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.
Differential Calculus for Dummies
(in 6 slides)
Notation

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

\[
\frac{d}{dx} : \frac{f}{\mathbb{R} \to \mathbb{R}} \to \frac{f'}{\mathbb{R} \to \mathbb{R}}
\]

\[
\frac{d}{dx} : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})
\]

\[
D : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})
\]
Partial Derivatives

\[ \frac{\partial}{\partial x} : \begin{array}{c} f \colon \mathbb{R}^n \to \mathbb{R} \\ R^n \to R \end{array} \to \begin{array}{c} f' \colon \mathbb{R}^n \to \mathbb{R} \\ R^n \to R \end{array} \]

\[ \frac{\partial}{\partial x} : (\mathbb{R}^n \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R}) \]

\[ D_i : (\mathbb{R}^n \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R}) \]
Gradients

$$\nabla f \mathbf{x} = (\mathcal{D}_1 f \mathbf{x}), \ldots, (\mathcal{D}_n f \mathbf{x})$$

$$\nabla : (\mathbb{R}^n \to \mathbb{R}) \to (\mathbb{R}^n \to \mathbb{R}^n)$$
Jacobians

\[ f : \mathbb{R}^m \to \mathbb{R}^n \]

\[ f : (\mathbb{R}^m \to \mathbb{R})^n \]

\[ (J f x)[i, j] = (\nabla (f[i]))[j] \]

\[ J : (\mathbb{R}^m \to \mathbb{R}^n) \to (\mathbb{R}^m \to \mathbb{R}^{m \times n}) \]
Operators

\( \mathcal{D}, \nabla, \) 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

\[(f \circ g) \, x = (g \, f) \, x = g \, (f \, x)\]

\[\mathcal{D} \, (g \, f) \, x = (\mathcal{D} \, g \, f \, x) \, (\mathcal{D} \, f \, x)\]

\[\mathcal{J} \, (g \, f) \, x = (\mathcal{J} \, g \, f \, x) \, (\mathcal{J} \, f \, x)\]
Everything You Always Wanted to Know About the Lambda Calculus*

(in 7 slides)

*But Were Afraid To Ask
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.
int f(int n)
{ int i, p = 1;
  for (i = 1; i<n; i++)
    { p = p*i;}
  return p;}

\[ f \triangleq \begin{cases} 
  1 & \text{if } n = 0 \\
  n \times f(n-1) & \text{else} 
\end{cases} \]
Higher-Order Functions

\[ \sum_{i=1}^{n} \exp(i) \]
\[ \prod_{i=1}^{n} \sin(i) \]

\[
\text{FOLD}(i, a, f, g) \triangleq \begin{cases} 
  a & \text{if } i = 0 \\
  \text{FOLD}((i - 1), g(a,(f\ i))), f, g) & \text{else}
\end{cases}
\]

\[
\sum_{i=1}^{n} 2i + 1 \]

\[
f_i \triangleq 2i + 1
\]

\[
\text{FOLD}(n, 0, f, +)
\]
\[
\text{FOLD}(n, 0, (\lambda i \ 2i + 1), +)
\]
Closures

\[(\lambda x \, 2x) \, 3 = 6\]

\[((\lambda x \lambda y \, x + y) \, 3) \, 4 = 7\]

\[(\lambda x \lambda y \, x + y) \, 3 = ?\]

\[(\lambda x \lambda y \, x + y) \, 3 = (\{x \mapsto 3\}, \lambda y \, x + y)\]

\[\lambda x \lambda y \, x + y\]

\[\lambda (x, y) \, x + y\]
Tail Recursion (Steele 1976)

\[
f n \triangleq \begin{cases} 
  \text{if } n = 0 & \text{then } 1 \\
  \text{else } n \times (f \ (n - 1)) & \text{fi}
\end{cases}
\]

\[
g \ (i, p) \triangleq \begin{cases} 
  \text{if } i = 0 & \text{then } p \\
  \text{else } g \ ((i - 1), (p \times i)) & \text{fi}
\end{cases}
\]

