An Intellectual History of Automatic Differentiation

Papers We Love

May 24th, 2017

What is Automatic Differentiation?

What it's not:

- Numeric approximation (Lagrangian interpolation, etc.)
    - *Relatively* fast, but inexact (e.g. Runge's phenomenon).
- Symbolic integration (computer algebra systems, e.g. MATLAB, Maple, Mathematica)
    - Exact, but slow and *ugly* ...who enjoyed high school calculus?

What is Automatic Differentiation?

But what *is* it? An example in Haskell:

```
λ> let sin' = int cos'
λ> let cos' = 1 - int sin'
λ> take 5 $ sin'
[0 % 1,1 % 1,0 % 1,(-1) % 6,0 % 1,
1 % 120,0 % 1,(-1) % 5040,0 % 1,1 % 362880]
λ> take 5 $ cos'
[1 % 1,0 % 1,(-1) % 2,0 % 1,1 % 24,
0 % 1,(-1) % 720,0 % 1,1 % 40320,0 % 1]
```

Exact *and* fast! Plus no ugly subscripts :)

Why does this even work?

> “ The Jargon File makes a distinction between deep magic, which refers to code based on esoteric theoretical knowledge, and black magic, which refers to code based on techniques that appear to work but which lack a theoretical explanation. ”
>
> \- Wikipedia

It's not a coincidence. AD is baked into the very structure of computation....

The Operational Calculus

You may have seen some of these:

$$\begin{aligned}
\operatorname{grad} f &= \nabla f &&= (df)^\sharp \\
\operatorname{div} F &= \nabla \cdot F &&= \star d \star (F^\flat) \\
\operatorname{curl} F &= \nabla \times F &&= \left(\star d (F^\flat)\right)^\sharp \\
\Delta f &= \nabla^2 f &&= \star d \star d f,
\end{aligned}$$

Since Leibniz, mathematicians were developing notations for calculus by overloading mathematical symbols much as we do with operators in languages like C++ and Haskell.

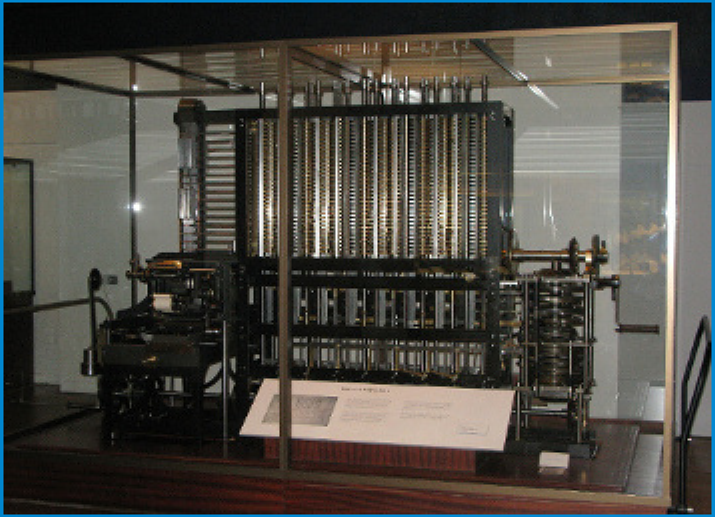
The Operational Calculus

During the late 19th century this became known as the **operational calculus** and was especially popular in mathematical physics:

- George Boole: *A Treatise on Differential Equations* (1859)
- Oliver Heaviside: *On Operators in Physical Mathematics* (1893)
- Norbert Wiener: *The Operational Calculus* (1926)

Computation

Differentiation was a motivating example for computation from the very beginning:



In 1822, 15 years before the general purpose Analytical Engine, Charles Babbage described the **Difference Engine** sparking interest in analogue computers for the purpose of calculating derivatives that would last well into the 20th century.

Computation

However, the fact that differentiation itself could not be formalized as a mathematical function continued to plague logicians.

Until...

