Karl Gustav Jacob Jacobi

Friedrich Julius Richelot

Carl Gottfried Neumann

William Edward Story

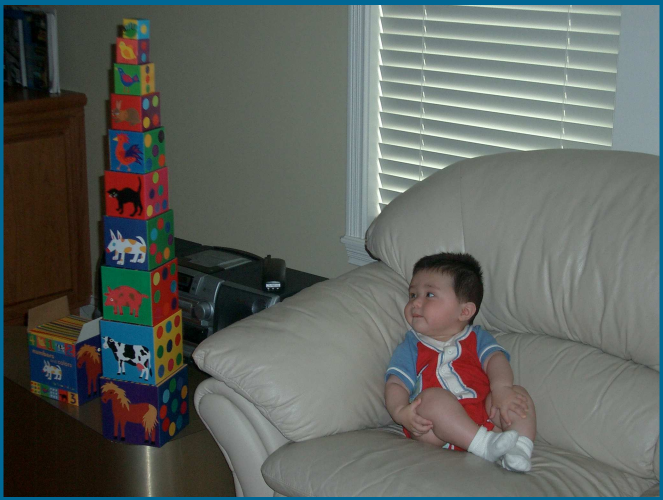
Solomon Lefschetz

Albert William Tucker

Marvin Lee Minsky

Patrick Henry Winston	Scott Fahlman
Robert C. Berwick	David Touretzky
Jeffrey Mark Siskind	Barak Avrum Pearlmutter

Jonathan Eden Siskind



Algorithmic Differentiation of Functional Programs

Jeffrey Mark Siskind

School of Electrical and Computer Engineering
Purdue University
Electrical Engineering Building, Room 313E
465 Northwestern Avenue
West Lafayette IN 47907–2035 USA
voice: 765/496–3197
fax: 765/494–6440
`qobi@purdue.edu`
`http://www.ece.purdue.edu/~qobi`

June 15, 2004

Joint work with Barak Pearlmutter.

Lambda: the Ultimate Calculus

Jeffrey Mark Siskind

School of Electrical and Computer Engineering
Purdue University
Electrical Engineering Building, Room 313E
465 Northwestern Avenue
West Lafayette IN 47907–2035 USA
voice: 765/496–3197
fax: 765/494–6440
`qobi@purdue.edu`
`http://www.ece.purdue.edu/~qobi`

June 15, 2004

Joint work with Barak Pearlmutter.

Backpropagation through Functional Programs

Jeffrey Mark Siskind

School of Electrical and Computer Engineering
Purdue University
Electrical Engineering Building, Room 313E
465 Northwestern Avenue
West Lafayette IN 47907–2035 USA
voice: 765/496–3197
fax: 765/494–6440
`qobi@purdue.edu`
`http://www.ece.purdue.edu/~qobi`

June 15, 2004

Joint work with Barak Pearlmutter.

Lambda: the Ultimate Neural Network

Jeffrey Mark Siskind

School of Electrical and Computer Engineering
Purdue University
Electrical Engineering Building, Room 313E
465 Northwestern Avenue
West Lafayette IN 47907–2035 USA
voice: 765/496–3197
fax: 765/494–6440
`qobi@purdue.edu`
`http://www.ece.purdue.edu/~qobi`

June 15, 2004

Joint work with Barak Pearlmutter.

Symbolicism: the Ultimate Connectionism

Jeffrey Mark Siskind

School of Electrical and Computer Engineering
Purdue University
Electrical Engineering Building, Room 313E
465 Northwestern Avenue
West Lafayette IN 47907–2035 USA
voice: 765/496–3197
fax: 765/494–6440
qobi@purdue.edu
http://www.ece.purdue.edu/~qobi

June 15, 2004

Joint work with Barak Pearlmutter.

Maybe the Brain Really Does Run Lisp After All

Jeffrey Mark Siskind

School of Electrical and Computer Engineering
Purdue University
Electrical Engineering Building, Room 313E
465 Northwestern Avenue
West Lafayette IN 47907–2035 USA
voice: 765/496–3197
fax: 765/494–6440
`qobi@purdue.edu`
`http://www.ece.purdue.edu/~qobi`

June 15, 2004

Joint work with Barak Pearlmutter.

Leibnitz (1664) + Church (1941) = Siskind & Pearlmutter (2004)

Differential Calculus for Dummies

(in 6 slides)

Notation

- x, y, $\mathbf{x}$, f, g, h, p, x', x_1, $[\,]$
- comma, left associates
- juxtaposition, left associates
  - function application
  - function composition
  - matrix-vector multiplication
  - matrix-matrix multiplication
  - scalar-scalar multiplication
  - $\prod$

Derivatives

$$\frac{d}{dx} \quad : \quad \underbrace{f}_{\mathbb{R}\to\mathbb{R}} \to \underbrace{f'}_{\mathbb{R}\to\mathbb{R}}$$

$$\frac{d}{dx} \quad : \quad (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

$$\mathcal{D} \quad : \quad (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

Partial Derivatives

$$\frac{\partial}{\partial x} \quad : \quad \underbrace{f}_{\mathbb{R}^n \to \mathbb{R}} \to \underbrace{f'}_{\mathbb{R}^n \to \mathbb{R}}$$

$$\frac{\partial}{\partial x} \quad : \quad (\mathbb{R}^n \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R})$$

$$\mathcal{D}_i \quad : \quad (\mathbb{R}^n \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R})$$