\[
f n \triangleq g(n, 1)
\]
Continuations (Landin 1965, Reynolds 1972)

\[ f x \triangleq e_1 \quad f'(c, x) \triangleq e e_1 \]
\[ g x \triangleq e_2 \quad g'(c, x) \triangleq e e_2 \]
\[ h x \triangleq e_3 \quad h'(c, x) \triangleq e e_3 \]
\[ p x \triangleq h(g(f x)) \quad p'(c, x) \triangleq f((\lambda x_1 g((\lambda x_2 h(c, x_2)), x_1)), x) \]
The Lambda Calculus

\[
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ fi } \sim ((\text{if } e_1) (\lambda x \ e_2)) (\lambda x \ e_3) \] [ ]

\[ e ::= x \mid e_1 \ e_2 \mid \lambda x \ e \]
Compositional Derivative Operators—I

\[ f_n \cdots f_1 \]

\[ \mathcal{J} (f_n \cdots f_1) \]

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

\( \mathcal{J} (f_n \cdots f_1) \) is not compositional in \((\mathcal{J} f_1), \ldots, (\mathcal{J} f_n)\).
Compositional Derivative Operators—II

\[ \nabla f \triangleq \lambda(x, \hat{x}) J f x \hat{x} \]

\[ \nabla f \triangleq \lambda(x, \hat{y}) (J f x)^T \hat{y} \]

- \( x \) is a **primal** variable
- \( \hat{x} \) is a **forward adjoint** variable
- \( \check{x} \) is a **reverse adjoint** variable

The rows and columns of \( J f x \) can be computed as \( \nabla f (x, e) \) and \( \nabla f (x, e) \) for basis vectors \( e \) respectively.
Compositional Derivative Operators—III

\[ \nabla (g f) = \lambda(x, \dot{x}) \nabla g \left((f x), (\nabla f (x, \dot{x}))\right) \]

\[ \check{\nabla} (g f) = \lambda(x, \dot{y}) \check{\nabla} f \left(x, (\nabla g ((f x), \dot{y}))\right) \]

One cannot compose \( \check{\nabla} f \) with \( \nabla g \) because the input and output of \( \check{\nabla} f \) are not of the same type. Similarly for \( \check{\nabla} f \).
Compositional Derivative Operators—IV

\[ \mathcal{J} f \triangleq \lambda(x, \hat{x}) (f(x), (\nabla f(x, \hat{x}))) \]
\[ \mathcal{F} f \triangleq \lambda(x, \check{x}) (f(x), (\hat{\check{x}} \lambda \check{y} \nabla f(x, \check{y}))) \]

- \( \lambda \check{y} \nabla f(x, \check{y}) \) is a local backpropagator
- \( \hat{x} \) is an input backpropagator
- their composition as an output backpropagator
Compositional Derivative Operators—V

\textbf{Adjoint} (x, \dot{x}) = \dot{x}
\textbf{Backpropagator} (x, \ddot{x}) = \ddot{x}

\frac{\partial f}{\partial x} = \lambda(x, \dot{x}) \text{ Adjoint} (\mathcal{J}(x, \dot{x}))
\frac{\partial f}{\partial x} = \lambda(x, \dot{y}) \text{ Backpropagator} (\mathcal{J}(x, I)) \dot{y}

\mathcal{J} (g f) = (\mathcal{J} g) (\mathcal{J} f)
\mathcal{J} (g f) = (\mathcal{J} g) (\mathcal{J} f)
\mathcal{J} (f_n \cdots f_1) = (\mathcal{J} f_n) \cdots (\mathcal{J} f_1)
\mathcal{J} (f_n \cdots f_1) = (\mathcal{J} f_n) \cdots (\mathcal{J} f_1)
Traditional Forward-Mode AD—I

\[
\begin{align*}
    x_1 &= f_1(x_0) \\
    x_2 &= f_2(x_1) \\
    & \vdots \\
    x_n &= f_n(x_{n-1})
\end{align*}
\]

