Taking Derivatives of Functional Programs

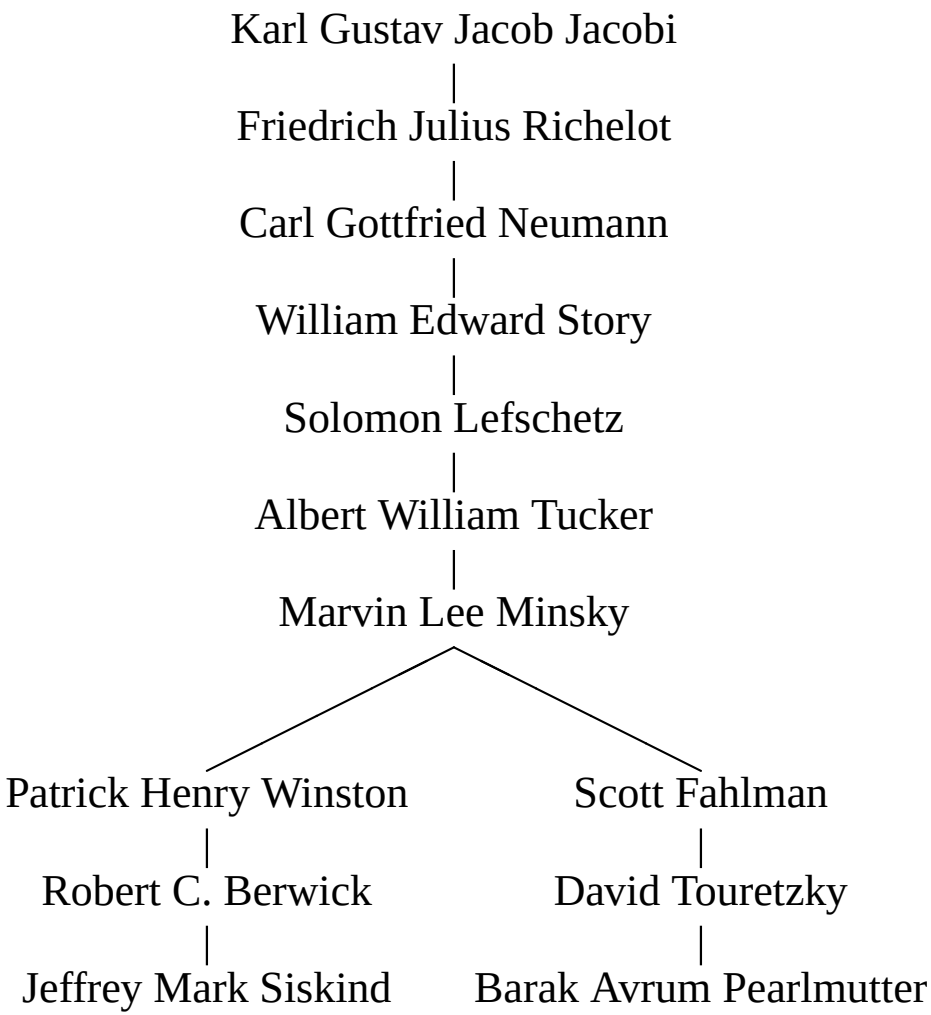
AD in a Functional Framework

Jeffrey Mark Siskind
qobi@purdue.edu

School of Electrical and Computer Engineering
Purdue University

New England Programming Languages and Systems
27 October 2005

Joint work with Barak A. Pearlmutter.


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

Outline

Outline

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.

(p. 1 ¶4)

Church, A. (1941). *The Calculi of Lambda Conversion*. Princeton University Press, Princeton, NJ.

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

Leibnitz (1664) + Church (1941) = Siskind & Pearlmutter (2005)

Leibnitz, G. W. (1664). A new method for maxima and minima as well as tangents, which is impeded neither by fractional nor irrational quantities, and a remarkable type of calculus for this. *Acta Eruditorum*.