The Lambda Calculus

> “ It is, of course, not excluded that the range of arguments or range of values of a function should consist wholly or partly of functions. The derivative, as this notion appears in the elementary differential calculus, is a familiar mathematical example of a function for which both ranges consist of functions. ”

- Alonzo Church
The Calculi of Lambda Conversion (1941)

LISP

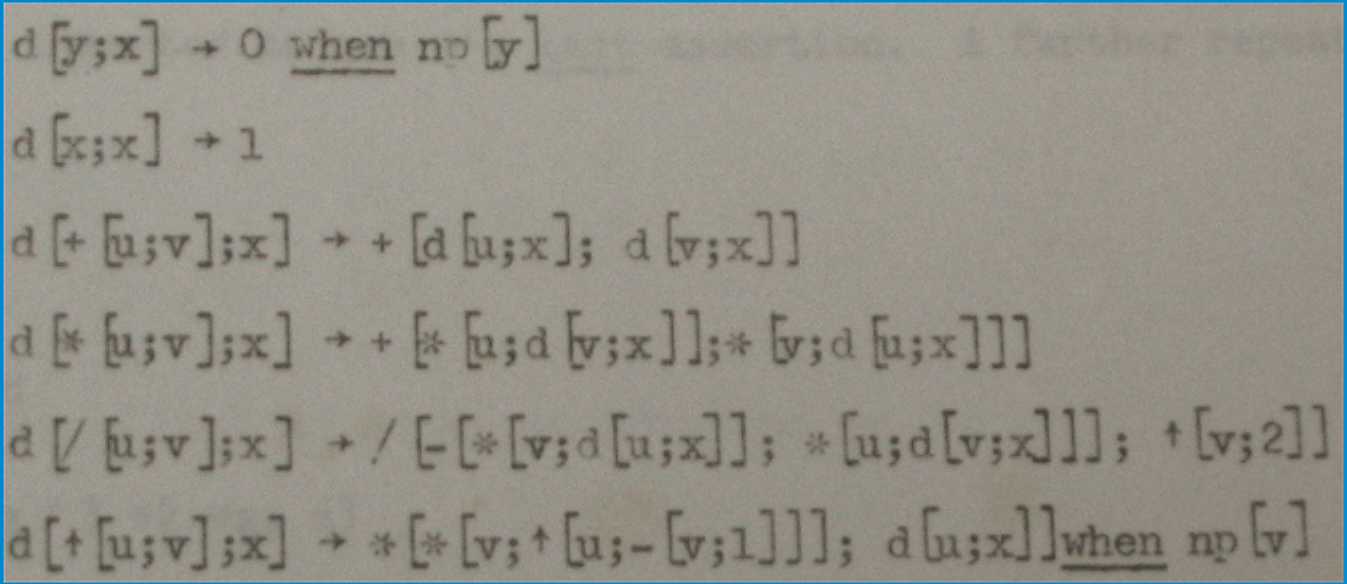
In 1958 John McCarthy based LISP on Church's untyped lambda caclulus.

When demonstrating it to an informal audience at MIT in 1959 he built up from basic list processing to higher order functions in the course of an hour.

The culmination of McCarthy's lecture? A simple program for univariate differentiation.

LISP

In 1970 Fred McBride (father of Conor McBride) added pattern matching to a dialect of Lisp for his dissertation, *Computer Aided Manipulation of Symbols*. Like McCarthy, he used it to demonstrate a short program for automatic differentiation:



```
d[y;x] → 0 when np[y]
d[x;x] → 1
d[+[u;v];x] → +[d[u;x]; d[v;x]]
d[*[u;v];x] → +[*[u;d[v;x]];*[v;d[u;x]]]
d[/[u;v];x] → /[-[*[v;d[u;x]]; *[u;d[v;x]]]; ↑[v;2]]
d[↑[u;v];x] → *[*[v;↑[u;-[v;1]]]; d[u;x]]when np[v]
```

The isomorphism of differentiation with (lazy) list processing was given by Dusko Pavlovic and Martín Escardó in *Calculus in Coinductive Form* (1998).