Gradients

$$
\begin{aligned}
\nabla\ f\ \mathbf{x} &= (\mathcal{D}_1\ f\ \mathbf{x}), \ldots, (\mathcal{D}_n\ f\ \mathbf{x}) \\
\boldsymbol{\nabla} &: (\mathbb{R}^n \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R}^n)
\end{aligned}
$$

Jacobians

$$\begin{aligned} f &: \mathbb{R}^m \to \mathbb{R}^n \\ \mathbf{f} &: (\mathbb{R}^m \to \mathbb{R})^n \\ (\mathcal{J}\ f\ \mathbf{x})[i,j] &= (\nabla\ (\mathbf{f}[i]))[j] \\ \mathcal{J} &: (\mathbb{R}^m \to \mathbb{R}^n) \to (\mathbb{R}^m \to \mathbb{R}^{m\times n}) \end{aligned}$$

Operators

$\mathcal{D}$, ∇, and $\mathcal{J}$ are traditionally called *operators*.
A more modern term is *higher-order functions*.
Higher-order functions are common in mathematics, physics, and engineering:

> summations, comprehensions, quantifications, optimizations, integrals, convolutions, filters, edge detectors, Fourier transforms, differential equations, Hamiltonians, . . .

The Chain Rule

$$(f \circ g)\ x = (g\ f)\ x = g\ (f\ x)$$

$$\mathcal{D}\ (g\ f)\ x = (\mathcal{D}\ g\ f\ x)\ (\mathcal{D}\ f\ x)$$

$$\mathcal{J}\ (g\ f)\ \mathbf{x} = (\mathcal{J}\ g\ f\ \mathbf{x})\ (\mathcal{J}\ f\ \mathbf{x})$$

Everything You Always Wanted to Know About the Lambda Calculus*

(in 7 slides)

*But Were Afraid To Ask

Church (1941)

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.

Gottfried Leibniz
|
Jacob Bernoulli
|
Johann Bernoulli
|
Leonhard Euler
|
Joseph Louis Lagrange
|
Simeon Poisson
|
Michel Chasles
|
Hubert Anson Newton
|
Eliakim Hastings Moore
|
Oswald Veblen
|
Alonzo Church

Functional Programming

```
int f(int n)
{ int i, p = 1;
  for (i = 1; i<n; i++)
  { p = p*i;}
  return p;}
```

$$f\ n \stackrel{\triangle}{=} \begin{array}{l} \textbf{if}\ n = 0 \\ \textbf{then}\ 1 \\ \textbf{else}\ n \times (f\ (n-1))\ \textbf{fi} \end{array}$$

Higher-Order Functions

$$\sum_{i=1}^{n} \exp(i)$$

$$\prod_{i=1}^{n} \sin(i)$$

$$\begin{aligned} \textsc{fold}\ (i, a, f, g) \stackrel{\triangle}{=}\ & \textbf{if}\ i = 0 \\ & \textbf{then}\ a \\ & \textbf{else}\ \textsc{fold}\ ((i-1), (g\ (a, (f\ i))), f, g)\ \textbf{fi} \end{aligned}$$

$\textsc{fold}\ (n, 0, \exp, +)$

$\textsc{fold}\ (n, 1, \sin, \times)$

$$\sum_{i=1}^{n} 2i + 1$$

$f\ i \stackrel{\triangle}{=} 2i + 1$

$\textsc{fold}\ (n, 0, f, +)$

$\textsc{fold}\ (n, 0, (\lambda i\ 2i + 1), +)$

Closures

$(\lambda x\ 2x)\ 3 = 6$

$((\lambda x\ \lambda y\ x + y)\ 3)\ 4 = 7$

$(\lambda x\ \lambda y\ x + y)\ 3 = ?$

$(\lambda x\ \lambda y\ x + y)\ 3 = \langle\{x \mapsto 3\}, \lambda y\ x + y\rangle$

$\lambda x\ \lambda y\ x + y$

$\lambda(x, y)\ x + y$

Tail Recursion (Steele 1976)

$$f\ n \triangleq \begin{array}{l}\textbf{if}\ n = 0\\ \textbf{then}\ 1\\ \textbf{else}\ n \times (f\ (n-1))\ \textbf{fi}\end{array}$$

$$\begin{array}{rcl} g\ (i,p) & \triangleq & \begin{array}{l}\textbf{if}\ i = 0\\ \textbf{then}\ p\\ \textbf{else}\ g\ ((i-1),(p \times i))\ \textbf{fi}\end{array}\\ f\ n & \triangleq & g(n,1)\end{array}$$

Marvin Lee Minsky
|
Gerald Jay Sussman
|
Guy Lewis Steele, Jr.

Continuations (Landin 1965, Reynolds 1972)

$$
\begin{array}{lcl@{\qquad}lcl}
f\ x & \triangleq & e_1 & f'\ (c, x) & \triangleq & c\ e_1 \\
g\ x & \triangleq & e_2 & g'\ (c, x) & \triangleq & c\ e_2 \\
h\ x & \triangleq & e_3 & h'\ (c, x) & \triangleq & c\ e_3 \\
p\ x & \triangleq & h\ (g\ (f\ x)) & p'\ (c, x) & \triangleq & f\ ((\lambda x_1\ g\ ((\lambda x_2\ h\ (c, x_2)), x_1)), x)
\end{array}
$$

The Lambda Calculus

$$\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 \textbf{ fi} \rightsquigarrow (((\textsc{if}\ e_1)\ (\lambda x\ e_2))\ (\lambda x\ e_3))\ [\,]$$

$$e ::= x \mid e_1\ e_2 \mid \lambda x\ e$$

Compositional Derivative Operators—I

$$f_n \cdots f_1$$

$$\mathcal{J}\ (f_n \cdots f_1)$$

$$\mathcal{J}\ (f_n \cdots f_1) = \lambda \mathbf{x}\ \prod_{i=n}^{1} \left(\left(\mathcal{J}\ f_i\ \left(\prod_{j=i-1}^{1} f_j \right) \right)\ \mathbf{x} \right)$$

$\mathcal{J}\ (f_n \cdots f_1)$ is not compositional in $(\mathcal{J}\ f_1), \ldots, (\mathcal{J}\ f_n)$.