Differential Calculus for Dummies

(in 6 slides)

Derivatives

$$\frac{\mathrm{d}ax^2}{\mathrm{d}x} \rightsquigarrow 2ax$$

Derivatives

$$\frac{\mathrm{d}ax^2}{\mathrm{d}x} \rightsquigarrow 2ax$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : \underbrace{f}_{\mathbb{R}\to\mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R}\to\mathbb{R}}$$

Derivatives

$$\frac{\mathrm{d}ax^2}{\mathrm{d}x} \rightsquigarrow 2ax$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : \underbrace{f}_{\mathbb{R}\to\mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R}\to\mathbb{R}}$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

Derivatives

$$\frac{\mathrm{d}ax^2}{\mathrm{d}x} \rightsquigarrow 2ax$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : \underbrace{f}_{\mathbb{R}\to\mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R}\to\mathbb{R}}$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

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

Derivatives

$$\frac{\mathrm{d}ax^2}{\mathrm{d}x} \rightsquigarrow 2ax$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : \underbrace{f}_{\mathbb{R}\to\mathbb{R}} \mapsto \underbrace{f'}_{\mathbb{R}\to\mathbb{R}}$$

$$\frac{\mathrm{d}}{\mathrm{d}x} : (\mathbb{R} \to \mathbb{R}) \to (\mathbb{R} \to \mathbb{R})$$

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

$$\mathcal{D}\ \lambda x\ ax^2$$

Partial Derivatives

$$\frac{\partial a x^2 y^3}{\partial x} \qquad \frac{\partial a x^2 y^3}{\partial y}$$

Partial Derivatives

$$\frac{\partial a x^2 y^3}{\partial x}$$

$$\mathcal{D}\ \lambda x\ a x^2 y^3$$

$$\frac{\partial a x^2 y^3}{\partial y}$$

$$\mathcal{D}\ \lambda y\ a x^2 y^3$$

Partial Derivatives

$$\frac{\partial a x^2 y^3}{\partial x} \qquad \frac{\partial a x^2 y^3}{\partial y}$$

$$\mathcal{D}\ \lambda x\ a x^2 y^3 \qquad \mathcal{D}\ \lambda y\ a x^2 y^3$$

$$\mathcal{D}_1\ \lambda(x,y)\ a x^2 y^3 \qquad \mathcal{D}_2\ \lambda(x,y)\ a x^2 y^3$$

Partial Derivatives

$$\frac{\partial a x^2 y^3}{\partial x} \qquad \frac{\partial a x^2 y^3}{\partial y}$$

$$\mathcal{D}\ \lambda x\ a x^2 y^3 \qquad \mathcal{D}\ \lambda y\ a x^2 y^3$$

$$\mathcal{D}_1\ \lambda(x,y)\ a x^2 y^3 \qquad \mathcal{D}_2\ \lambda(x,y)\ a x^2 y^3$$

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

Partial Derivatives

$$\frac{\partial a x^2 y^3}{\partial x} \qquad \frac{\partial a x^2 y^3}{\partial y}$$

$$\mathcal{D}\ \lambda x\ a x^2 y^3 \qquad \mathcal{D}\ \lambda y\ a x^2 y^3$$

$$\mathcal{D}_1\ \lambda(x,y)\ a x^2 y^3 \qquad \mathcal{D}_2\ \lambda(x,y)\ a x^2 y^3$$

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

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

Partial Derivatives

$$\frac{\partial a x^2 y^3}{\partial x} \qquad \frac{\partial a x^2 y^3}{\partial y}$$

$$\mathcal{D}\ \lambda x\ a x^2 y^3 \qquad \mathcal{D}\ \lambda y\ a x^2 y^3$$

$$\mathcal{D}_1\ \lambda(x,y)\ a x^2 y^3 \qquad \mathcal{D}_2\ \lambda(x,y)\ a x^2 y^3$$