\[
\begin{align*}
    \dot{x}_1 &= f_1(x_0) \\
    \dot{x}_2 &= f_2(x_1) \\
    & \vdots \\
    \dot{x}_n &= f_n(x_{n-1})
\end{align*}
\]
Traditional Forward-Mode AD—II

\[
x_j = f(x_i) \quad x_j, \dot{x}_j = (f(x_i), ((\mathcal{D} f(x_i)) \dot{x}_i)) \\
x_k = f(x_i, x_j) \quad x_k, \dot{x}_k = (f(x_i, x_j), ((\mathcal{D}_1 f(x_i, x_j)) \dot{x}_i) + (\mathcal{D}_2 f(x_i, x_j) \dot{x}_j))
\]
Traditional Reverse-Mode AD—I

\[ x_1 = f_1 x_0 \]
\[ x_2 = f_2 x_1 \]
\[ \vdots \]
\[ x_n = f_n x_{n-1} \]

\[ x_1, \dot{x}_1 = \mathcal{J} f_1 (x_0, \dot{x}_0) \]
\[ x_2, \dot{x}_2 = \mathcal{J} f_2 (x_1, \dot{x}_1) \]
\[ \vdots \]

\[ x_n, \dot{x}_n = \mathcal{J} f_n (x_{n-1}, \dot{x}_{n-1}) \]

\[ (\dot{x}_n I) \dot{x}_n \]
Traditional Reverse-Mode AD—II

\[ x_1 = f_1 x_0 \quad x_1, \bar{x}_1 = \mathcal{J} f_1 (x_0, \bar{x}_0) \]
\[ x_2 = f_2 x_1 \quad x_2, \bar{x}_2 = \mathcal{J} f_2 (x_1, \bar{x}_1) \]
\[ \vdots \]
\[ x_n = f_n x_{n-1} \quad x_n, \bar{x}_n = \mathcal{J} f_n (x_{n-1}, \bar{x}_{n-1}) \]
\[ \bar{x}_{n-1} = \nabla f_n (x_n, \bar{x}_n) \]
\[ \bar{x}_{n-2} = \nabla f_{n-1} (x_{n-1}, \bar{x}_{n-1}) \]
\[ \vdots \]
\[ \bar{x}_0 = \nabla f_1 (x_1, \bar{x}_1) \]
Traditional Reverse-Mode AD—III

\[
\begin{align*}
x_j &= f_{x_i} & x_i &= f_{x_i} \\
\dot{x}_i &= \dot{x}_i + (D f_{x_i} \dot{x}_j) \\
x_k &= f_{(x_i, x_j)} & x_k &= f_{(x_i, x_j)} \\
\dot{x}_i &= \dot{x}_i + (D_1 f_{(x_i, x_j)} \dot{x}_k) \\
\dot{x}_j &= \dot{x}_j + (D_2 f_{(x_i, x_j)} \dot{x}_k)
\end{align*}
\]
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:

\[ \tau ::= \textbf{null} \mid \textbf{boolean} \mid \mathbb{R} \mid \tau_1 \times \tau_2 \mid \tau_1 \rightarrow \tau_2 \]

• Primitive procedures that take two arguments take them as a pair.

• Except that \texttt{cons} is curried.
We use $e_1, e_2$ as shorthand for $(\text{cons } e_1) e_2$.

We allow lambda expressions to have tuples as parameters as shorthand for the appropriate destructuring. For example:

$$\lambda(x_1, (x_2, x_3)) \ldots x_2 \ldots \sim \lambda x \ldots (\text{car} (\text{cdr} x)) \ldots$$
Sensitivity Types

null = null

Boolean = null

$\mathbb{R} = \mathbb{R}$

$\tau_1 \times \tau_2 = \tau_1 \times \tau_2$

$\tau_1 \rightarrow \tau_2 = \text{null}$
The Type of $\vec{J}$

$\vec{J} : (\tau_1 \rightarrow \tau_2) \rightarrow ((\tau_1 \times \tau_1) \rightarrow (\tau_2 \times \tau_2))$