Compositional Derivative Operators—II

$$
\begin{aligned}
\overrightarrow{\nabla} f &\stackrel{\triangle}{=} \lambda(\mathbf{x}, \acute{\mathbf{x}})\ \mathcal{J}\ f\ \mathbf{x}\ \acute{\mathbf{x}} \\
\overleftarrow{\nabla} f &\stackrel{\triangle}{=} \lambda(\mathbf{x}, \grave{\mathbf{y}})\ (\mathcal{J}\ f\ \mathbf{x})^T\ \grave{\mathbf{y}}
\end{aligned}
$$

- $\mathbf{x}$ is a *primal* variable
- $\acute{\mathbf{x}}$ is a *forward adjoint* variable
- $\grave{\mathbf{x}}$ is a *reverse adjoint* variable

The rows and columns of $\mathcal{J}\ f\ \mathbf{x}$ can be computed as $\overleftarrow{\nabla}\ f\ (\mathbf{x}, \mathbf{e})$ and $\overrightarrow{\nabla}\ f\ (\mathbf{x}, \mathbf{e})$ for basis vectors $\mathbf{e}$ respectively.

Compositional Derivative Operators—III

$$
\begin{aligned}
\overrightarrow{\nabla}\,(g\ f) &= \lambda(\mathbf{x},\acute{\mathbf{x}})\ \overrightarrow{\nabla}\ g\ ((f\ \mathbf{x}),(\overrightarrow{\nabla}\ f\ (\mathbf{x},\acute{\mathbf{x}})))\\
\overleftarrow{\nabla}\,(g\ f) &= \lambda(\mathbf{x},\grave{\mathbf{y}})\ \overleftarrow{\nabla}\ f\ (\mathbf{x},(\overleftarrow{\nabla}\ g\ ((f\ \mathbf{x}),\grave{\mathbf{y}})))
\end{aligned}
$$

One cannot compose $\overrightarrow{\nabla}\ f$ with $\overrightarrow{\nabla}\ g$ because the input and output of $\overrightarrow{\nabla}\ f$ are not of the same type. Similarly for $\overleftarrow{\nabla}\ f$.

Compositional Derivative Operators—IV

$$\begin{aligned}\overrightarrow{\mathcal{J}}\, f &\stackrel{\triangle}{=} \lambda(\mathbf{x},\acute{\mathbf{x}})\ (f\ \mathbf{x}),(\overrightarrow{\nabla}\ f\ (\mathbf{x},\acute{\mathbf{x}}))\\ \overleftarrow{\mathcal{J}}\, f &\stackrel{\triangle}{=} \lambda(\mathbf{x},\tilde{\mathbf{x}})\ (f\ \mathbf{x}),(\tilde{\mathbf{x}}\ \lambda\grave{\mathbf{y}}\ \overleftarrow{\nabla}\ f\ (\mathbf{x},\grave{\mathbf{y}}))\end{aligned}$$

- $\lambda\grave{\mathbf{y}}\ \overleftarrow{\nabla}\ f\ (\mathbf{x},\grave{\mathbf{y}})$ is a *local backpropagator*
- $\tilde{\mathbf{x}}$ is an *input backpropagator*
- their composition as an *output backpropagator*

Compositional Derivative Operators—V

$$
\begin{aligned}
\textsc{Adjoint}\ (\mathbf{x}, \acute{\mathbf{x}}) &= \acute{\mathbf{x}}\\
\textsc{Backpropagator}\ (\mathbf{x}, \tilde{\mathbf{x}}) &= \tilde{\mathbf{x}}\\
\overrightarrow{\nabla}\ f &= \lambda(\mathbf{x}, \acute{\mathbf{x}})\ \textsc{Adjoint}\ (\overrightarrow{\mathcal{J}}\ (\mathbf{x}, \acute{\mathbf{x}}))\\
\overleftarrow{\nabla}\ f &= \lambda(\mathbf{x}, \grave{\mathbf{y}})\ \textsc{Backpropagator}\ (\overleftarrow{\mathcal{J}}\ (\mathbf{x}, I))\ \grave{\mathbf{y}}\\
\overrightarrow{\mathcal{J}}\ (g\ f) &= (\overrightarrow{\mathcal{J}}\ g)\ (\overrightarrow{\mathcal{J}}\ f)\\
\overleftarrow{\mathcal{J}}\ (g\ f) &= (\overleftarrow{\mathcal{J}}\ g)\ (\overleftarrow{\mathcal{J}}\ f)\\
\overrightarrow{\mathcal{J}}\ (f_n\ \cdots\ f_1) &= (\overrightarrow{\mathcal{J}}\ f_n)\ \cdots\ (\overrightarrow{\mathcal{J}}\ f_1)\\
\overleftarrow{\mathcal{J}}\ (f_n\ \cdots\ f_1) &= (\overleftarrow{\mathcal{J}}\ f_n)\ \cdots\ (\overleftarrow{\mathcal{J}}\ f_1)
\end{aligned}
$$