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

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

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

Gradients

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

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

Jacobians

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

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

$$(\mathcal{J}\ f\ \mathbf{x})[i,j] = (\nabla\ (\mathbf{f}[i]))[j]$$

$$\mathcal{J} : (\mathbb{R}^n \to \mathbb{R}^m) \to (\mathbb{R}^n \to \mathbb{R}^{m \times n})$$

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

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

The Chain Rule

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

$$\frac{\mathrm{d}g}{\mathrm{d}x} = \frac{\mathrm{d}g}{\mathrm{d}f}\frac{\mathrm{d}f}{\mathrm{d}x}$$

The Chain Rule

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

$$\frac{\mathrm{d}g}{\mathrm{d}x} = \frac{\mathrm{d}g}{\mathrm{d}f}\frac{\mathrm{d}f}{\mathrm{d}x}$$

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

The Chain Rule

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

$$\frac{\mathrm{d}g}{\mathrm{d}x} = \frac{\mathrm{d}g}{\mathrm{d}f}\frac{\mathrm{d}f}{\mathrm{d}x}$$

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

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

Outline

Outline

Straight-Line Code and Jacobians

$$\mathbf{x}_1 = f_1\ \mathbf{x}_0$$
$$\vdots$$
$$\mathbf{x}_n = f_n\ \mathbf{x}_{n-1}$$

$$f = f_1 \circ \cdots \circ f_n$$

$$\mathcal{J}\ f\ \mathbf{x}_0 = (\mathcal{J}\ f_n\ \mathbf{x}_{n-1}) \times \cdots \times (\mathcal{J}\ f_1\ \mathbf{x}_0)$$
$$(\mathcal{J}\ f\ \mathbf{x}_0)^\top = (\mathcal{J}\ f_1\ \mathbf{x}_0)^\top \times \cdots \times (\mathcal{J}\ f_n\ \mathbf{x}_{n-1})^\top$$

One Way to Compute the Jacobian

$$\overrightarrow{\mathbf{X}_1} = (\mathcal{J}\ f_1\ \mathbf{x}_0)$$
$$\overrightarrow{\mathbf{X}_2} = (\mathcal{J}\ f_2\ \mathbf{x}_1) \times \overrightarrow{\mathbf{X}_1}$$
$$\vdots$$
$$\overrightarrow{\mathbf{X}_n} = (\mathcal{J}\ f_n\ \mathbf{x}_{n-1}) \times \overrightarrow{\mathbf{X}_{n-1}}$$

$$\overrightarrow{\mathbf{X}_n} = \mathcal{J}\ f\ \mathbf{x}_0$$

Forward-Mode AD

$$\overline{\mathbf{x}_1'} = (\mathcal{J}\ f_1\ \mathbf{x}_0) \times \overline{\mathbf{x}_0'}$$

$$\vdots$$

$$\overline{\mathbf{x}_n'} = (\mathcal{J}\ f_n\ \mathbf{x}_{n-1}) \times \overline{\mathbf{x}_{n-1}'}$$

$$\overline{\mathbf{x}_n'} = (\mathcal{J}\ f\ \mathbf{x}_0) \times \overline{\mathbf{x}_0'}$$

Wengert, R. E. (1964). A simple automatic derivative evaluation program. *Communications of the ACM*, **7**(8):463–4.

Interleaving Forward Mode

$$\begin{aligned}
\mathbf{x}_1 &= f_1\ \mathbf{x}_0 & \overline{\mathbf{x}_1'} &= (\mathcal{J}\ f_1\ \mathbf{x}_0) \times \overline{\mathbf{x}_0'} \\
&\vdots & &\vdots \\
\mathbf{x}_n &= f_n\ \mathbf{x}_{n-1} & \overline{\mathbf{x}_n'} &= (\mathcal{J}\ f_n\ \mathbf{x}_{n-1}) \times \overline{\mathbf{x}_{n-1}'}
\end{aligned}$$

$$\begin{aligned}
\mathbf{x}_1 &= f_1\ \mathbf{x}_0 \\
\overline{\mathbf{x}_1'} &= (\mathcal{J}\ f_1\ \mathbf{x}_0) \times \overline{\mathbf{x}_0'} \\
&\vdots \\
\mathbf{x}_n &= f_n\ \mathbf{x}_{n-1} \\
\overline{\mathbf{x}_n'} &= (\mathcal{J}\ f_n\ \mathbf{x}_{n-1}) \times \overline{\mathbf{x}_{n-1}'}
\end{aligned}$$

