The derivative, as this notion appears in the elementary differential calculus, is a familiar mathematical example of a function for which both [the domain and the range] consist of functions.
—Alonzo Church, 1941

August 15, 2013
By Alexey Radul

Introduction to AUTOMATIC DIFFERENTIATION

Automatic differentiation may be one of the best scientific computing techniques you've never heard of. If you work with computers and real numbers at the same time, I think you stand to benefit from at least a basic understanding of AD, which I hope this article will provide; and even if you are a veteran automatic differentiator, perhaps you might enjoy my take on it.

WHAT IT IS

Wikipedia said it very well:

> Automatic differentiation (AD), also called algorithmic differentiation or computational differentiation, is a set of techniques to numerically evaluate the derivative of a function specified by a computer program ... 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.

This bears repeating: **any derivative or gradient**, of any function you can program, or **of any program** that computes a function, [*] with **machine accuracy** and **ideal asymptotic efficiency**. This is good for

- real-parameter optimization (many good methods are gradient-based)
- sensitivity analysis (local sensitivity = ∂(result)/∂(input))
- physical modeling (forces are derivatives of potentials; equations of motion are derivatives of Lagrangians and Hamiltonians; etc)
- probabilistic inference (e.g., Hamiltonian Monte Carlo)
- machine learning
- and who knows how many other scientific computing applications.

My goal for this article is to provide a clear, concise introduction to the technology of AD. In particular, I look to dispel some of the misconceptions that seem to impede the adoption of AD, but to also warn of the traps that do await the unwary.

CONTENTS

- What automatic differentiation isn't
  - AD is not numerical differentiation
  - AD is not symbolic differentiation
- Forward mode computes directional derivatives

- Reverse mode computes directional gradients
- There are subtleties:
    - Overhead of nonstandard interpretation
    - Perturbation confusion
    - Derivatives of higher-order functions
    - Approximate the derivative, don't differentiate the approximation
    - Leaky abstractions
- There are AD tools out there
- You can learn more about it
- Notes

WHAT IT ISN'T: *Symbolic or Numerical Differentiation*

Everyone who hears the name for the first time always thinks that automatic differentiation is either symbolic differentiation or numerical differentiation, but it's much better.

AD *is not* NUMERICAL DIFFERENTIATION

Numerical differentiation is the technique one would obviously think of from the standard definition of a derivative. Since

$$\frac{df(x)}{dx} = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h},$$

you can clearly approximate the left hand side by evaluating the right hand side at a small but nonzero h. This has the advantage of being blindingly easy to code, but the disadvantages of costing $O(n)$ evaluations of f for gradients in n dimensions, and of catching you between the rock of truncation error and the hard place of roundoff error.[1] Techniques have of course been developed to mitigate these problems,[2] but these techniques increase rapidly in programming complexity.

AD *is not* SYMBOLIC DIFFERENTIATION

Symbolic differentiation is the technique you were taught in Calc 101:

$$\begin{aligned}\frac{d(u(x)+v(x))}{dx} &= \frac{du(x)}{dx} + \frac{dv(x)}{dx}, \\ \frac{d(u(x)v(x))}{dx} &= u(x)\frac{dv(x)}{dx} + \frac{du(x)}{dx}v(x), \\ &\text{etc.}\end{aligned}$$

This technique is perfectly mechanical, given access to the expression that defines the function of interest, so there is no reason it could not be carried out by computer (and, in fact, computer algebra systems like Maxima and Mathematica do implement it).

The advantage of symbolic differentiation is that you already know it, and, if the original f is simple enough, it can produce a legible symbolic description of the derivative, which can be very valuable for further thinking. However, from the perspective of just computing answers, symbolic differentiation suffers from two drawbacks. The first is that your Calc 101 course didn't teach you what to do with

$$\frac{d(\text{make-point}(x, y))}{dx},$$

or other constructs that show up in computer programs. What's the derivative of a data structure anyway (even one as simple as a 2D point)? This question does have a good answer, but it nevertheless makes symbolically differentiating computer programs considerably more subtle than it might at first appear.