Traditional Forward-Mode AD—I

$$
\begin{array}{rclcrcl}
\mathbf{x}_1 & = & f_1\ \mathbf{x}_0 & \quad & \mathbf{x}_1, \acute{\mathbf{x}}_1 & = & \overrightarrow{\mathcal{J}}\ f_1\ (\mathbf{x}_0, \acute{\mathbf{x}}_0) \\
\mathbf{x}_2 & = & f_2\ \mathbf{x}_1 & & \mathbf{x}_2, \acute{\mathbf{x}}_2 & = & \overrightarrow{\mathcal{J}}\ f_2\ (\mathbf{x}_1, \acute{\mathbf{x}}_1) \\
& \vdots & & & & \vdots & \\
\mathbf{x}_n & = & f_n\ \mathbf{x}_{n-1} & & \mathbf{x}_n, \acute{\mathbf{x}}_n & = & \overrightarrow{\mathcal{J}}\ f_n\ (\mathbf{x}_{n-1}, \acute{\mathbf{x}}_{n-1})
\end{array}
$$

Traditional Forward-Mode AD—II

$$\begin{array}{rclrcl} x_j & = & f\ x_i & x_j, \acute{x}_j & = & (f\ x_i), ((\mathcal{D}\ f\ x_i)\acute{x}_i) \\ x_k & = & f\ (x_i, x_j) & x_k, \acute{x}_k & = & (f\ (x_i, x_j)), ((\mathcal{D}_1\ f\ (x_i, x_j)\ \acute{x}_i) + \\ & & & & & \quad (\mathcal{D}_2\ f\ (x_i, x_j)\ \acute{x}_j)) \end{array}$$

Traditional Reverse-Mode AD—I

$$
\begin{aligned}
\mathbf{x}_1 &= f_1\ \mathbf{x}_0 \\
\mathbf{x}_2 &= f_2\ \mathbf{x}_1 \\
&\ \vdots \\
\mathbf{x}_n &= f_n\ \mathbf{x}_{n-1}
\end{aligned}
\qquad\qquad
\begin{aligned}
\mathbf{x}_1, \tilde{\mathbf{x}}_1 &= \overleftarrow{\mathcal{J}}\ f_1\ (\mathbf{x}_0, \tilde{\mathbf{x}}_0) \\
\mathbf{x}_2, \tilde{\mathbf{x}}_2 &= \overleftarrow{\mathcal{J}}\ f_2\ (\mathbf{x}_1, \tilde{\mathbf{x}}_1) \\
&\ \vdots \\
\mathbf{x}_n, \tilde{\mathbf{x}}_n &= \overleftarrow{\mathcal{J}}\ f_n\ (\mathbf{x}_{n-1}, \tilde{\mathbf{x}}_{n-1}) \\
(\tilde{\mathbf{x}}_n\ I)\ \grave{\mathbf{x}}_n &
\end{aligned}
$$

Traditional Reverse-Mode AD—II

$$\begin{array}{rcl} \mathbf{x}_1 & = & f_1\ \mathbf{x}_0 \\ \mathbf{x}_2 & = & f_2\ \mathbf{x}_1 \\ & \vdots & \\ \mathbf{x}_n & = & f_n\ \mathbf{x}_{n-1} \end{array} \qquad \begin{array}{rcl} \mathbf{x}_1, \tilde{\mathbf{x}}_1 & = & \overleftarrow{\mathcal{J}}\ f_1\ (\mathbf{x}_0, \tilde{\mathbf{x}}_0) \\ \mathbf{x}_2, \tilde{\mathbf{x}}_2 & = & \overleftarrow{\mathcal{J}}\ f_2\ (\mathbf{x}_1, \tilde{\mathbf{x}}_1) \\ & \vdots & \\ \mathbf{x}_n, \tilde{\mathbf{x}}_n & = & \overleftarrow{\mathcal{J}}\ f_n\ (\mathbf{x}_{n-1}, \tilde{\mathbf{x}}_{n-1}) \\ \grave{\mathbf{x}}_{n-1} & = & \overleftarrow{\nabla}\ f_n\ (\mathbf{x}_n, \grave{\mathbf{x}}_n) \\ \grave{\mathbf{x}}_{n-2} & = & \overleftarrow{\nabla}\ f_{n-1}\ (\mathbf{x}_{n-1}, \grave{\mathbf{x}}_{n-1}) \\ & \vdots & \\ \grave{\mathbf{x}}_0 & = & \overleftarrow{\nabla}\ f_1\ (\mathbf{x}_1, \grave{\mathbf{x}}_1) \end{array}$$

Traditional Reverse-Mode AD—III

$$
\begin{array}{rclcrcl}
x_j & = & f\ x_i & \quad & x_j & = & f\ x_i \\
 & & & & \grave{x}_i & = & \grave{x}_i + (\mathcal{D}\ f\ x_i\ \grave{x}_j) \\
x_k & = & f\ (x_i, x_j) & & x_k & = & f\ (x_i, x_j) \\
 & & & & \grave{x}_i & = & \grave{x}_i + (\mathcal{D}_1\ f\ (x_i, x_j)\ \grave{x}_k) \\
 & & & & \grave{x}_j & = & \grave{x}_j + (\mathcal{D}_2\ f\ (x_i, x_j)\ \grave{x}_k)
\end{array}
$$

VLAD: Functional Language for AD—I

- Similar to SCHEME.
- Only functional (side-effect free) constructs are supported.
- The only data types supported are the empty list, Booleans, real numbers, pairs, and procedures that take one argument and return one result. Thus VLAD objects are all of the following type:

$$\tau ::= \textbf{null} \mid \textbf{boolean} \mid \mathbb{R} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$$

- Primitive procedures that take two arguments take them as a pair.
- Except that `cons` is curried.