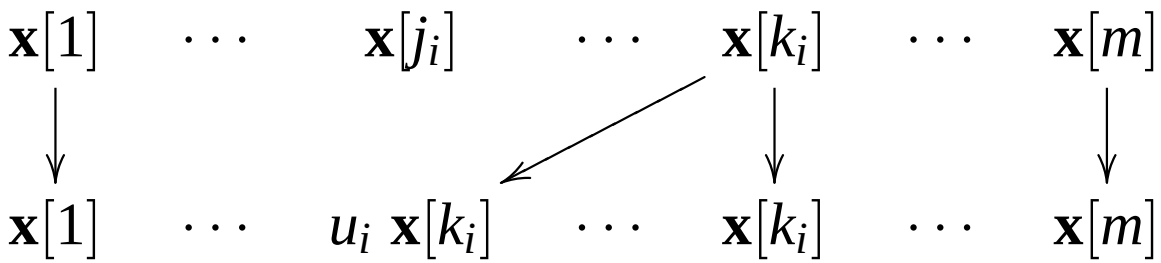
Forward Mode as a Transformation

$$\left.\begin{array}{l} \mathbf{x}_1 = f_1\ \mathbf{x}_0 \\ \quad\vdots \\ \mathbf{x}_n = f_n\ \mathbf{x}_{n-1} \end{array}\right\} \rightsquigarrow \left\{\begin{array}{l} \overrightarrow{\mathbf{x}_1} = \overrightarrow{f_1}\ \overrightarrow{\mathbf{x}_0} \\ \quad\vdots \\ \overrightarrow{\mathbf{x}_n} = \overrightarrow{f_n}\ \overrightarrow{\mathbf{x}_{n-1}} \end{array}\right.$$