$\vec{J} : \tau \rightarrow \tau$

$\underline{null} = \text{null}$

$\underline{boolean} = \text{boolean}$

$\underline{R} = R$

$\underline{\tau_1 \times \tau_2} = \tau_1 \times \tau_2$

$\underline{\tau_1 \rightarrow \tau_2} = (\tau_1 \times \tau_1) \rightarrow (\tau_2 \times \tau_2)$
The Definition of $\mathcal{J}$ on Non-Closures

\[\mathcal{J} x = x\]
\[\mathcal{J} (x_1, x_2) = (\mathcal{J} x_1, (\mathcal{J} x_2))\]
\[\mathcal{J} f = \lambda(x, \dot{x}) (f x, (\dot{x} (D f x)))\]
\[\mathcal{J} f = \lambda((x_1, x_2), (\dot{x}_1, \dot{x}_2))\]
\[f (x_1, x_2), ((\dot{x}_1 (D_1 f (x_1, x_2))) + (\dot{x}_2 (D_2 f (x_1, x_2))))\]
\[\mathcal{J} f = \lambda(x, \dot{x}) (f x), [[]\]
\[\mathcal{J} f = \lambda((x_1, x_2), (\dot{x}_1, \dot{x}_2)) (f (x_1, x_2)), [[]\]
\[\mathcal{J} \text{ CAR} = \lambda((x_1, x_2), (\dot{x}_1, \dot{x}_2)) x_1, \dot{x}_1\]
\[\mathcal{J} \text{ CONS} = \lambda(x_1, \dot{x}_1) \lambda(x_2, \dot{x}_2) (x_1, x_2), (\dot{x}_1, \dot{x}_2)\]
The Definition of $\vec{\mathcal{J}}$ on Closures

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

$\vec{x} \leadsto x$ when $x$ is bound

$\vec{x} \leadsto x, (\emptyset \ x)$ when $x$ is free

$\vec{e_1 \ e_2} \leadsto (\text{CAR} \ \vec{e_1}) \ \vec{e_2}$

$\lambda x \ e \leadsto (\lambda x \ \vec{e}), [\ ]$

$0 \ x \triangleq \begin{cases} \text{if} \ (\text{REAL?} \ x) \ \text{then} \ 0 \\ \text{elif} \ (\text{PAIR?} \ x) \ \text{then} \ (\emptyset \ (\text{CAR} \ x)), (\emptyset \ (\text{CDR} \ x)) \\ \text{else} \ [\ ] \ \text{fi} \end{cases}$
The Type of $\mathcal{J}$

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

$\mathcal{J} : \tau \rightarrow \tau$

\[
\begin{align*}
\text{null} & = \text{null} \\
\text{boolean} & = \text{boolean} \\
\overline{R} & = R \\
\overline{\tau_1 \times \tau_2} & = \overline{\tau_1} \times \overline{\tau_2} \\
\overline{\tau_1 \rightarrow \tau_2} & = (\tau_1 \times (\overline{\tau_1} \rightarrow \overline{\tau_3})) \rightarrow (\tau_2 \times (\overline{\tau_2} \rightarrow \overline{\tau_3}))
\end{align*}
\]
The Definition of $\mathcal{J}$ on Non-Closures

$\mathcal{J} x = x$

$\mathcal{J} (x_1, x_2) = (\mathcal{J} x_1), (\mathcal{J} x_2)$

$\mathcal{J} f = \lambda(x, \bar{x}) (f x), (\bar{x} (D f x))$

$\mathcal{J} f = \lambda((x_1, x_2), \bar{x})$

$(f (x_1, x_2)), (\bar{x} ((D_1 f (x_1, x_2)), (D_2 f (x_1, x_2))))$

$\mathcal{J} f = \lambda(x, \bar{x}) (f x), \lambda y \emptyset x$

$\mathcal{J} f = \lambda((x_1, x_2), \bar{x}) (f (x_1, x_2)), \lambda y \emptyset (x_1, x_2)$

$\mathcal{J} \text{CAR} =\lambda((x_1, x_2), \bar{x}) x_1, \lambda y \emptyset (0 x_2)$