VLAD: F̲unctional L̲anguage for A̲D̲—II

- We use e_1, e_2 as shorthand for $(\textsc{cons}\ e_1)\ e_2$.
- We allow lambda expressions to have tuples as parameters as shorthand for the appropriate destructuring. For example:

$$\lambda(x_1,(x_2,x_3))\ \ldots x_2 \ldots \ \leadsto\ \lambda x\ \ldots(\textsc{car}\ (\textsc{cdr}\ x))\ldots$$

- $\overrightarrow{\mathcal{J}}$, $\overleftarrow{\mathcal{J}}$

Sensitivity Types

$$\begin{aligned}
\overline{\mathbf{null}} &= \mathbf{null} \\
\overline{\mathbf{boolean}} &= \mathbf{null} \\
\overline{\mathbb{R}} &= \mathbb{R} \\
\overline{\tau_1 \times \tau_2} &= \overline{\tau_1} \times \overline{\tau_2} \\
\overline{\tau_1 \rightarrow \tau_2} &= \mathbf{null}
\end{aligned}$$

The Type of $\overrightarrow{\mathcal{J}}$

$\overrightarrow{\mathcal{J}} : (\tau_1 \rightarrow \tau_2) \rightarrow ((\tau_1 \times \overline{\tau_1}) \rightarrow (\tau_2 \times \overline{\tau_2}))$
$\overrightarrow{\mathcal{J}} : \tau \rightarrow \overrightarrow{\tau}$

$$\begin{array}{rcl}
\overrightarrow{\mathbf{null}} & = & \mathbf{null} \\
\overrightarrow{\mathbf{boolean}} & = & \mathbf{boolean} \\
\overrightarrow{\mathbb{R}} & = & \mathbb{R} \\
\overrightarrow{\tau_1 \times \tau_2} & = & \overrightarrow{\tau_1} \times \overrightarrow{\tau_2} \\
\overrightarrow{\tau_1 \rightarrow \tau_2} & = & (\tau_1 \times \overline{\tau_1}) \rightarrow (\tau_2 \times \overline{\tau_2})
\end{array}$$

The Definition of $\overrightarrow{\mathcal{J}}$ on Non-Closures

$$
\begin{aligned}
\overrightarrow{\mathcal{J}}\; x &= x\\
\overrightarrow{\mathcal{J}}\; (x_1, x_2) &= (\overrightarrow{\mathcal{J}}\; x_1), (\overrightarrow{\mathcal{J}}\; x_2)\\
\overrightarrow{\mathcal{J}}\; f &= \lambda(x, \acute{x})\; (f\; x), (\acute{x}\; (\mathcal{D}\; f\; x))\\
\overrightarrow{\mathcal{J}}\; f &= \lambda((x_1, x_2), (\acute{x}_1, \acute{x}_2))\\
&\ \ (f\; (x_1, x_2)), ((\acute{x}_1\; (\mathcal{D}_1\; f\; (x_1, x_2))) + (\acute{x}_2\; (\mathcal{D}_2\; f\; (x_1, x_2))))\\
\overrightarrow{\mathcal{J}}\; f &= \lambda(x, \acute{x})\; (f\; x), [\,]\\
\overrightarrow{\mathcal{J}}\; f &= \lambda((x_1, x_2), (\acute{x}_1, \acute{x}_2))\; (f\; (x_1, x_2)), [\,]\\
\overrightarrow{\mathcal{J}}\; \textsc{car} &= \lambda((x_1, x_2), (\acute{x}_1, \acute{x}_2))\; x_1, \acute{x}_1\\
\overrightarrow{\mathcal{J}}\; \textsc{cons} &= \lambda(x_1, \acute{x}_1)\; \lambda(x_2, \acute{x}_2)\; (x_1, x_2), (\acute{x}_1, \acute{x}_2)
\end{aligned}
$$

The Definition of $\overrightarrow{\mathcal{J}}$ on Closures

$$\overrightarrow{\mathcal{J}}\ \langle\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, \lambda x\ e\rangle = \langle\{x_1 \mapsto \overrightarrow{\mathcal{J}}\ v_1, \ldots, x_n \mapsto \overrightarrow{\mathcal{J}}\ v_n\}, \overrightarrow{\lambda x\ e}\rangle$$

$$\begin{array}{rcll}
\overrightarrow{x} & \leadsto & x & \text{when } x \text{ is bound}\\
\overrightarrow{x} & \leadsto & x, (\underline{0}\ x) & \text{when } x \text{ is free}\\
\overrightarrow{e_1\ e_2} & \leadsto & (\textsc{car}\ \overrightarrow{e_1})\ \overrightarrow{e_2} & \\
\overrightarrow{\lambda x\ e} & \leadsto & (\lambda x\ \overrightarrow{e}), [\,] &
\end{array}$$

$$\begin{array}{rl}
\underline{0}\ x \stackrel{\triangle}{=} & \textbf{if}\ (\textsc{real?}\ x)\ \textbf{then}\ 0\\
& \textbf{elif}\ (\textsc{pair?}\ x)\ \textbf{then}\ (\underline{0}\ (\textsc{car}\ x)), (\underline{0}\ (\textsc{cdr}\ x))\\
& \textbf{else}\ [\,]\ \textbf{fi}
\end{array}$$