Among other examples, they give the commuting square for the infinite Taylor series we saw earlier:

$$\begin{tikzcd}
\mathbb{A} \arrow[rr, "\langle O, D \rangle"] \arrow[d, bend right, "\mathsf{T}"'] \arrow[d, phantom, "\cong"] & & \mathbb{R} \times \mathbb{A} \arrow[d, bend right, "\mathbb{R} \times \mathsf{T}"'] \arrow[d, phantom, "\cong"] \\
\mathbb{R}^{<\omega} \arrow[u, bend right, "\tilde{\mathsf{T}}"'] \arrow[rr, "\langle \mathbf{head}, \mathbf{tail} \rangle"'] & & \mathbb{R} \times \mathbb{R}^{<\omega} \arrow[u, bend right, "\mathbb{R} \times \tilde{\mathsf{T}}"']
\end{tikzcd}$$

Imperative AD

The first literature on AD was by Robert Edwin Wengert in 1964.

A Simple Automatic Derivative Evaluation Program (1964) is one of many claims to the first dissertation ever written in the field of computer science.

Imperative AD

The technique was popular in the numerical computing mainstream for some time:

- Many AD tools, particularly in Fortran and C++, are compiled by Argonne National Laboratory: http://www.autodiff.org/.
- However AD was largely abandoned in favor of "numerical methods," particularly with the advent of GPUs for fast matrix processing.

Then functional programming took over...

Lazy Evaluation

AD is particularly elegantly expressed using stream processing, a concept first formalized by Peter Landin in _Correspondence Between ALGOL 60 and Church's Lambda-notation_ (1965).

This started a whole field of research into non-strict, or lazy, evaluation methods. A seminal paper that implemented a lazy version of McCarthy's Lisp interpreter was Daniel Friedman and David Wise's *CONS Should Not Evaluate Its Arguments* (1975).

Lazy Evaluation

The Lisp community quickly abandoned lazy evaluation, but it later became popular in other functional languages: KRC, Miranda, and Haskell.

Philip Wadler, one of the original developers of Haskell, examined lazy lists in *The Essence Of Functional Programming* (1992):

> “ It is difficult to see how to make this change in an impure language. Perhaps one might create some form of coroutine facility. ”

With hindsight, this is not difficult to see at all...

Coroutines

What came to be the standard approach to functional AD first appeared in 1977 in an unpublished paper by Gilles Kahn & David MacQueen, *Coroutines and Networks of Parallel Processes*.

The paper focused on a a coroutine-based approach to generating prime numbers in ML using the Sieve of Eratosthenes. An AD package was only mentioned in the conclusion with no code provided.

SICP

Both the prime sieve and power series programs became canonical examples of the power of lazy evaluation, likely owing to their inclusion in Gerald Sussman and Harold Abelson's *Structure and Interpretation of Computer Programs*.

Sussman would later release a more general AD implementation as part of his SCMUTILS package used in *Structure and Interpretation of Classical Mechanics*, co-written with Jack Wisdom.

Unix

Kahn and MacQueen's paper also caught the eye of Doug McIlroy, then the head of the Computing Techniques Research Department at Bell Labs that birthed Unix and C.

McIlroy was present at John McCarthy's original AD demo and had himself programmed one of the earliest implementations of the prime sieve using coroutines in 1968.

Unix

McIlroy is best known for adding pipelines to Unix, which enabled the "the Unix philosophy" of composing many single-purpose programs through a common interface: text-streams.

Standard I/O is fundamentally lazy, it inputs and outputs only as much as the program needs.

Oleg Kiselyov even pointed out the similarity between Unix pipes and the IO monad.