The second drawback of symbolic differentiation is evident in the rule for multiplication:

$$\frac{d(u(x)v(x))}{dx} = u(x)\frac{dv(x)}{dx} + \frac{du(x)}{dx}v(x).$$

Notice that if f is a product, then f and df have some computations in common (namely, u and v). Notice also that, on the right hand side, $u(x)$ and $du(x)/dx$ appear separately. If you just proceed to symbolically differentiate u and plug its derivative into the appropriate place, you will have duplicated any computation that appears in common between u and du; and deeply nested duplications are bad. Careless symbolic differentiation can easily produce exponentially large symbolic expressions which take exponentially long to evaluate.

In principle, the exponentially large expression should have lots of duplication, so it ought to be possible to simplify it to something fast afterwards—if you can store it in memory long enough. So to prevent intermediate expression bulge, you would want to interleave the differentiating with the simplifying. You can think of (forward mode) automatic differentiation as a way to organize symbolic differentiation to retain the sharing of intermediate results between the main computation and the derivative; which coincidentally also extends nicely to non-numeric programming constructs.

FORWARD MODE *Computes* DIRECTIONAL DERIVATIVES

One of the wrinkles with using AD is that it comes in two basic flavors (and therefore, advanced usage admits many combinations and variations). The easier basic flavor to describe and understand (and implement and use) is called *forward mode*, which computes directional derivatives.

To do forward mode AD on a program, do a nonstandard interpretation of the program, as follows. Define "dual numbers" as formal truncated Taylor series of the form $x + \varepsilon x'$. Define arithmetic on dual numbers by $\varepsilon^2 = 0$, and by interpreting any non-dual number y as $y + \varepsilon 0$. So:

$$\begin{aligned} (x + \varepsilon x') + (y + \varepsilon y') &= (x + y) + \varepsilon(x' + y') \\ (x + \varepsilon x')(y + \varepsilon y') &= (xy) + \varepsilon(xy' + x'y) \quad \text{No quadratic term} \\ &\text{etc.} \end{aligned}$$

The coefficients of ε should look familiar: they are just the corresponding symbolic derivative rules. We are setting up a regime where

$$f(x + \varepsilon x') = f(x) + \varepsilon f'(x)x'. \qquad (1)$$

In other words, these dual numbers are just data structures for carrying the derivative around together with the undifferentiated answer (called the *primal*). The chain rule works—two applications of (1) give

$$\begin{aligned} f(g(x+\varepsilon x')) &= f(g(x)+\varepsilon g'(x)x') \\ &= f(g(x)) + \varepsilon f'(g(x))g'(x)x', \end{aligned}$$

where the coefficient of ε on the right hand side is exactly the derivative of the composition of f and g. That means that since we implement the primitive operations to respect the invariant (1), all compositions of them will too. This, in turn, means that we can extract the derivative of a function of interest by evaluating it in this nonstandard way on an initial input with a 1 coefficient for ε:

$$\left.\frac{df(x)}{dx}\right|_x = \text{epsilon-coefficient}(\text{dual-version}(f)(x+\varepsilon 1)).$$

This also generalizes naturally to taking directional derivatives of $f : \mathbb{R}^n \to \mathbb{R}^m$.

That's all there is to forward mode, conceptually. There are of course some subtleties, but first let's look at what we have: We have a completely mechanical way to evaluate derivatives at a point. This mechanism works for any function that can be composed out of the primitive operations that we have extended to dual numbers. It turns every arithmetic operation into a fixed number of new arithmetic operations, so the arithmetic cost goes up by a small constant factor. We also do not introduce any gross numerical sins, so we can reasonably expect the accuracy of the derivative to be no worse than that of the primal computation.[3]

This technique also naturally extends to arbitrary program constructs—dual numbers are just data, so things like data structures can just contain them. As long as no arithmetic is done on the dual number, it will just sit around and happily be a dual number; and if it is ever taken out of the data structure and operated on again, then the differentiation will continue. The semantic difficulties around what the derivative of a data structure is only arise at the boundary of AD, if the function of interest f produces a data structure rather than a number, but do not embarrass the internal invariants. If you look at it from the point of view of the dual number, AD amounts to partially evaluating f with respect to that input to derive a symbolic expression, symbolically differentiating that expression, and collapsing the result (in a particular way) to make sure it isn't too big; all interleaved to prevent intermediate expression bulges.