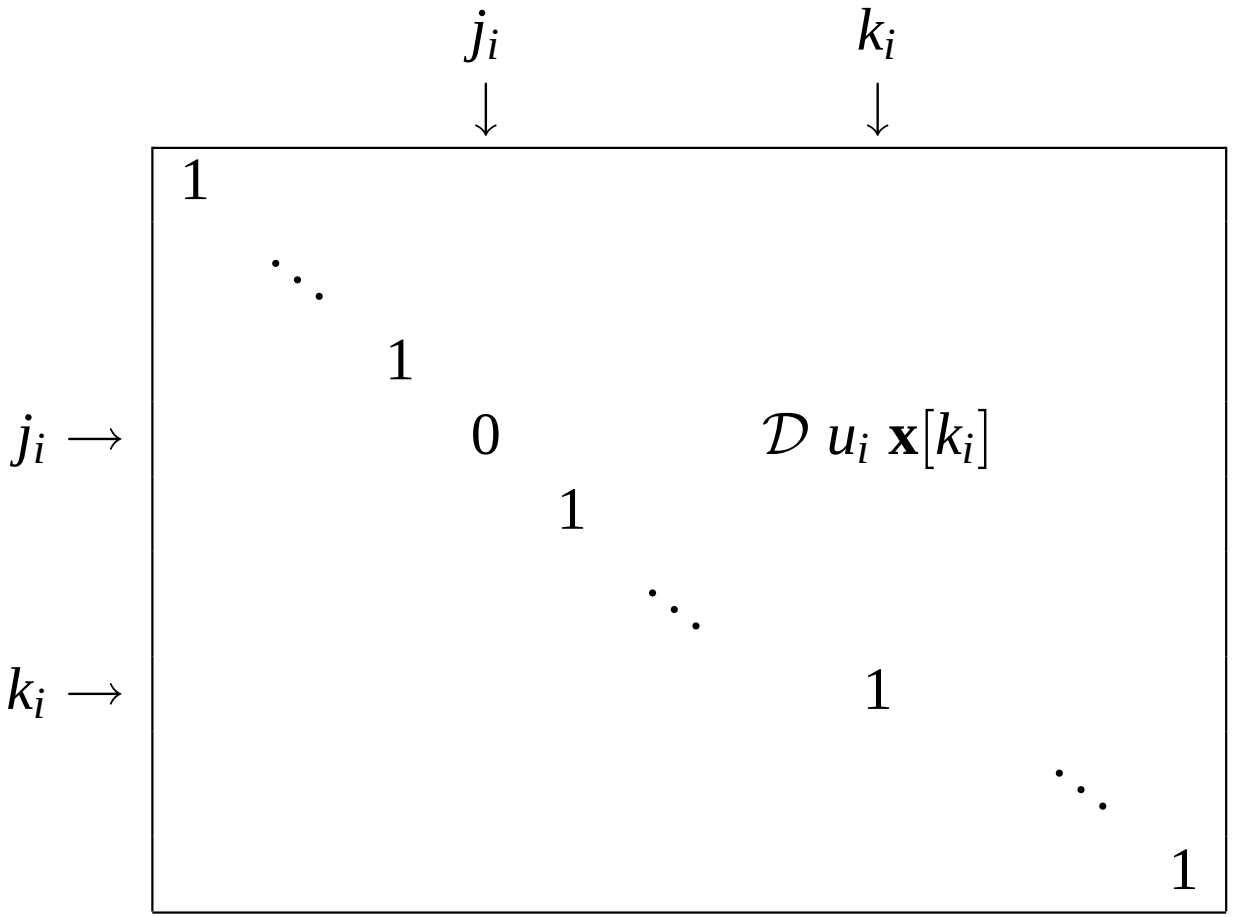
$$\begin{aligned} \overrightarrow{\mathbf{x}} &= (\mathbf{x}, \overline{\mathbf{x}'}) \\ \overrightarrow{f}\ (\mathbf{x}, \overline{\mathbf{x}'}) &= ((f\ \mathbf{x}), ((\mathcal{J}\ f\ \mathbf{x}) \times \overline{\mathbf{x}'})) \end{aligned}$$

A Unary Sparse Function

$$(f_i\ \mathbf{x})[j_i] = u_i\ \mathbf{x}[k_i]$$
$$(f_i\ \mathbf{x})[j'] = \mathbf{x}[j'] \qquad j' \neq j_i$$



The Jacobian of a Unary Sparse Function

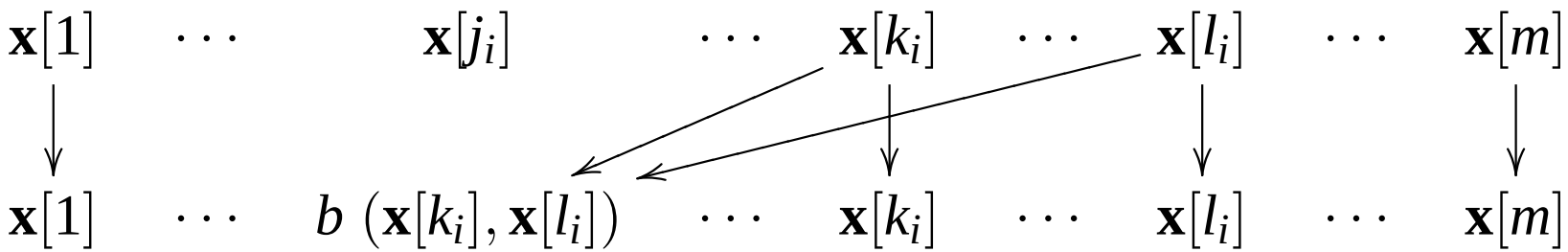


Computing $(\mathcal{J}\ f_i\ \mathbf{x}_{i-1}) \times \overline{\mathbf{x}_{i-1}}$ for a Unary Sparse Function