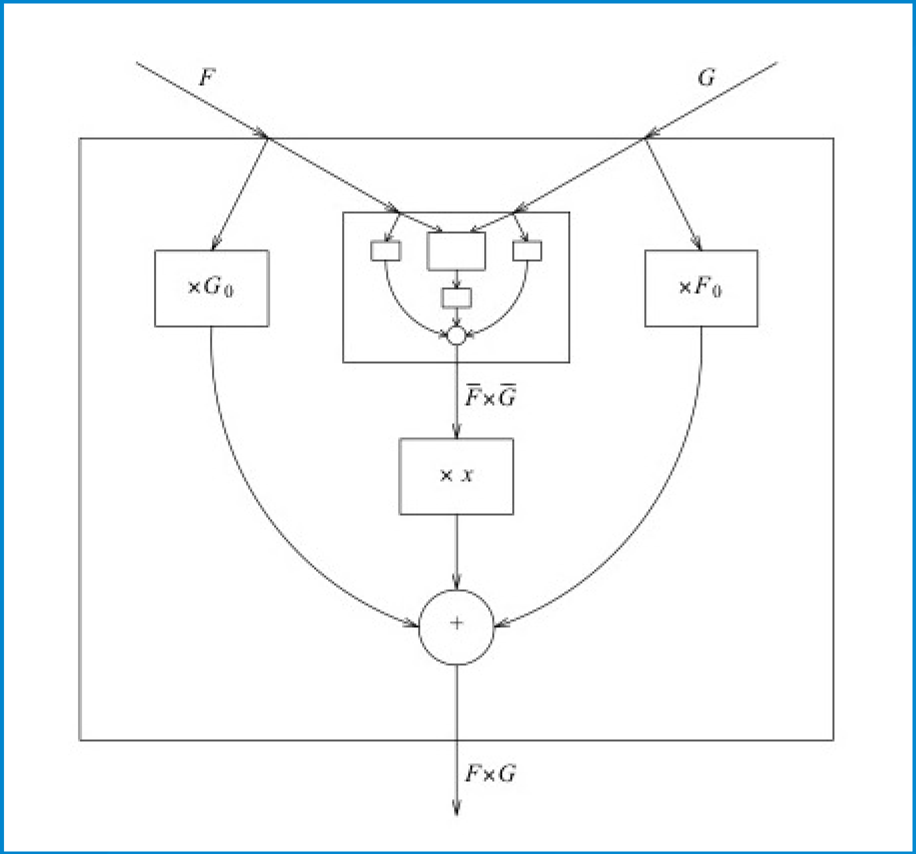
Concurrent AD

McIlroy would later describe his use of coroutines in terms of Tony Hoare's groundbreaking concurrency model *Communicating Sequential Processes* (1978).

In the 1980s Bell Lab's Rob Pike developed a series of languages based on Hoare's CSP model of concurrency, leading up to Google's Go language.

One such language, Newsqueak, provided the medium for McIlroy's first attempt at implementing Kahn and McQueen's coroutine-based AD program, which he published in the paper *Squinting at Power Series* (1989).

Concurrent AD



McIlroy's function for the Cauchy product using recursively generated channels

AD in Haskell

McIlroy later wrote a version of his power series program in Haskell, published in *Power Series, Power Serious* (1998) and *The Music of Streams* (2001).

The most basic version consisted of 17 one-liners:

```haskell
infixr 9 #
series f = f : repeat 0
instance (Num a, Eq a) => Num [a] where
   fromInteger c = series(fromInteger c)
   negate (f:ft) = -f : -ft
   (f:ft) + (g:gt) = f+g : ft+gt
   (f:ft) * gs@(g:gt) = f*g : ft*gs + series(f)*gt
instance (Fractional a, Eq a) => Fractional [a] where
   (f:ft) / (g:gt) = qs
     where qs = f/g : series(1/g)*(ft-qs*gt)
(f:ft) # gs@(0:gt) = f : gt*(ft#gs)
revert (0:ft) = rs where rs = 0 : 1/(ft#rs)
int fs = 0 : zipWith (/) fs [1..]
diff (_:ft) = zipWith (*) ft [1..]
tans = revert(int(1/(1:0:1)))
sins = int coss
coss = 1 - int sins
```

He described it as, "The most beautiful code I've ever written."