Another way to look at forward mode of a function $f : \mathbb{R}^n \to \mathbb{R}^m$ is to consider the Jacobian (matrix of partial derivatives) $J_x f$ at some point x. The function f perforce consists of a sequence of primitive operations applied to various data in its internal storage. If we call this sequence $f_k \circ \ldots \circ f_3 \circ f_2 \circ f_1$ (for k primitive operations, with data flowing from right to left), then the full Jacobian of f decomposes as

$$Jf = Jf_k \cdot \ldots \cdot Jf_3 \cdot Jf_2 \cdot Jf_1,$$

where the points at which each intermediate Jacobian is taken are determined by the computation of f thus far.[4] Now, the nice thing about this decomposition is that even though the full Jacobian Jf may be a horrible huge dense matrix, each of the pieces Jf_i is necessarily very sparse, because each primitive f_i only operates on a tiny part of the data (in fact, for f_i unary or binary, $Jf_i - I$ will have just one or two non-zero entries, respectively). So we can look at the result of forward-mode automatic differentiation as an implicit representation of the point-indexed family of matrices $J_x f$,[5] in the form of a program that computes matrix-vector products $J_x f \cdot v$ given x and v.

REVERSE MODE *Computes* DIRECTIONAL GRADIENTS

As mentioned in the previous section, forward mode AD applied to a function f amounts to an implicit representation of the Jacobian Jf that takes advantage of the factorization of Jf into extremely sparse factors Jf_i,

$$Jf = Jf_k \cdot \ldots \cdot Jf_3 \cdot Jf_2 \cdot Jf_1.$$

Forward mode uses this decomposition to compute directional derivatives, which are vector products $J_x f \cdot v$. In modern practice, however, *gradients* of functions $f : \mathbb{R}^n \to \mathbb{R}$ are generally much more valuable: they enable multidimensional optimization methods, multiparameter sensitivity analyses, simulations of systems following potentials, etc. The concept for a gradient of $f : \mathbb{R}^n \to \mathbb{R}$ generalizes trivially to directional gradients of $g : \mathbb{R}^n \to \mathbb{R}^m$; for x in $\mathbb{R}^n$ and u in $\mathbb{R}^m$, the directional gradient of g at x with respect to u is just the ordinary gradient at x of $g(x) \cdot u$.

The directional gradient is a symmetric concept to directional derivative: where the directional derivative in the direction v is $J_x f \cdot v$, the directional gradient is $u^T \cdot J_x f$. One might therefore hope that the decomposition

$$Jf = Jf_k \cdot \ldots \cdot Jf_3 \cdot Jf_2 \cdot Jf_1$$

would make a good implicit representation of Jf for directional gradients as well. Indeed it does, but with a twist. The twist is that for directional derivatives, the information from x about the point at which to take each Jacobian flows through this product in the same direction as the information from v about what to multiply by (to wit, right to left). For directional gradients, however, they flow in opposite directions, necessitating computing and storing some explicit representation of the decomposition, based on the point x, first, before being able to multiply through by u^T to compute the desired gradient. This can be done—the method is called *reverse mode*—but it introduces both code complexity and runtime cost in the form of managing this storage (traditionally called the "tape"). In particular, the space requirements of raw reverse mode are proportional to the *runtime* of f. There are various techniques for mitigating this problem, but it remains without an ideal solution.

