal{P}_0 \otimes T(\mathcal{V}^*)$$
> for any $k \in \mathbb{N}$. If all elements of $\mathcal{P}_0$ are analytic, than so are the elements of $\mathcal{P}_n$.

Definition. Let $\mathcal{P}_0$ be a differentiable programming space. The space $\mathcal{P}_n < \mathcal{F}_n : \mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*)$ spanned by $\mathcal{D}^n\mathcal{P}_0$ over K, where $\mathcal{D}^n = \{\partial^k | 0 \le k \le n\}$, is called a differentiable programming space of order n.

> **Corollary**. A differentiable programming space of order n, $\mathcal{P}_n : \mathcal{V} \to \mathcal{V} \otimes T(\mathcal{V}^*)$, can be embedded into the tensor product of the function space $\mathcal{P}_0 : \mathcal{V} \to \mathcal{V}$ and the subspace $T_n(\mathcal{V}^*)$ of the tensor algebra of the dual of the virtual space $\mathcal{P}$. By taking the limit as $n \to \infty$, we consider
> $$\mathcal{P}_\infty < \mathcal{P}_0 \otimes \mathcal{T}(\mathcal{V}^*),$$
> where
> $$\mathcal{T}(\mathcal{V}^*) = \prod_{k=0}^{\infty} (\mathcal{V}^*)^{\otimes k}$$
> is the tensor series algebra, the algebra of the infinite formal tensor series, which is a completion of the tensor algebra $T(\mathcal{V}^*)$ in suitable topology.

Proofs can be found in.[9]

This means that we can represent calculation of derivatives of the map $P : \mathcal{V} \to \mathcal{V}$, with only one mapping τ. We define the operator τ_n as a direct sum of operators

$$\tau_n = 1 + \partial + \partial^2 + \cdots + \partial^n$$

The image $\tau_k P(\mathbf{x})$ is a multitensor of order k, which is a direct sum of the maps value and all derivatives of order $n \le k$, all evaluated at the point $\mathbf{x}$

$$\tau_k P(\mathbf{x}) = P(\mathbf{x}) + \partial_{\mathbf{x}} P(\mathbf{x}) + \partial^2_{\mathbf{x}} P(\mathbf{x}) + \cdots + \partial^k_{\mathbf{x}} P(\mathbf{x}).$$

The operator τ_n satisfies the recursive relation.

$$\tau_{k+1} = 1 + \partial\tau_k,$$

that can be used to recursively construct programming spaces of arbitrary order. Only explicit knowledge of $\tau : \mathcal{P}_0 \to \mathcal{P}_1$ is required for the construction of $\mathcal{P}_n$ from $\mathcal{P}_1$, which is evident from the above theorem.

Virtual tensor machine

The paper [9] proposed an abstract virtual machine capable of constructing and implementing the theory. Such a machine provides a framework for analytic study of algorithmic procedures through algebraic means.

> **Claim**. The tuple $(\mathcal{V}, \mathcal{P}_0)$ and the belonging tensor series algebra are sufficient conditions for the existence and construction of infinitely differentiable programming spaces $\mathcal{P}_\infty$, through linear combinations of elements of $\mathcal{P}_0 \otimes \mathcal{T}(\mathcal{V}^*)$.

This claim allows a simple definition of such a machine.

Definition (Virtual tensor machine). The tuple $M = \langle \mathcal{V}, \mathcal{P}_0 \rangle$ is a virtual tensor machine, where

- $\mathcal{V}$ is a finite dimensional vector space
- $\mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*)$ is the virtual memory space
- $\mathcal{P}_0$ is an analytic programming space over $\mathcal{V}$

Tensor series expansion

Expansion into a series offers valuable insights into programs through methods of analysis.

There exists a space spanned by the set $\mathcal{D}^n = \{\partial^k | 0 \le k \le n\}$ over a field K. Thus, the expression

$$e^{h\partial} = \sum_{n=0}^{\infty} \frac{(h\partial)^n}{n!}$$

is well defined. The operator $e^{h\partial}$ is a mapping between function spaces

$$e^{h\partial} : \mathcal{P} \to \mathcal{P}_\infty.$$

It also defines a map

$$e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*),$$

by taking the image of the map $e^{h\partial}(P)$ at a certain point $\mathbf{v} \in \mathcal{V}$.