$\mathcal{J} \text{CONS} = \lambda(x_1, \bar{x}_1)$

$(\lambda(x_1, x_2), \lambda y (\bar{x}_1 (\text{CAR} y)) \oplus (\bar{x}_2 (\text{CDR} y))), \lambda y \bar{x}_1 (0 x_1)$

$x_1 \oplus x_2 \triangleq\text{if \ NULL?} \ x_1 \ \text{then} \ [] \ \text{elif} \ \text{REAL?} \ x_1 \ \text{then} \ x_1 + x_2 \ \text{else} \ ((\text{CAR} \ x_1) \oplus (\text{CAR} \ x_2)), ((\text{CDR} \ x_1) \oplus (\text{CDR} \ x_2)) \ \text{fi}$
The Definition of $\mathcal{J}$ on Closures

$\mathcal{J}(\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}, \lambda x \ e) = (\{x_1 \mapsto \mathcal{J} v_1, \ldots, x_n \mapsto \mathcal{J} v_n\}, \lambda x \ e)$

- $\tilde{x} \leadsto x$ when $x$ is bound
- $\tilde{x} \leadsto x, \lambda y \ (\text{CDR } x_0) \ (\text{CAR } x_0)$ when $x$ is free
- $\tilde{e_1 \ e_2} \leadsto (\text{CAR } \tilde{e_1}) \ \tilde{e_2}$
- $\lambda x \ e \leadsto (\lambda x \ \tilde{e}), \lambda y \ (\text{CDR } x_0) \ (\text{CAR } x_0)$
Fanout—The Problem

\[ \lambda x_0 \text{ let } x_1 \triangleq x_0 + x_0; \]
\[ x_2 \triangleq x_1 + x_1; \]
\[ \vdots \]
\[ x_n \triangleq x_{n-1} + x_{n-1} \]
\[ \text{in } x_n \text{ end} \]
Fanout—One Solution

\[ \text{FAN} \triangleq \lambda f \lambda x \ f \ (x, x) \]

\[ \lambda x \ x + x \leadsto \lambda x \ \text{FAN} \ (\lambda(x_1, x) \ \text{FAN} \ (\lambda(x_2, x_3) x_1 + x_2 + x_3) \ x) \ x \]

\[ \overleftarrow{\text{FAN}} \triangleq \lambda (f, \tilde{f}) \ (\lambda(x, \tilde{x}) \ \text{let} \ \hat{y} \triangleq f ((x, x), I); \ \hat{y} \triangleq \text{CAR} \ \hat{y}; \ \hat{y} \triangleq \text{CDR} \ \hat{y} \]

\[ \begin{align*}
\text{in } y, \lambda \hat{y} \ \text{let} \ & \ \hat{x} \triangleq \hat{y} \ \\ & \text{in } \hat{x} \ ((\text{CAR} \ \hat{x}) \oplus (\text{CDR} \ \hat{x})) \ \text{end} \end{align*} \]

\[ \lambda \hat{y} \ \tilde{f} \ 0 \ f \]
Derivatives

\[ D f \triangleq \lambda x \text{ CDR}(\mathcal{J} f (x, 1)) \]
\[ D f \triangleq \lambda x (\text{CDR}(\mathcal{J} f (x, I))) 1 \]
Roots using Newton-Raphson

\[ \text{Root} \left( f, x, \varepsilon \right) \triangleq \begin{cases} \text{let } x' \triangleq x - \frac{f}{Df(x)} \\ \text{in if } |x - x'| \leq \varepsilon \text{ then } x \text{ else } \text{Root} \left( f, x', \varepsilon \right) \end{cases} \]
Univariate Optimizer (Line Search)

\[ \text{Argmin} \ (f, x, \epsilon) \overset{\Delta}{=} \text{Root} \ ((D f), x, \epsilon) \]
\[ \nabla f \triangleq \lambda x \text{ let } n \triangleq \text{LENGTH } x \\
\text{ in MAP } ((\lambda i \text{ cdr } (J f (x, (e (1, i, n))))), (i, n)) \text{ end} \]