"Worse is Better" Was a Lie

...and thus automatic differentiation is the missing bridge between Unix & C and Lisp & Functional Programming.

Functional AD Taken Seriously

One year prior to McIlroy's "Power Serious," a researcher named Jerzy Karczmarczuk published another Haskell version using a different approach:

- Focus on finite polynomials (coining the phrase "lazy tower" for the derivatives)
- Dual numbers (tuples of doubles) used to represent the value of a function and its derivative at a given point
- Generating new Haskell functions to calculate derivatives, allowing use of built-in functional composition

Jerzy Karczmarczuk

Karczmarczuk's *Generating Power of Lazy Semantics* (1997) became a seminal paper in the field and he went on to write numerous others:

- *Functional Coding of Differential Forms* (1999)
- *Functional Differentiation of Computer Programs* (2000)
- *Adjoint Codes in Functional Framework* (2000)
- *Lazy Time Reversal, and Automatic Differentiation* (2002)

AD Modes

Forward, reverse (adjoint), or mixed mode?

- Forward
    - Application of the chain rule from left to right
    - Or inside to outside when thought of in terms of functional composition
    - i.e. the way you learned in high school calculus
    - Generally considering the most straightforward to implement

AD Modes

- Reverse mode:
    - Application of the chain rules from right to left
    - Or outside to inside in terms of functional composition
    - For this last reason, much less intuitive and more difficult to implement
    - However, extremely useful for certain applications (machine learning...)

AD Modes

- Mixed mode:
    - What is sounds like: a combination of both directions

AD Techniques: Data-Driven

- Either returning the value of a derivative...
- ...or the derivative itself represented as a value (as in McIlroy's Haskell version)
- Generally considered the most primitive method and only useful for power series...
- ...however, this assumes the inability to compose functions once they're output as data.
- McIlroy showed this actually *can* be done by converting functions to Horner form

AD Techniques: Functions

Using operator overloading:

- Karczmarczuk's method, also imperative implementations (i.e. FADBAD++)
- Also Conal Elliot: *Beautiful Differentiation* (2009)
- Upside vs. data-driven approach: allows use of built-in functional composition

AD Techniques: Functions

Downsides of operator overloading approach:

- Introduces problem of confusing levels of derivatives, i.e. overloaded operators cannot be applied to derivatives at multiple levels
- Referred to as "perturbation confusion" of "confusion of infinitesimals"
- Makes reverse mode very difficult
- Current dominant Haskell package, Edward Kmett's AD library, started as a Stack Overflow answer about reverse mode in Haskell

AD Techniques: Source Generation

Derivative functions are generated using compile-time metaprogramming:

- Solves the problems presented by operator overloading
- Used in several extremely fast Fortran packages
- DiffSharp:
    - Source transformation using the F# quotations evaluator
    - Benefits from incremental compilation using .NET's LINQ framework

Siskind and Pearlmutter

Jeffrey Siskind and Barak Pearlmutter:

- By far the most prolific AD researchers.
- Mainly working in Scheme and Haskell, but also DiffSharp and a Lisp dialect AD as primitives.
- First to point out problems with the operator overloading approach in the classic paper *Perturbation Confusion and Referential Transparency: Correct Functional Implementation of Forward-Mode AD* (2005)

Siskind and Pearlmutter

Went on to publish numerous others including:

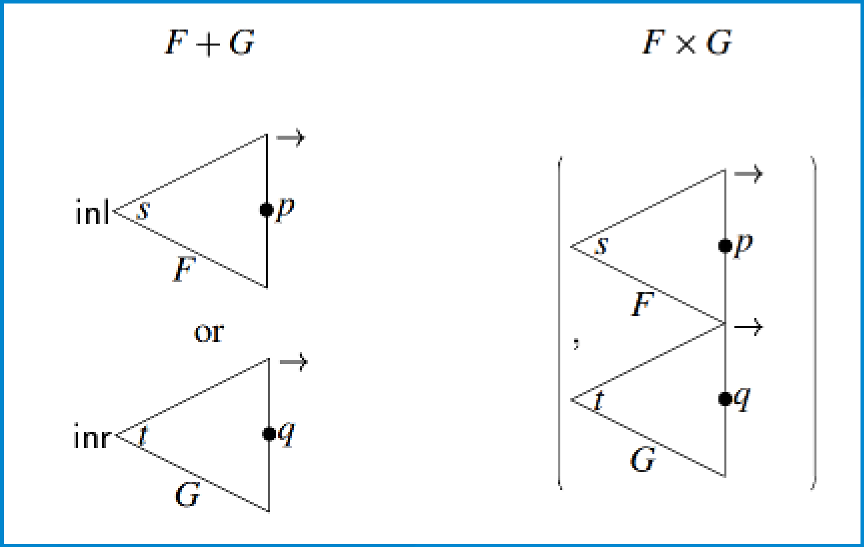
- *Lazy Multivariate Higher-Order Forward-Mode AD* (2007)
- *Nesting Forward-Mode AD in a Functional Framework* (2007)
- *Reverse-Mode AD in a Functional Framework: Lambda the Ultimate Backpropagator* (2008)
- *Putting the Automatic Back into AD* (2008)

Derivatives of Types

Seminal paper is Conor McBride's *The Derivative of a Regular Type is its Type of One-Hole Contexts* (2001). Already presented at Papers We Love.

> “ The derivative is thus the sum of terms corresponding to each one-hole context for a zipper in the expression. Perhaps the key to the connection can be found by focusing not on what is being infinitesimally varied, but on what, for the sake of a linear approximation to the curve, is being kept the same. ”

Derivatives of Types



Other papers on type-level derivatives:

- *∂ for Data: Differentiating Data Structures* (2005) Conor McBride, Thorsten Altenkirch, et al
- *The Two Dualities of Computation: Negative and Fractional Types* (2012) James & Sabry

Derivatives of Types

- Requires dependent types, i.e. a specification of the relationship between the parametric types of the containers and the data they hold.
- Interestingly, the concept of universes in type theory is isomorphic to that of the functional approach to differentiation in that operators have different meanings on different levels.
- Differential geometry is also being formalized in category theory as R-modules, which turn out to correspond to types in the simply typed version of the differential lambda calculus...

Differential Lambda Calculus

Thomas Ehrhard and Laurent Regnier in the *The Differential Lambda-Calculus* (2001)

Builds on McBride's work, but refining the notion of "regular types" to variables in the lambda calculus using linear logic:

- Extends the Taylor formula (computing derivatives by composing Taylor series) to bound variables in lambda terms
- One can also think of the arguments to curried functions in typed lambda calculi as having a correspondence with terms in Taylor series.

- Partial derivatives are represented as substitutions over different bound variables.
- A *purely* differential lambda calculus, i.e. one with only bound variables, means that all derivatives except for that of zero are partial.
- The chain rule is applied in a manner similar to encoding of Church numerals by repeated application of the successor function.
- Reduction rules for lambda calculus hold for differentiation: partials are function bodies that relate to only one argument in a multi-ariadic function.
- The chain rule is literally just beta-reduction.

Differential Lambda Calculus

In other words...the differential lambda calculus is Church's dream realized!