We may construct a map from the space of programs, to the space of polynomials. Note that the space of multivariate polynomials $\mathcal{V} \to K$ is isomorphic to symmetric algebra $S(\mathcal{V}^*)$, which is in turn a quotient of tensor algebra $T(\mathcal{V}^*)$. To any element of $\mathcal{V} \otimes T(\mathcal{V}^*)$ one can attach corresponding element of $\mathcal{V} \otimes S(\mathcal{V}^*)$ namely a polynomial map $\mathcal{V} \to \mathcal{V}$. Thus, we consider the completion of the symmetric algebra $S(\mathcal{V}^*)$ as the *Formal power series* $\mathcal{S}(\mathcal{V}^*)$, which is in turn isomorphic to a quotient of *tensor series algebra* $\mathcal{T}(\mathcal{V}^*)$, arriving at

$$e^{h\partial} : \mathcal{P} \times \mathcal{V} \to \mathcal{V} \otimes \mathcal{S}(\mathcal{V}^*)$$

For any element $\mathbf{v}_0 \in \mathcal{V}$, the expression $e^{h\partial}(\cdot, \mathbf{v}_0)$ is a map $\mathcal{P} \to \mathcal{V} \otimes \mathcal{S}(\mathcal{V}^*)$, mapping a program to a *Formal power series*. We can express the correspondence between multi-tensors in $\mathcal{V} \otimes T(\mathcal{V}^*)$ and polynomial maps $\mathcal{V} \to \mathcal{V}$ given by multiple contractions for all possible indices.

> **Theorem**. For a program $P \in \mathcal{P}$ the expansion into an infinite tensor series at the point $\mathbf{v}_0 \in \mathcal{V}$ is expressed by multiple contractions
> $$P(\mathbf{v}_0 + h\mathbf{v}) = \left((e^{h\partial}P)(\mathbf{v}_0)\right) \cdot \mathbf{v} = \sum_{n=0}^{\infty} \frac{h^n}{n!} \partial^n P(\mathbf{v}_0) \cdot (\mathbf{v}^{\otimes n})$$

Proof can be found in.[9] Evaluated at $h = 1$, the operator is a generalization of the Shift operator widely used in physics. For a specific $v_0 \in \mathcal{V}$ it is here on denoted by

$$e^{\partial}|_{v_0} : \mathcal{P} \to \mathcal{V} \otimes \mathcal{T}(\mathcal{V}^*).$$

When the choice of $v_0 \in \mathcal{V}$ is arbitrary, we omit it from expressions for brevity. Following this work, a similar approach was taken by others [10].

Operator of program composition

Theory offers a generalization of both forward and reverse mode of automatic differentiation to arbitrary order, under a single invariant operator in the theory. This condenses complex notions to simple expressions allowing meaningful manipulations before being applied to a particular programming space.

> **Theorem**. Composition of maps $\mathcal{P}$ is expressed as
> $$e^{h\partial}(f \circ g) = \exp(\partial_f e^{h\partial_g})(g, f)$$
> where $\exp(\partial_f e^{h\partial_g}) : \mathcal{P} \times \mathcal{P} \to \mathcal{P}_\infty$ is an operator on pairs of maps (g, f), where ∂_g is applied to g and ∂_f to f.

Proof can be found in.[9]

Both forward and reverse mode (generalized to arbitrary order) are obtainable using this operator, by fixing the appropriate one of the two maps. This generalizes both concepts under a single operator in the theory. For example, by considering projections of the operator onto the space spanned by $\mathcal{D} = \{1, \partial\}$, and fixing the second map g, we retrieve the basic first order forward mode of automatic differentiation, or reverse mode, by fixing f.

Thus the operator alleviates the need for explicit implementation of the higher order chain rule (see Faà di Bruno's formula), as it is encoded in the structure of the operator itself, which can be efficiently implemented by manipulating its generating map (see [9]).

Order reduction for nested applications

It is useful to be able to use the k-th derivative of a program $P \in \mathcal{P}$ as part of a different differentiable program P_1. As such, we must be able to treat the derivative itself as a differentiable program $P'^k \in \mathcal{P}$, while only coding the original program P.

> **Theorem**. There exists a reduction of order map $\phi : \mathcal{P}_n \to \mathcal{P}_{n-1}$ satisfying
> $$\forall P_1 \in \mathcal{P}_0 \quad \exists P_2 \in \mathcal{P}_0 \quad \phi^k \circ e^{\partial}_n(P_1) = e^{\partial}_{n-k}(P_2)$$
> for each $n \ge 1$, where e^{∂}_n is the projection of the operator e^{∂} onto the set $\mathcal{D}^n = \{\partial^k | 0 \le k \le n\}$.