\[ \nabla f \triangleq \lambda x \text{ cdr } (J f (x, I)) 1 \]
Gradient Descent

\[ \text{GradientDescent} \left( f, x, \epsilon \right) \triangleq \]

\[
\begin{align*}
\text{let } & g \triangleq \nabla f \ x \\
\text{in } & \text{if } ||g|| \leq \epsilon \\
& \text{then } x \\
& \text{else } \text{GradientDescent} \left( f, (x + (\text{Argmin} \ ((\lambda k \ f \ (x + kg)), 0, \epsilon)) \ g), \epsilon \right) \\
\text{fi end}
\end{align*}
\]
$f^{-1} \triangleq \lambda y \ Root ((\lambda x \ |(f \ x) - y|), x_0, \epsilon)$
A Rational Agent

- The world is \( w: \text{state} \times \text{action} \to \text{state} \)
- Agent perception is \( p_B: \text{state} \to \text{observation} \)
- Agent reward is \( r_B: \text{observation} \to \mathbb{R} \)
- 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

\[
\text{AGENT} (w_B, p_{BB}, r_{BB}, o) \overset{\triangle}{=} \text{ARGMAX} \left( (\lambda a \, r_{BB} (p_{BB} (w_B ((p_{BB}^{-1} o), a))), a_0, \epsilon) \right)
\]
A Pair of Interacting Rational Agents
(von Neumann & Morgenstern 1944)

\[
\text{DOUBLEAGENT} (w_A, w_{AB}, p_{AA}, p_{AB}, p_{ABB}, r_{AA}, r_{ABB}, o) \triangleq \\
\text{ARGMAX} \\
(\lambda a' r_{AA} \\
(p_{AA} \\
(w_A ((w_A ((p_{AA}^{-1} o), a)), \\
(\text{ARGMAX} \\
((\lambda a' r_{ABB} (p_{ABB} (w_{AB} ((p_{ABB}^{-1} ((p_{AA}^{-1} o), a))))), a')), \\
a_0, o))))), \\
a_0, o, \epsilon))
\]
Carl Gauss
Christoph Gudermann
Karl Weierstrass
Hermann Schwarz
Leopold Fejér
John von Neumann
Neural Nets
(Rumelhart, Hinton, & Williams 1986)

\[
\text{Neuron} (w, x) \triangleq \text{Sigmoid} (w \cdot x)
\]

\[
\text{NeuralNet} (w, x) \triangleq \text{Neuron} (w'', \ldots \text{Neuron} (w', x') \ldots)
\]

\[
\text{Error} w \triangleq ||[y_1; \ldots; y_n] - [	ext{NeuralNet} (w, x_1); \ldots; \text{NeuralNet} (w, x_n)]||
\]

\[
\text{GradientDescent} (\text{Error}, w_0, \epsilon)
\]
Supervised Machine Learning
(Function Approximation)

\[ \text{Error } \theta \triangleq ||[y_1; \ldots; y_n] - [f(\theta, x_1); \ldots; f(\theta, x_n)]|| \]

\text{GradientDescent} (\text{Error}, \theta_0, \epsilon)
Maximum Likelihood Estimation (Fisher 1921)

\[
\text{ARGMAX} \left( \lambda \theta \prod_{x \in X} P(x|\theta), \theta_0, \epsilon \right)
\]
Engineering Design

\[
\text{\textbf{PerformanceOf SplineControlPoints}} \triangleq \\
\text{let wing} \triangleq \text{SplineToSurface SplineControlPoints}; \\
\text{airflow} \triangleq \text{PDEsolver (wing, NavierStokes)}; \\
\text{lift, drag} \triangleq \text{SurfaceIntegral (wing, airflow, force)}; \\
\text{performance} \triangleq \text{DesignMetric (lift, drag, (weight wing))} \\
\text{in performance end}
\]

\[
\text{GradientDescent (PerformanceOf SplineControlPoints}_{0, \epsilon})
\]
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
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 allows 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.
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.