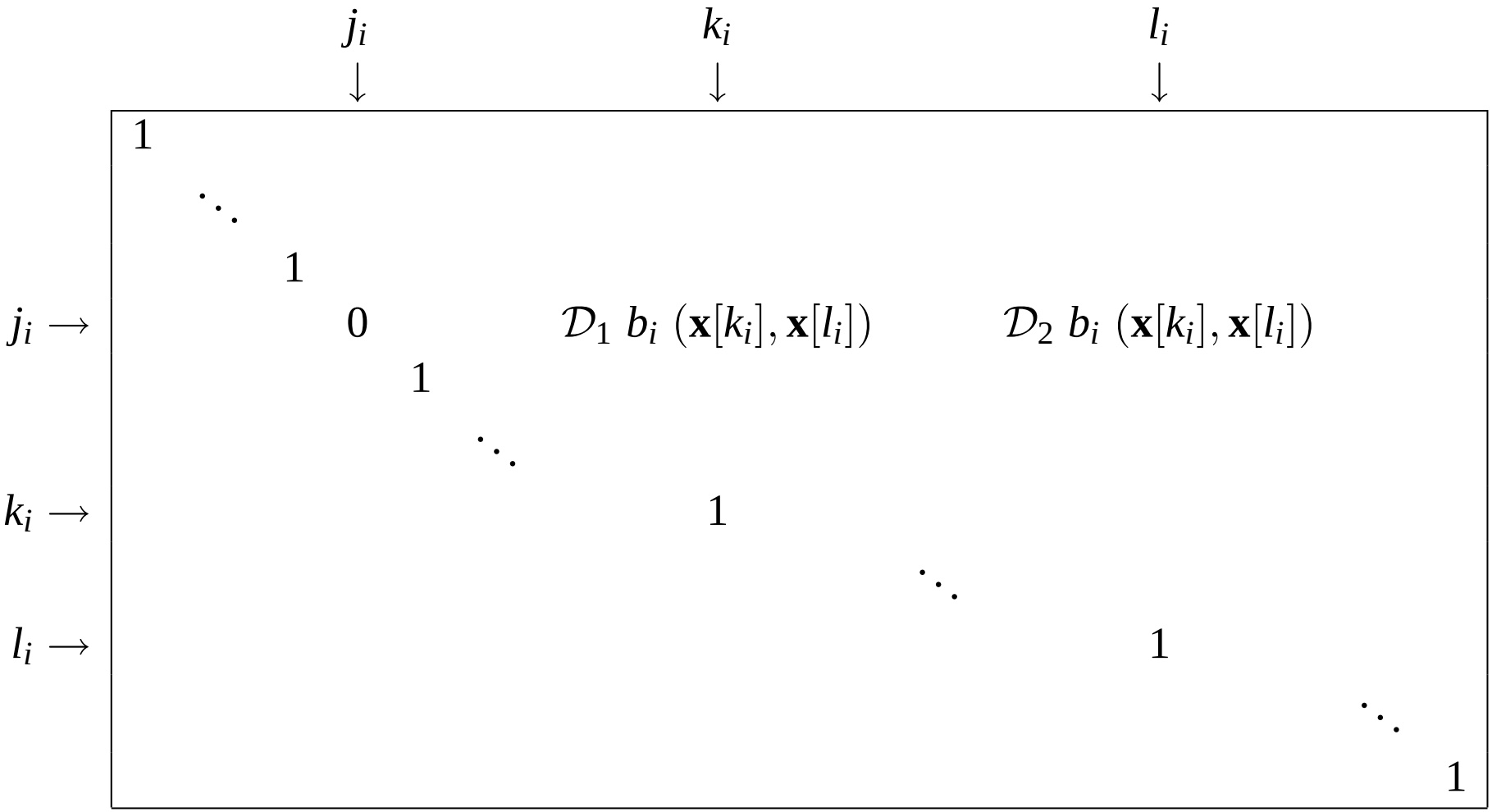
$$\begin{pmatrix} \overline{\mathbf{x}}[1] \\ \vdots \\ \overline{\mathbf{x}}[j_i - 1] \\ (\mathcal{D}\ u_i\ \mathbf{x}[k_i]) \times \overline{\mathbf{x}}[k_i] \\ \overline{\mathbf{x}}[j_i + 1] \\ \vdots \\ \overline{\mathbf{x}}[k_i] \\ \vdots \\ \overline{\mathbf{x}}[m] \end{pmatrix} = \begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & 0 & & & \mathcal{D}\ u_i\ \mathbf{x}[k_i] & & \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & & & & 1 & & \\ & & & & & & & \ddots & \\ & & & & & & & & 1 \end{pmatrix} \begin{pmatrix} \overline{\mathbf{x}}[1] \\ \vdots \\ \overline{\mathbf{x}}[j_i - 1] \\ \overline{\mathbf{x}}[j_i] \\ \overline{\mathbf{x}}[j_i + 1] \\ \vdots \\ \overline{\mathbf{x}}[k_i] \\ \vdots \\ \overline{\mathbf{x}}[m] \end{pmatrix}$$