By the above Theorem, n-differentiable k-th derivatives of a program $P \in \mathcal{P}_0$ can be extracted by

$${}^n P^{k\prime} = \phi^k \circ e^{\partial}_{n+k}(P) \in \mathcal{P}_n.$$

Thus, we gained the ability of writing a differentiable program acting on derivatives of another program, stressed as crucial by other authors.[11]

Implementation

Forward-mode AD is implemented by a nonstandard interpretation of the program in which real numbers are replaced by dual numbers, constants are lifted to dual numbers with a zero epsilon coefficient, and the numeric primitives are lifted to operate on dual numbers. This nonstandard interpretation is generally implemented using one of two strategies: *source code transformation* or *operator overloading*.

Source code transformation (SCT)

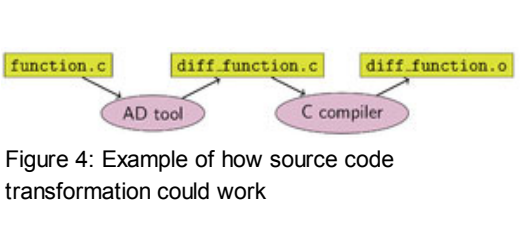

Figure 4: Example of how source code transformation could work

The source code for a function is replaced by an automatically generated source code that includes statements for calculating the derivatives interleaved with the original instructions.

Source code transformation can be implemented for all programming languages, and it is also easier for the compiler to do compile time optimizations. However, the implementation of the AD tool itself is more difficult.

Operator overloading (OO)

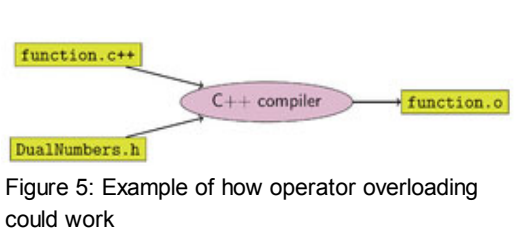

Figure 5: Example of how operator overloading could work

Operator overloading is a possibility for source code written in a language supporting it. Objects for real numbers and elementary mathematical operations must be overloaded to cater for the augmented arithmetic depicted above. This requires no change in the form or sequence of operations in the original source code for the function to be differentiated, but often requires changes in basic data types for numbers and vectors to support overloading and often also involves the insertion of special flagging operations.

Operator overloading for forward accumulation is easy to implement, and also possible for reverse accumulation. However, current compilers lag behind in optimizing the code when compared to forward accumulation.

Operator overloading, for both forward and reverse accumulation, can be well-suited to applications where the objects are vectors of real numbers rather than scalars. This is because the tape then comprises vector operations; this can facilitate computationally efficient implementations where each vector operation performs many scalar operations. Vector adjoint algorithmic differentiation (vector AAD) techniques may be used, for example, to differentiate values calculated by Monte-Carlo simulation.

Examples of operator-overloading implementations of automatic differentiation in C++ are the Adept and Stan libraries.