The Type of $\overleftarrow{\mathcal{J}}$

$\overleftarrow{\mathcal{J}} : (\tau_1 \rightarrow \tau_2) \rightarrow ((\tau_1 \times (\overline{\tau_1} \rightarrow \overline{\tau_3})) \rightarrow (\tau_2 \times (\overline{\tau_2} \rightarrow \overline{\tau_3})))$
$\overleftarrow{\mathcal{J}} : \tau \rightarrow \overleftarrow{\tau}$

$$
\begin{array}{rcl}
\overleftarrow{\textbf{null}} & = & \textbf{null} \\
\overleftarrow{\textbf{boolean}} & = & \textbf{boolean} \\
\overleftarrow{\mathbb{R}} & = & \mathbb{R} \\
\overleftarrow{\tau_1 \times \tau_2} & = & \overleftarrow{\tau_1} \times \overleftarrow{\tau_2} \\
\overleftarrow{\tau_1 \rightarrow \tau_2} & = & (\tau_1 \times (\overline{\tau_1} \rightarrow \overline{\tau_3})) \rightarrow (\tau_2 \times (\overline{\tau_2} \rightarrow \overline{\tau_3}))
\end{array}
$$

The Definition of $\overleftarrow{\mathcal{J}}$ on Non-Closures

$$
\begin{aligned}
\overleftarrow{\mathcal{J}}\ x &= x\\
\overleftarrow{\mathcal{J}}\ (x_1, x_2) &= (\overleftarrow{\mathcal{J}}\ x_1), (\overleftarrow{\mathcal{J}}\ x_2)\\
\overleftarrow{\mathcal{J}}\ f &= \lambda(x, \tilde{x})\ (f\ x), (\tilde{x}\ (\mathcal{D}\ f\ x))\\
\overleftarrow{\mathcal{J}}\ f &= \lambda((x_1, x_2), \tilde{x})\\
&\quad\ (f\ (x_1, x_2)), (\tilde{x}\ ((\mathcal{D}_1\ f\ (x_1, x_2)), (\mathcal{D}_2\ f\ (x_1, x_2))))\\
\overleftarrow{\mathcal{J}}\ f &= \lambda(x, \tilde{x})\ (f\ x), \lambda\grave{y}\ \underline{0}\ x\\
\overleftarrow{\mathcal{J}}\ f &= \lambda((x_1, x_2), \tilde{x})\ (f\ (x_1, x_2)), \lambda\grave{y}\ \underline{0}\ (x_1, x_2)\\
\overleftarrow{\mathcal{J}}\ \textsc{car} &= \lambda((x_1, x_2), \tilde{x})\ x_1, \lambda\grave{y}\ \grave{y}, (\underline{0}\ x_2)\\
\overleftarrow{\mathcal{J}}\ \textsc{cons} &= \lambda(x_1, \tilde{x}_1)\\
&\quad\ (\lambda(x_2, \tilde{x}_2)\ (x_1, x_2), \lambda\grave{y}\ (\tilde{x}_1\ (\textsc{car}\ \grave{y})) \oplus (\tilde{x}_2\ (\textsc{cdr}\ \grave{y}))), \lambda\grave{y}\ \tilde{x}_1\ (\underline{0}\ x_1)
\end{aligned}
$$

$$
\begin{aligned}
x_1 \oplus x_2 \stackrel{\triangle}{=}\ & \textbf{if } \textsc{null}?\ x_1 \textbf{ then } [\,]\\
& \textbf{elif } \textsc{real}?\ x_1 \textbf{ then } x_1 + x_2\\
& \textbf{else } ((\textsc{car}\ x_1) \oplus (\textsc{car}\ x_2)), ((\textsc{cdr}\ x_1) \oplus (\textsc{cdr}\ x_2)) \textbf{ fi}
\end{aligned}
$$

The Definition of $\overleftarrow{\mathcal{J}}$ on Closures

$$\overleftarrow{\mathcal{J}}\ \langle\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, \lambda x\ e\rangle = \langle\{x_1 \mapsto \overleftarrow{\mathcal{J}}\ v_1, \ldots, x_n \mapsto \overleftarrow{\mathcal{J}}\ v_n\}, \overleftarrow{\lambda x\ e}\rangle$$

$$\begin{array}{rcll}
\overleftarrow{x} & \leadsto & x & \text{when } x \text{ is bound}\\
\overleftarrow{x} & \leadsto & x, \lambda y\ (\textsc{cdr}\ x_0)\ (\underline{0}\ (\textsc{car}\ x_0)) & \text{when } x \text{ is free}\\
\overleftarrow{e_1\ e_2} & \leadsto & (\textsc{car}\ \overleftarrow{e_1})\ \overleftarrow{e_2} & \\
\overleftarrow{\lambda x\ e} & \leadsto & (\lambda x\ \overleftarrow{e}), \lambda y\ (\textsc{cdr}\ x_0)\ (\underline{0}\ (\textsc{car}\ x_0)) &
\end{array}$$

Fanout—The Problem

$$
\begin{array}{lrcl}
\lambda x_0\ \textbf{let} & x_1 & \stackrel{\triangle}{=} & x_0 + x_0; \\
 & x_2 & \stackrel{\triangle}{=} & x_1 + x_1; \\
 & & \vdots & \\
 & x_n & \stackrel{\triangle}{=} & x_{n-1} + x_{n-1} \\
 & \multicolumn{3}{l}{\textbf{in}\ x_n\ \textbf{end}}
\end{array}
$$