SUBTLETIES

OVERHEAD *of Nonstandard* INTERPRETATION

You will note that I have been careful to describe the performance of AD in terms of arithmetic operations rather than runtime. While the technique ensures that the amount of arithmetic goes up by no more than a small constant,[6] managing this arithmetic can introduce a lot of overhead if done carelessly. For instance, I introduced forward mode by defining arithmetic on dual numbers. If you implement AD by actually allocating data structures for holding dual numbers, you will incur a great deal of overhead indeed: every arithmetic operation will now involve accessing and allocating memory, which on modern computers is much more expensive than arithmetic. An implementation by operator overloading may also introduce method dispatches with their attendant costs. If this is compared to raw numerical computation of the primal, the slowdown can easily be an order of magnitude or two. How best to manage this problem is the subject of ongoing work, and shapes the design space of available AD tools.

PERTURBATION CONFUSION

The other major implementation issue for AD is the possibility of a class of bugs that are collectively called perturbation confusion. The essential problem is that if two ongoing differentiations affect the same piece of code, the coefficients of the two formal epsilons they introduce (called perturbations) need to be kept distinct. However, it is very easy to make bugs in the AD implementation that confuse them in various ways. This problem especially bedevils the performance-minded implementations, because they try to avoid allocating and checking runtime storage for explicit objects to represent the different perturbations. Such situations arise naturally when nesting AD, that is, taking derivatives of functions that internally take derivatives (or use AD-generated derivatives).

DERIVATIVES *of* HIGHER-ORDER FUNCTIONS

We have so far been discussing directional derivatives and directional gradients of functions $f : \mathbb{R}^n \to \mathbb{R}^m$. If f should internally use data structures or higher-order functions or other programming-language machinery, the nonstandard interpretation view of AD turns out to just scale naturally. In particular, forward mode by operator overloading requires exactly zero additional work, because the dual numbers get carried around in just the right way. Reverse mode takes a little more care, but can also be done.

Given that one knows how to use AD to get derivatives of any

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

it becomes tempting to generalize the interface by defining a vector space structure on all types in one's programming language, and getting directional derivatives of

$$f : \mathbb{R}^n \to \alpha$$

and directional gradients of

$$g : \alpha \to \mathbb{R}^m$$

for all types α. This is especially tempting since the internal workings of AD point the way to such a vector space structure already. Doing this increases the flexibility of AD appreciably, but turns out to entail more subtleties still; because appropriate representations need to be defined for tangents (or cotangents for reverse mode) that are separated from their primals, and because assumptions on which one might have rested a strategy for avoiding perturbation confusion may need to be revised or enforced (especially if α is a function type).

APPROXIMATE *the* DERIVATIVE, *don't* DIFFERENTIATE *the* APPROXIMATION

AD as a concept has one serious weakness that I know of (as opposed to subtle points that implementers need to get right but for which the answers are in principle known), and that is that it does not commute with approximation. That is, AD will give you derivatives of the procedure you actually programmed, which may not be good approximations of the derivatives of the ideal function that your procedure was approximating.

An example will elucidate: let your mind wander for a moment, back to a time when the e^x function was computed by an explicit subroutine, say `exp`, instead of being microcoded in the chip (or even implemented directly in hardware). This subroutine can only compare, add, and multiply, so it must, perforce, compute not the true e^x function but some piecewise-rational approximation to it. Now imagine automatically differentiating some computation that calls this `exp` routine. For lack of any annotation to the contrary, AD will descend into the implementation of `exp`. What will happen to it? The piecewise structure of `exp` will remain, but the rational function in each piece will get differentiated, producing a different rational function (of degree 1 less). This remains an approximation of the derivative of e^x, of course, but we have access to a much better approximation, namely `exp` itself! So AD increased the approximation error in our computation more than was strictly necessary.

Now, this particular example is a non-problem today, because `exp` is always taken as a primitive and (an approximation to) its derivative is specified as part of the input to the AD system. There are, however, other such examples, where some mathematical function f that can only be computed approximately has a well-defined mathematical derivative (often expressed in terms of f itself), and where differentiating an approximation to f produces much worse answers than explicitly approximating the (known) derivative of f. For instance, situations like this arise when f is something like a numerical integration routine being differentiated with respect to either the bounds of integration (the right answer is obvious) or some parameter of the integrand (less obvious). Unfortunately, I know of no AD system that allows the user to add their own higher-order functions to the set of primitives whose derivatives the AD system knows, so this subtlety must stand as a caution to users of AD for now.