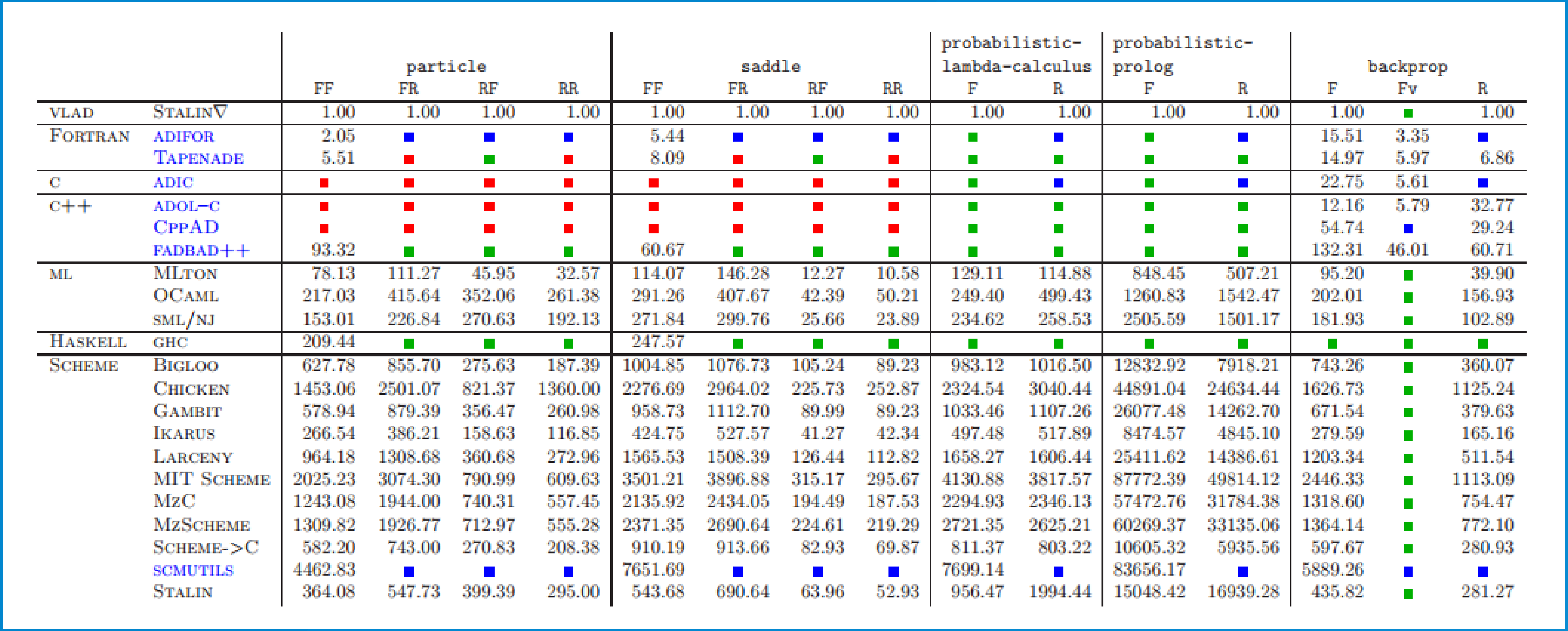
VLAD: a purely functional language with built-in AD

Differential lambda calculus was implemented by Siskind & Pearlmutter in their Stalingrad interpreter for the Lisp dialect VLAD:

- Allows differentiation to commute as it does using symbolic methods (Schwarz lemma)
- Eliminates reflection using SSA, resulting is a 3-5x speedup vs. Fortran-based source transformation, 50x vs. C++ template-based approaches, and 250x (!) compared to the best Haskell libaries

Benchmarks

Using the examples of minimax of a saddle curve and a particle simulation using Euler's equations:



		particle				saddle				probabilistic-lambda-calculus		probabilistic-prolog		backprop		
		FF	FR	RF	RR	FF	FR	RF	RR	F	R	F	R	F	Fv	R
VLAD	STALIN∇	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	🟩	1.00
FORTRAN	ADIFOR	2.05	🟦	🟦	🟦	5.44	🟦	🟦	🟦	🟩	🟦	🟩	🟦	15.51	3.35	🟦
	TAPENADE	5.51	🟥	🟩	🟥	8.09	🟥	🟩	🟥	🟩	🟩	🟩	🟩	14.97	5.97	6.86
C	ADIC	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟦	🟩	🟦	22.75	5.61	🟦
C++	ADOL–C	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟩	🟩	12.16	5.79	32.77
	CPPAD	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟩	🟩	🟩	🟩	54.74	🟦	29.24
	FADBAD++	93.32	🟩	🟩	🟩	60.67	🟩	🟩	🟩	🟩	🟩	🟩	🟩	132.31	46.01	60.71
ML	MLTON	78.13	111.27	45.95	32.57	114.07	146.28	12.27	10.58	129.11	114.88	848.45	507.21	95.20	🟩	39.90
	OCAML	217.03	415.64	352.06	261.38	291.26	407.67	42.39	50.21	249.40	499.43	1260.83	1542.47	202.01	🟩	156.93
	SML/NJ	153.01	226.84	270.63	192.13	271.84	299.76	25.66	23.89	234.62	258.53	2505.59	1501.17	181.93	🟩	102.89
HASKELL	GHC	209.44	🟩	🟩	🟩	247.57	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩
SCHEME	BIGLOO	627.78	855.70	275.63	187.39	1004.85	1076.73	105.24	89.23	983.12	1016.50	12832.92	7918.21	743.26	🟩	360.07
	CHICKEN	1453.06	2501.07	821.37	1360.00	2276.69	2964.02	225.73	252.87	2324.54	3040.44	44891.04	24634.44	1626.73	🟩	1125.24
	GAMBIT	578.94	879.39	356.47	260.98	958.73	1112.70	89.99	89.23	1033.46	1107.26	26077.48	14262.70	671.54	🟩	379.63
	IKARUS	266.54	386.21	158.63	116.85	424.75	527.57	41.27	42.34	497.48	517.89	8474.57	4845.10	279.59	🟩	165.16
	LARCENY	964.18	1308.68	360.68	272.96	1565.53	1508.39	126.44	112.82	1658.27	1606.44	25411.62	14386.61	1203.34	🟩	511.54
	MIT SCHEME	2025.23	3074.30	790.99	609.63	3501.21	3896.88	315.17	295.67	4130.88	3817.57	87772.39	49814.12	2446.33	🟩	1113.09
	MZC	1243.08	1944.00	740.31	557.45	2135.92	2434.05	194.49	187.53	2294.93	2346.13	57472.76	31784.38	1318.60	🟩	754.47
	MZSCHEME	1309.82	1926.77	712.97	555.28	2371.35	2690.64	224.61	219.29	2721.35	2625.21	60269.37	33135.06	1364.14	🟩	772.10
	SCHEME->C	582.20	743.00	270.83	208.38	910.19	913.66	82.93	69.87	811.37	803.22	10605.32	5935.56	597.67	🟩	280.93
	SCMUTILS	4462.83	🟦	🟦	🟦	7651.69	🟦	🟦	🟦	7699.14	🟦	83656.17	🟦	5889.26	🟦	🟦
	STALIN	364.08	547.73	399.39	295.00	543.68	690.64	63.96	52.93	956.47	1994.44	15048.42	16939.28	435.82	🟩	281.27

Efficient Implementation of a Higher-Order Language with Built-In AD (2016)

The AD Renaissance: Machine Learning

Good news: AD is becoming popular again for practical use!

Why? Primarily for machine learning: backpropagation == the chain rule.

- Autograd for NumPy has been integrated with Torch
- Google's Ceres Solver (C++ numerical programming tool for ML) includes an AD option

The AD Renaissance: Machine Learning

More cutting edge:

- In *Learning to Transduce with Unbounded Memory* (2015) DeepMind demonstrated that training an LSTM using "differentiable data structures" (stacks, queues, and dequeues of matrices with AD built into them) allowed them to achieve the same results in one pass as in four using gradient descent through approximation.
- They've since moved on to designing "differentiable neural computers."

The AD Renaissance: Modelling

There's also interest in quantitative finance and other fields that require modelling stochastic processes:

- *Smoking Adjoints: Fast Monte Carlo Greeks* (2004) Giles & Glasserman
- *Adjoints and Automatic (Algorithmic) Differentiation in Computational Finance* (2011) Cristian Homescu
- The Stan probabilistic programming language developed at Columbia University includes an AD implementation in C++