Fanout—One Solution

$$\textsc{fan} \stackrel{\triangle}{=} \lambda f\ \lambda x\ f\ (x, x)$$

$$\lambda x\ x + x + x \rightsquigarrow \lambda x\ \textsc{fan}\ (\lambda(x_1, x)\textsc{fan}\ (\lambda(x_2, x_3)x_1 + x_2 + x_3)\ x)\ x$$

$$\begin{aligned}\overleftarrow{\mathcal{J}}\ \textsc{fan} \stackrel{\triangle}{=} \lambda(f, \tilde{f})\ (\lambda(x, \tilde{x})\ &\textbf{let}\ \hat{y} \stackrel{\triangle}{=} f\ ((x, x), I);\ y \stackrel{\triangle}{=} \textsc{car}\ \hat{y};\ \tilde{y} \stackrel{\triangle}{=} \textsc{cdr}\ \hat{y} \\ &\textbf{in}\ y, \lambda \grave{y}\ \textbf{let}\ \grave{x} \stackrel{\triangle}{=} \tilde{y}\ \grave{y} \\ &\qquad\qquad \textbf{in}\ \tilde{x}\ ((\textsc{car}\ \grave{x}) \oplus (\textsc{cdr}\ \grave{x}))\ \textbf{end}\ \textbf{end}), \\ \lambda \grave{y}\ \tilde{f}\ \underline{0}\ f\ &\end{aligned}$$

Derivatives

$$
\begin{aligned}
\mathcal{D}\ f &\stackrel{\triangle}{=} \lambda x\ \textsc{cdr}\ (\overrightarrow{\mathcal{J}}\ f\ (x, 1)) \\
\mathcal{D}\ f &\stackrel{\triangle}{=} \lambda x\ (\textsc{cdr}\ (\overleftarrow{\mathcal{J}}\ f\ (x, I)))\ 1
\end{aligned}
$$

Roots using Newton-Raphson

$$
\begin{aligned}
\textsc{Root}\ (f, x, \epsilon) \stackrel{\triangle}{=}\ & \mathbf{let}\ x' \stackrel{\triangle}{=} x - \tfrac{x}{\mathcal{D}\ f\ x} \\
& \mathbf{in\ if}\ |x - x'| \leq \epsilon\ \mathbf{then}\ x\ \mathbf{else}\ \textsc{Root}\ (f, x', \epsilon)\ \mathbf{fi\ end}
\end{aligned}
$$

Univariate Optimizer (Line Search)

$$\textsc{Argmin}\ (f, x, \epsilon) \stackrel{\triangle}{=} \textsc{Root}\ ((\mathcal{D}\ f), x, \epsilon)$$

Gradients

$$
\begin{aligned}
\nabla\ f \quad &\triangleq \quad \lambda x\ \textbf{let}\ n \triangleq \textsc{Length}\ x \\
&\qquad\quad \textbf{in}\ \textsc{Map}\ ((\lambda i\ \textsc{cdr}\ (\overrightarrow{\mathcal{J}}\ f\ (x, (e\ (1, i, n))))), (\iota\ n))\ \textbf{end} \\
\nabla\ f \quad &\triangleq \quad \lambda x\ (\textsc{cdr}\ (\overleftarrow{\mathcal{J}}\ f\ (x, I)))\ 1
\end{aligned}
$$

Gradient Descent

$$
\begin{array}{l}
\textsc{GradientDescent}\ (f, x, \epsilon) \stackrel{\triangle}{=} \\
\quad \textbf{let}\ g \stackrel{\triangle}{=} \nabla\ f\ x \\
\quad \textbf{in}\ \textbf{if}\ ||g|| \leq \epsilon \\
\qquad \textbf{then}\ x \\
\qquad \textbf{else}\ \textsc{GradientDescent}\ (f, (x + (\textsc{Argmin}\ ((\lambda k\ f\ (x + kg)), 0, \epsilon))\ g), \epsilon) \\
\qquad\quad \textbf{fi}\ \textbf{end}
\end{array}
$$

Function Inversion

$$f^{-1} \stackrel{\triangle}{=} \lambda y\ \textsc{Root}\ ((\lambda x\ |(f\ x) - y|), x_0, \epsilon)$$

A Rational Agent

- The world is $w : \textbf{state} \times \textbf{action} \rightarrow \textbf{state}$
- Agent perception is $p_B : \textbf{state} \rightarrow \textbf{observation}$
- Agent reward is $r_B : \textbf{observation} \rightarrow \mathbb{R}$
- Goal is to maximize $r_B(p_B(w(s,a)))$
- But agent doesn't have s, w, p_B, and r_B
- Observation $o = p_B(s)$
- Models w_B, p_{BB}, and r_{BB} of w, p_B, and r_B respectively

$$\textsc{Agent}\ (w_B, p_{BB}, r_{BB}, o) \stackrel{\triangle}{=} \textsc{Argmax}\ ((\lambda a\ r_{BB}\ (p_{BB}\ (w_B\ ((p_{BB}^{-1}\ o), a)))), a_0, \epsilon)$$