LEAKY ABSTRACTIONS

Differentiation is, like I would expect any nonstandard interpretation to be, a pretty invasive procedure to execute upon a function. As such, care must be taken in the implementation of any AD system if its use is not to leak across abstraction boundaries.

The most egregious form of such leakage is what one might call a *caller derives* discipline: when a routine (e.g., a maximum-finder) that accepts a function and wants to know its derivatives simply demands to be called with an additional parameter which computes the requested derivative. Such an interface has the advantage of being simple to code (for the implementer of the maximum-finder) and of not pinning the user to any particular methodology for producing the needed derivatives (in fact, such derivative functions are often calculated and coded by hand). The downsides are that routines like this become ever more difficult to

- compose (what set of derivatives, of what orders and with respect to what inputs, does a minimax routine need? How about a minimax that uses second order derivatives for quadratic methods for the min and the max?),
- interchange (because the entire call stack from client to method needs to be adjusted if a different set of derivative information should be required),

- and use (because every time you wish to apply it to a different function, you have to supply a new set of derivatives)
- correctly (the coordination of having to supply the proper derivative of the proper function is an additional source of programming error).

In contrast, a completely *callee derives* discipline allows the external interfaces of all methods for the same task (such as maximum-finding) to be the same (at least on this point), and invokes AD automatically whenever some method requires the derivative of an input function. The advantages of callee derives are improved software engineering: more complex methods are easier to define, implement, and use; experimentation with different approaches to a problem becomes easier; methods that adaptively choose whether to use derivatives or not become possible. The downside is that some AD system is required, and mechanisms for the user to supply their own derivative computations (should they be better than the automatic ones) become more complex (when they are available at all).

It also appears that some AD systems operating on statically typed base languages occupy an intermediate ground: derivatives of functions of interest can be computed automatically by the callee, but whether (and how much) it does this is apparent from said callee's type signature. The pros and cons of this situation are, in my view, a mix of those of the two extremes between which it lies.

AD TOOL SPACE

Now that you know what automatic differentiation is all about, how can you go about using it? There are a fair number of AD tools out there. Evaluating all of them is beyond the scope of this article, but I want to paint a general picture in broad strokes, and arm you with the considerations wherewith to evaluate available tools on your own.

What does one want of an AD library or tool? There are of course the things one wants of any library or tool—maturity, Freedom, applicability to one's programming language and ecosystem, etc. But in the case of AD, the subtleties discussed in the previous section translate into particular desiderata. You want your tool to get all the subtle points right; stated positively, these desiderata are:

- Availability of reverse mode, as this is much harder to implement (correctly) than forward mode.
- Good performance of the generated derivatives.
- Correct nesting. Empirically, reverse over reverse is particularly tough to implement (though there may be ways to avoid needing it).
- Handling of language features (especially higher-order functions), at least when used internally by the program being differentiated.
- Hooks for custom derivatives (for those cases when you know you can do better than the AD system).
- Enabling proper abstractions.

I know of three classes of mechanisms for implementing the nonstandard interpretation necessary for AD; they vary in which of the above desiderata they make easy or difficult.

OPERATOR OVERLOADING

In this approach, you overload all the arithmetic operations in your language of choice to react to dual numbers appropriately, and you observe that now every function stands ready to compute its values and/or its derivatives, depending on what input you give it.

- Good performance is tough, because operator overloading puts a great deal of pressure on the compiler of the base language to get rid of all the extra dispatches, allocations, and memory references that it introduces. Many of them are not up to the challenge.
- Care must still be taken to avoid perturbation confusion in the presence of nested AD. Explicitly carrying around tags for the perturbations is simple, but adds yet more pressure on the underlying compiler.
- Language features like data structures and higher-order functions are typically handled for free, at least if they only appear internally to the function being differentiated (rather than across its interface). On the other hand, the language has to support operator overloading for this to work in the first place.
- I have not seen systems like this that handle custom derivatives gracefully, though it should be possible to expose the API of the overloading data structures so that the user can teach their own functions to dispatch on them.
- Callee derives is essentially the default with this method; though the underlying language's type system may need to be appeased. On the other hand, said type system could also be good for reducing perturbation confusion and for generating good code (by enabling the underlying compiler to do type-driven optimizations).