References

1. Neidinger, Richard D. (2010). "Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming" (http://academics.davidson.edu/math/neidinger/SIAMRev74362.pdf) (PDF). *SIAM Review*. **52** (3): 545–563. doi:10.1137/080743627 (https://doi.org/10.1137/080743627).
2. Baydin, Atilim Gunes; Pearlmutter, Barak; Radul, Alexey Andreyevich; Siskind, Jeffrey (2018). "Automatic differentiation in machine learning: a survey" (http://jmlr.org/papers/v18/17-468.html). *Journal of Machine Learning Research*. **18**: 1–43.
3. R.E. Wengert (1964). "A simple automatic derivative evaluation program". *Comm. ACM*. **7**: 463–464. doi:10.1145/355586.364791 (https://doi.org/10.1145/355586.364791).
4. Bartholomew-Biggs, Michael; Brown, Steven; Christianson, Bruce; Dixon, Laurence (2000). "Automatic differentiation of algorithms" (http://ac.els-cdn.com/S0377042700004222/1-s2.0-S0377042700004222-main.pdf?_tid=61070b3c-31f1-11e5-a550-00000aacb35f&acdnat=1437735029_c639352923bb07e65307eb75c420d42f) (PDF). *Journal of Computational and Applied Mathematics*. **124** (1-2): 171–190. Bibcode:2000JCoAM.124..171B (http://adsabs.harvard.edu/abs/2000JCoAM.124..171B). doi:10.1016/S0377-0427(00)00422-2 (https://doi.org/10.1016/S0377-0427%2800%2900422-2).
5. Linnainmaa, Seppo (1970). The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors. Master's Thesis (in Finnish), Univ. Helsinki, 6-7.
6. Linnainmaa, Seppo (1976). Taylor expansion of the accumulated rounding error. BIT Numerical Mathematics, 16(2), 146-160.
7. Griewank, Andreas (2012). Who Invented the Reverse Mode of Differentiation?. Optimization Stories, Documenta Matematica, Extra Volume ISMP (2012), 389-400.
8. Naumann, Uwe (April 2008). "Optimal Jacobian accumulation is NP-complete". *Mathematical Programming*. **112** (2): 427–441. doi:10.1007/s10107-006-0042-z (https://doi.org/10.1007/s10107-006-0042-z).
9. Sajovic, Žiga; Vuk, Martin (2016). "Operational calculus on programming spaces". arXiv:1610.07690 (https://arxiv.org/abs/1610.07690) [math.FA (https://arxiv.org/archive/math.FA)].
10. izzo, Dario; Biscani, Francesci (2016). "Differentiable Genetic Programming". arXiv:1611.04766 (https://arxiv.org/abs/1611.04766) [math.FA (https://arxiv.org/archive/math.FA)].
11. Pearlmutter, Barak A.; Siskind, Jeffrey M (May 2008). "Putting the Automatic Back into AD: Part II". *ECE Technical Reports*.

Literature

- Rall, Louis B. (1981). *Automatic Differentiation: Techniques and Applications*. Lecture Notes in Computer Science. **120**. Springer. ISBN 3-540-10861-0.
- Griewank, Andreas; Walther, Andrea (2008). *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (http://www.ec-securehost.com/SIAM/OT105.html). Other Titles in Applied Mathematics. **105** (2nd ed.). SIAM. ISBN 978-0-89871-659-7.
- Neidinger, Richard (2010). "Introduction to Automatic Differentiation and MATLAB Object-Oriented Programming" (http://academics.davidson.edu/math/neidinger/SIAMRev74362.pdf) (PDF). *SIAM Review*. **52** (3): 545–563. doi:10.1137/080743627 (https://doi.org/10.1137/080743627). Retrieved 2013-03-15.

External links

- www.autodiff.org (http://www.autodiff.org/), An "entry site to everything you want to know about automatic differentiation"
- Automatic Differentiation of Parallel OpenMP Programs (http://www.autodiff.org/?module=Applications&application=HC1)
- Automatic Differentiation, C++ Templates and Photogrammetry (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.7749&rep=rep1&type=pdf)
- Automatic Differentiation, Operator Overloading Approach (http://www.vivlabs.com/subpage_ad.php)
- Compute analytic derivatives of any Fortran77, Fortran95, or C program through a web-based interface (http://tapenade.inria.fr:8080/tapenade/index.jsp) Automatic Differentiation of Fortran programs
- Description and example code for forward Automatic Differentiation in Scala (http://www.win-vector.com/dfiles/AutomaticDifferentiationWithScala.pdf)
- finmath-lib automatic differentiation extensions (http://finmath.net/finmath-lib-automaticdifferentiation-extensions), Automatic differentiation for random variables (Java implementation of the stochastic automatic differentiation).
- Adjoint Algorithmic Differentiation: Calibration and Implicit Function Theorem (http://developers.opengamma.com/quantitative-research/Adjoint-Algorithmic-Differentiation-OpenGamma.pdf)
- C++ Template-based automatic differentiation article (http://www.quantandfinancial.com/2017/02/automatic-differentiation-templated.html) and implementation (https://github.com/omartinsky/QuantAndFinancial/tree/master/autodiff_templated)
- Tangent (https://github.com/google/tangent) Source-to-Source Debuggable Derivatives (https://research.googleblog.com/2017/11/tangent-source-to-source-debuggable.html)
- [1] (http://www.nag.co.uk/doc/techrep/pdf/tr5_10.pdf), Exact First- and Second-Order Greeks by Algorithmic Differentiation
- [2] (http://www.nag.co.uk/Market/articles/adjoint-algorithmic-differentiation-of-gpu-accelerated-app.pdf), Adjoint Algorithmic Differentiation of a GPU Accelerated Application
- [3] (http://www.nag.co.uk/Market/seminars/Uwe_AD_Slides_July13.pdf), Adjoint Methods in Computational Finance Software Tool Support for Algorithmic Differentiation