A Pair of Interacting Rational Agents (von Neumann & Morgenstern 1944)

$$
\begin{array}{l}
\textsc{DoubleAgent}\ (w_A, w_{AB}, p_{AA}, p_{AB}, p_{ABB}, r_{AA}, r_{ABB}, o) \triangleq \\
\quad \textsc{Argmax} \\
\quad (\,(\lambda a\ r_{AA} \\
\qquad\quad (\,p_{AA} \\
\qquad\qquad (w_A\,(\,(w_A\,((p_{AA}^{-1}\ o), a)), \\
\qquad\qquad\qquad (\,\textsc{Argmax} \\
\qquad\qquad\qquad\quad ((\lambda a'\ r_{ABB}\ (p_{ABB}\ (w_{AB}\ ((p_{ABB}^{-1}\ (p_{AB}\ (w_A\ ((p_{AA}^{-1}\ o), a)))), a')))), \\
\qquad\qquad\qquad\quad a_0, \epsilon)))))), \\
\quad\ a_0, \epsilon)
\end{array}
$$

Carl Gauss
|
Christoph Gudermann
|
Karl Weierstrass
|
Hermann Schwarz
|
Leopold Fejér
|
John von Neumann

Neural Nets
(Rumelhart, Hinton, & Williams 1986)

$$
\begin{aligned}
\textsc{Neuron}\ (w, x) &\stackrel{\triangle}{=} \textsc{Sigmoid}\ (w \cdot x) \\
\textsc{NeuralNet}\ (w, x) &\stackrel{\triangle}{=} \textsc{Neuron}\ (w'', \ldots \textsc{Neuron}\ (w', x') \ldots) \\
\textsc{Error}\ w &\stackrel{\triangle}{=} ||[y_1; \ldots; y_n] - \\
&\quad\ [\textsc{NeuralNet}\ (w, x_1); \ldots; \textsc{NeuralNet}\ (w, x_n)]||
\end{aligned}
$$

$\textsc{GradientDescent}\ (\textsc{Error}, w_0, \epsilon)$

Supervised Machine Learning (Function Approximation)

$$\textsc{Error}\ \theta \stackrel{\triangle}{=} ||[y_1; \ldots; y_n] - [f\ (\theta, x_1); \ldots; f\ (\theta, x_n)]||$$

$$\textsc{GradientDescent}\ (\textsc{Error}, \theta_0, \epsilon)$$

Maximum Likelihood Estimation (Fisher 1921)

$$\textsc{Argmax}\left(\left(\lambda\theta\ \prod_{x\in X} P(x|\theta)\right), \theta_0, \epsilon\right)$$

Engineering Design

$$
\begin{aligned}
&\textsc{PerformanceOf}\ \textsc{SplineControlPoints} \triangleq\\
&\quad \textbf{let}\ \ \textsc{wing} \triangleq \textsc{SplineToSurface}\ \textsc{SplineControlPoints};\\
&\qquad\ \ \textsc{airflow} \triangleq \textsc{PDEsolver}\ (\textsc{wing}, \textsc{NavierStokes});\\
&\qquad\ \ \textsc{lift}, \textsc{drag} \triangleq \textsc{SurfaceIntegral}\ (\textsc{wing}, \textsc{airflow}, \textsc{force});\\
&\qquad\ \ \textsc{performance} \triangleq \textsc{DesignMetric}\ (\textsc{lift}, \textsc{drag}, (\textsc{weight}\ \textsc{wing}))\\
&\quad \textbf{in}\ \textsc{performance}\ \textbf{end}
\end{aligned}
$$

$$\textsc{GradientDescent}\ (\textsc{PerformanceOf}, \textsc{SplineControlPoints}_0, \epsilon)$$

An Optimizing Compiler for VLAD

STALIN∇:

- polyvariant flow analysis (Shivers 1988)
- flow-directed lightweight closure conversion (Wand & Steckler 1994)
- flow-directed inlining
- compiling with continuations (Steele 1979, Appel 1992)
- unboxing
- partial evaluation

Alonzo Church
|
Stephen Cole Kleene
|
Robert Lee Constable
|
Steven Stanley Muchnick
|
Uwe Frederik Pleban
|
Peter Lee
|
Olin Shivers

Advantages—I

Functional programs represent the underlying mathematical notions more closely than imperative programs.

Advantages—II

Greater compositionality:

- root finders built on a derivative-taker
- line search built on root finders
- multivariate optimizers built on line search
- other multivariate optimizers (with identical APIs) build on Hessian-vector multipliers

  ⋮

Advantages—III

Greater modularity: by allowing the callee to specify the necessary AD, rather than insisting that the caller provide appropriately transformed functions, internals can be hidden and changed.

Advantages—IV

It is straightforward to generate higher-order derivatives, i.e. derivatives of derivatives.

Advantages—V

Differential forms become first-class higher-order functions that can be passed to optimizers or PDE solvers as part of an API. This allow one to easily express programming patterns, i.e. algorithm templates, that can be instantiated with different components as fillers. For example, one can construct an algorithm that needs an optimizer and leave the choice of optimizer unspecified, to be filled in later by passing the particular optimizer as a function parameter.

Advantages—VI

Gradients can even be taken through processes that themselves involve AD-based optimization or PDE solution.

Advantages—VII

In traditional AD formulations, the output of a reverse-mode transformation is a 'tape' that is a different kind of entity than user-written functions. It must be interpreted or run-time compiled. In contrast, in our approach, user-written functions, and the input and output of AD operators, are all the same kind of entity. Standard compilation techniques for functional programs can eliminate the need for interpretation or run-time compilation of derivatives and generate, at compile-time, code for derivatives that is as efficient as code for the primal calculation.