Such systems are numerous and include, among others

- r6rs-ad for R6RS Scheme,
- the AD inside Scmutils,
- the ad/fad/rad series of Haskell libraries, and
- FADBAD++ for C++ with templates.

EXTRALINGUISTIC *Source-to-Source* TRANSFORM

This approach is exemplified by Tapenade, ADIFOR, and ADIC. What you do is you write a program that reads source code files of the language you are defining AD for, and some meta-information about what function(s) to differentiate, carries out some appropriate analyses and transformations, and emits source files in that same language that compute the desired derivatives.

- Good performance is arguably more natural with this approach than with operator overloading, because you can, in principle, emit arbitrarily efficient code by optimizing the representations and operations used in your output program. Of course, this means you are essentially writing your own compiler, albeit one specialized to AD.
- Perturbation confusion tends, empirically, to be a big problem for these kinds of systems, presumably because they are aggressive about getting rid of the meta-information that would help to disambiguate perturbations.
- Supporting language features like data structures and higher-order functions becomes that much more work—not only must you es-

sentially reimplement them, you also have to figure out how they must transform under AD. On the other hand, the source language doesn't need to support operator overloading because you aren't using it.

- Custom derivatives are frequently supported by source-to-source transformation tools. I suspect that this is because the tools already follow the program's existing modularity boundaries, so swapping in a custom derivative for one the tool would otherwise produce itself is not difficult. However, these tools do not usually support higher-order functions, so supplying a custom derivative for an integration routine (that may rely upon derivatives of the integrand!) remains an open problem.
- Source-to-source tools naturally lean towards caller-derives, because callee-derives necessarily involves inspection of which functions end up flowing into places that need their derivatives; and modification of those control flows to include said derivative functions.

Source-to-source tools essentially amount to reimplementations of the language on which they operate. If that language is in any way complex or unclear, the tool's semantics are likely to diverge from those of the canonical implementation; and in any case this approach also imposes a nontrivial toolchain cost on its users.

NEW PROGRAMMING LANGUAGE *with* AD *Primitives*

This is the approach taken by Stalingrad, the research system that got me interested in all this, and, in a sense, my own DysVunctional Language.

- Good performance requires little or no additional effort for AD, above and beyond the work of making a performant programming language in the first place.
- Correct nesting still requires work. However, it may be possible to make the underlying compiler aggressive enough to eliminate the performance overhead of tagging the perturbations explicitly in the AD implementation (DVL attempts this).
- Supporting advanced language features again requires no additional work for AD, beyond the work of implementing them in the first place. And the AD implementation's semantics for those features will not diverge from the standard semantics, since they are integrated from the start.
- Custom derivatives remain a challenge; perhaps even a greater one than with other approaches. The whole point of defining a new language is that AD will operate smoothly on all of it; how then to introduce artificial barriers which AD does not cross, but instead uses user-supplied derivatives? Especially if the derivatives in question are themselves to be written in the same language (and need to be amenable to further differentiation, for nesting)?
- The callee-derives style of interface, however, becomes very natural indeed.

The additional disadvantage, of course, is that this approach imposes the technical cost of dealing with your new language's foreign interface, and the social cost of creating a new language that needs to be learned. These costs can be mitigated to some extent by making your new language similar to an existing language, but that pulls you towards the reimplementation problems inherent in the source-to-source transform approach. Farfallen is a research system of this shape that I worked on, whose language is similar to Fortran77.

FURTHER READING

http://autodiff.org is the automatic differentiation community portal. Among other things, it lists several textbooks, and has an enormous database of related academic publications.

