Algorithmic Differentiation, Functional Programming, and Iterate-to-Fixedpoint

Barak A. Pearlmutter
Department of Computer Science
University of New Mexico
Albuquerque, NM 87131
bap@cs.unm.edu
http://www.cs.unm.edu/~bap/

Algorithmic differentiation (AD) transforms straight line numeric code so that it calculates the derivative of the function originally calculated (Corliss et al., 2001). There are two varieties of AD: forward mode (Wengert, 1964), in which the transformed program calculates the product of the Jacobian of the original program with an input vector, and reverse mode (Speelpenning, 1980; Rall, 1981; Rumelhart et al., 1986), in which it calculates the product of the transpose of the Jacobian with an input vector. The functions produced by the forward and reverse mode AD of a function f can be characterized in terms of the standard derivative $\mathcal{D}f\, x = (df_i(x)/dx_j)_{ij}$ by

$$\begin{aligned} \overrightarrow{\mathcal{J}}\{f\}\, x\, \tilde{x} &= (\mathcal{D}f\, x)\, \tilde{x} \\ \overleftarrow{\mathcal{J}}\{f\}\, x\, \tilde{y} &= (\mathcal{D}f\, x)^T\, \tilde{y} \end{aligned}$$

where these operators have types

$$\begin{aligned} \mathcal{D}&: \quad (\mathbb{R}^n \to \mathbb{R}^m) \to (\mathbb{R}^n \to \mathbb{R}^{mn}) \\ \overrightarrow{\mathcal{J}}&: \quad (\mathbb{R}^n \to \mathbb{R}^m) \to (\mathbb{R}^n \to (\mathbb{R}^n \to \mathbb{R}^m)) \\ \overleftarrow{\mathcal{J}}&: \quad (\mathbb{R}^n \to \mathbb{R}^m) \to (\mathbb{R}^n \to (\mathbb{R}^m \to \mathbb{R}^n)) \end{aligned}$$

We extend the AD transforms to composite types,

$$\begin{aligned} \overrightarrow{\mathcal{J}}&: \quad (\alpha \to \beta) \to (\alpha \to (\overrightarrow{\alpha} \to \overrightarrow{\beta})) \\ \overleftarrow{\mathcal{J}}&: \quad (\alpha \to \beta) \to (\alpha \to (\overleftarrow{\beta} \to \overleftarrow{\alpha})) \end{aligned}$$

where the types $\overrightarrow{\alpha} = \overleftarrow{\alpha} = \alpha$ unless α contains a mapping, for which $\overrightarrow{\alpha \to \beta} = \alpha \to (\overrightarrow{\alpha} \to \overrightarrow{\beta})$ and $\overleftarrow{\alpha \to \beta} = \alpha \to (\overleftarrow{\beta} \to \overleftarrow{\alpha})$. $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ produce functions as efficient as the ones they are given, suffering only a small constant factor overhead in time. While $\overrightarrow{\mathcal{J}}$ is safe-for-space, $\overleftarrow{\mathcal{J}}$ is not. These $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ operators are the natural abstraction and extension of forward and reverse mode AD to a functional context. With them derivatives of higher-order functions become well defined, as in $\overleftarrow{\mathcal{J}}\{\text{map}\}$ or $\overrightarrow{\mathcal{J}}\{\lambda f \lambda g \lambda x . f x (g x)\}$.

Code in which a loop is iterated until some tolerance is reached have been problematic for AD, requiring manual assistance and very rough approximations (Giles, 2001; Gockenbach et al., 2001). We regard such loops as approximate iterate-to-fixedpoint calculations, and consider an explicit iterate-to-fixedpoint operator

$$\mathcal{F}_{\mathbf{a}}\{g(\mathbf{a}, \mathbf{b})\} = g(g(\cdots g(g(\mathbf{a}_0, \mathbf{b}), \mathbf{b}) \cdots, \mathbf{b}), \mathbf{b})$$

We find AD rules for $\mathcal{F}$ which unify and formalize methods in AD, machine learning, and mathematical physics.

$$\begin{aligned} &\overrightarrow{\mathcal{J}}\{\ \mathbf{z} = \mathcal{F}_{\mathbf{a}}\{g(\mathbf{a}, \mathbf{b})\}\ \} \\ &\qquad \Rightarrow \qquad \tilde{\mathbf{z}} = \mathcal{F}_{\tilde{\mathbf{a}}}\{\ \overrightarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z}, \mathbf{b})\}\tilde{\mathbf{a}} + \overrightarrow{\mathcal{J}}_{\mathbf{b}}\{g(\mathbf{z}, \mathbf{b})\}\tilde{\mathbf{b}}\ \} \\ &\overleftarrow{\mathcal{J}}\{\ \mathbf{z} = \mathcal{F}_{\mathbf{a}}\{g(\mathbf{a}, \mathbf{b})\}\ \} \\ &\qquad \Rightarrow \qquad \tilde{\mathbf{b}} = \overleftarrow{\mathcal{J}}_{\mathbf{b}}\{g(\mathbf{z}, \mathbf{b})\} \cdot \mathcal{F}_{\tilde{\mathbf{a}}}\{\ \overleftarrow{\mathcal{J}}_{\mathbf{z}}\{g(\mathbf{z}, \mathbf{b})\}\tilde{\mathbf{a}} + \tilde{\mathbf{z}}\ \} \end{aligned}$$

These extensions open the door to AD implementations with automatic efficient transformation of a broadened class of codes, including functional programs, optimization routines, and many numeric methods used in machine learning. More importantly, they allow many numerical algorithms to be expressed more clearly and succinctly than previously possible.

References

George Corliss, Christèle Faure, Andreas Griewank, Laurent Hascoët, and Uwe Naumann, editors. *Automatic Differentiation: From Simulation to Optimization*. Computer and Information Science. Springer, New York, NY, 2001.

Michael B. Giles. On the iterative solution of adjoint equations. In Corliss et al. (2001), chapter 16, pages 145–151.

Mark S. Gockenbach, Daniel R. Reynolds, and William W. Symes. Automatic differentiation and the adjoint state method. In Corliss et al. (2001), chapter 18, pages 161–166.

Louis B. Rall. *Automatic Differentiation: Techniques and Applications*, volume 120 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1981. ISBN 0–540–10861–0.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back–propagating errors. *Nature*, 323:533–536, 1986.

Bert Speelpenning. *Compiling Fast Partial Derivatives of Functions Given by Algorithms*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana-Champaign, IL, January 1980.

R. E. Wengert. A simple automatic derivative evaluation program. *Comm. ACM*, 7(8):463–464, 1964.