The formative literature for my own understanding of automatic differentiation was the output of the collaboration between Professors Siskind and Pearlmutter and their research groups; to which body of work I have even had the honor of contributing. For the reader who may wish to repeat this educational path, an annotated bibliography:

Jeffrey Mark Siskind and Barak A. Pearlmutter, Putting the Automatic Back into AD: Part I, What's Wrong, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA, Jan 2008, TR-ECE-08-02.

A screed on the state of high-performance automatic differentiation tooling. By reversing the senses of all the assertions, this can be read as a manifesto for what automatic differentiation tools should offer.

Barak A. Pearlmutter and Jeffrey Mark Siskind, Putting the Automatic Back into AD: Part II, Dynamic, Automatic, Nestable, and Fast, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA, Jan 2008, TR-ECE-08-03.

An argument, to the scientific computing community, that higher-order functions are valuable in general, and that differentiation is a higher-order function; describes AD in those terms. Argues that therefore implementation technology developed for dealing with higher-order functions would serve the community well. To that point, presents some performance results for the Stalingrad compiler.

Published, in somewhat condensed form, as Barak A. Pearlmutter and Jeffrey Mark Siskind. Using programming language theory to make AD sound and efficient. In Proceedings of the 5th International Conference on Automatic Differentiation, pages 79-90, Bonn, Germany, Aug 2008, doi:10.1007/978-3-540-68942-3_8. Springer-Verlag.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Perturbation Confusion and Referential Transparency: Correct Functional Implementation of Forward-Mode AD. Draft Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages (IFL2005), Sep 19-21, 2005, Dublin Ireland. (Also ps, djvu)

A concise paper on the perturbation confusion problem, in the context of forward mode AD embedded into a functional language.

Jeffrey Mark Siskind and Barak A. Pearlmutter. Nesting forward-mode AD in a functional framework. Higher Order and Symbolic Computation 21(4):361-76, 2008, doi:10.1007/s10990-008-9037-1. (Also ps)

A catalog of devices for avoiding perturbation confusion. The de-

vices are exhibited within complete implementations of a first-class forward mode AD operator in a functional-programming setting. I direct your attention to an argument, in footnote 6 near the end of the paper, to the effect that some kind of referential non-transparency is required to implement a correctly nestable derivative-taking operator. This is ironic, because the resulting operator is itself referentially transparent.

A somewhat expanded version is also available as Siskind, J.M. and Pearlmutter, B.A., Nesting Forward-Mode AD in a Functional Framework, Technical Report TR-ECE-08-09, School of Electrical and Computer Engineering, Purdue University, 2008.

Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-Mode AD in a functional framework: Lambda the ultimate backpropagator. TOPLAS 30(2):1-36, Mar 2008, doi:10.1145/1330017.1330018. (Also ps)

The full story on how to implement a first-class reverse mode AD operator in a higher-order language by (conceptually) reflective runtime code transformation (including transforming code that itself uses AD). This document is quite dense, but it does capture both what reverse mode is and how it is actually implemented in Stalingrad.

Barak A. Pearlmutter and Jeffrey Mark Siskind. Lazy Multivariate Higher-Order Forward-Mode AD. In Proceedings of the 2007 Symposium on Principles of Programming Languages (POPL 2007), Nice France, pages 155-60, doi:10.1145/1190215.1190242, Jan 2007. (Also ps, djvu)

The title basically says it all. This is a paper about how to do the job at all, not how to do it super-efficiently.

Jeffrey Mark Siskind and Barak A. Pearlmutter, Using Polyvariant Union-Free Flow Analysis to Compile a Higher-Order Functional-Programming Language with a First-Class Derivative Operator to Efficient Fortran-like Code, School of Electrical and Computer Engineering, Purdue University, West Lafayette, IN, USA, Jan 2008, TR-ECE-08-01.

An incredibly dense technical report on a compilation strategy aggressive enough to make AD very efficient. This text was the working blueprint (in addition to inspiration from the Stalingrad source code) which helped me get the implementation of DVL off the ground. Specifically, my Vunctional Language is a reasonably expository implementation of the flow analysis contained in the report, and DVL builds on that to implement AD in a different way.

While not for the faint of heart, this publication is an essential piece of the reproducibility of Siskind and Pearlmutter's other results, especially as regards program performance.

Oleksandr Manzyuk, Barak A. Pearlmutter, Alexey Andreyevich Radul, David R. Rush, and Jeffrey Mark Siskind, Confusion of Tagged Perturbations in Forward Automatic Differentiation of Higher-Order Functions, 2012, arxiv:1211.4892.

A spectacular bug that your implementation of AD may still have even if you have zealously read everything else and followed all the advice. Don't read this unless you really want to implement AD, or unless you need further convincing that the task is subtle.

Oleksandr Manzyuk, A Simply Typed λ-Calculus of Forward Automatic Differentiation, Electronic Notes in Theoretical Computer Science, 2012, Elsevier.

Semantics for a first-class tangent bundle operator embedded in a λ-calculus, paving the way to fully rigorous foundations for AD.

Oleksandr Manzyuk, Tangent bundles in differential λ-categories, 2012, arXiv:1202.0411.

Very heavy category-theoretic computations providing a philosophical justification for the definition of forward mode AD on functions that return arbitrary program objects (notably including functions) as opposed to values that are easily interpretable as vectors in $\mathbb{R}^n$.

NOTES

1. Truncation error is the inaccuracy you get from h not actually being zero. Roundoff error is the inaccuracy you get from valuable low-order bits of the final answer having to compete for machine-word space with high-order bits of $f(x+h)$ and $f(x)$, which the computer has to store just until they cancel in the subtraction at the end.↩

2. Better numerical differentiation lesson one: evaluate $(f(x+h) - f(x-h))/2h$ instead of $(f(x+h) - f(x))/h$. Your truncation error moves from first-order to second-order in h.↩

3. Disclaimer: I have not actually personally seen any literature doing numerical error analysis of programs produced by automatic differentiation. The technique does avoid the gross problem of catastrophic cancellation when subtracting $f(x)$ from $f(x+dx)$, but may do poorly in more subtle situations. Consider, for example, summing a set of numbers of varying magnitudes. Generally, the answer is most accurate if one adds the small numbers together first, before adding the result to the big ones, rather than adding small numbers to large haphazardly. But what if the sizes of the perturbations in one's computation did not correlate with the sizes of the primals? Then AD of a program that does the right thing with the primals will do the wrong thing with the perturbations.

It would be better, in this case, to define the summation function as a primitive (for AD's purposes). The derivative is also summation, but with the freedom to add the perturbations to each other in a different order. Doing this scalably, however, is an outstanding issue with AD.↩

4. Factoring the Jacobian in full: for the decomposition of a function f into primitives

$$f = f_k \circ \ldots \circ f_i \circ \ldots \circ f_3 \circ f_2 \circ f_1,$$

define the partial-result function f^i as

$$f^i = f_i \circ \ldots \circ f_3 \circ f_2 \circ f_1.$$

Then

$$J_x f = J_{f^{k-1}(x)} f_k \cdot \ldots \cdot J_{f^{i-1}(x)} f_i \cdot \ldots \cdot J_{f^2(x)} f_3 \cdot J_{f^1(x)} f_2 \cdot J_x f_1.$$

↩

5. For functions with data-dependent branches, the decomposition of the Jacobian depends on the input x, but that doesn't actually cause

any trouble—all actual computations done by AD are local at x. If the input x happens to be at a boundary between regions of definition of f, the answers may be somewhat surprising, but the semantic difficulties of defining derivatives of f at such points are not unique to AD. Numerical methods, in fact, start suffering from discontinuities even at some distance, because of truncation error.↩

6. In theory the arithmetic slowdown imposed by automatic differentiation is no worse than 6x or 3x depending on whether you take $\tan(x)$ as primitive or as the three primitives $\sin(x)/\cos(x)$. In practice it's closer to +10% because the expensive operations are exponentials, but they are their own derivatives.↩

← Conversations index