A Binary Sparse Function

$$(f_i\ \mathbf{x})[j_i] = b_i\ (\mathbf{x}[k_i], \mathbf{x}[l_i])$$
$$(f_i\ \mathbf{x})[j'] = \mathbf{x}[j'] \qquad j' \neq j_i$$



The Jacobian of a Binary Sparse Function



Computing $(\mathcal{J}\ f_i\ \mathbf{x}_{i-1}) \times \overline{\mathbf{x}_{i-1}}$ for a Binary Sparse Function

$$\begin{pmatrix} \overline{\mathbf{x}}[1] \\ \vdots \\ \overline{\mathbf{x}}[j_i - 1] \\ ((\mathcal{D}_1\ b_i\ (\mathbf{x}[k_i], \mathbf{x}[l_i])) \times \overline{\mathbf{x}}[k_i]) + ((\mathcal{D}_2\ b_i\ (\mathbf{x}[k_i], \mathbf{x}[l_i])) \times \overline{\mathbf{x}}[l_i]) \\ \overline{\mathbf{x}}[j_i + 1] \\ \vdots \\ \overline{\mathbf{x}}[k_i] \\ \vdots \\ \overline{\mathbf{x}}[l_i] \\ \vdots \\ \overline{\mathbf{x}}[m] \end{pmatrix} = \begin{pmatrix} 1 & & & & & & & & & & \\ & \ddots & & & & & & & & & \\ & & 1 & & & & & & & & \\ & & & 0 & & \mathcal{D}_1\ b_i\ (\mathbf{x}[k_i], \mathbf{x}[l_i]) & & \mathcal{D}_2\ b_i\ (\mathbf{x}[k_i], \mathbf{x}[l_i]) & & & \\ & & & & 1 & & & & & & \\ & & & & & \ddots & & & & & \\ & & & & & & 1 & & & & \\ & & & & & & & \ddots & & & \\ & & & & & & & & 1 & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & 1 \end{pmatrix} \begin{pmatrix} \overline{\mathbf{x}}[1] \\ \vdots \\ \overline{\mathbf{x}}[j_i - 1] \\ \overline{\mathbf{x}}[j_i] \\ \overline{\mathbf{x}}[j_i + 1] \\ \vdots \\ \overline{\mathbf{x}}[k_i] \\ \vdots \\ \overline{\mathbf{x}}[l_i] \\ \vdots \\ \overline{\mathbf{x}}[m] \end{pmatrix}$$

Forward Mode as a Sparse Transformation

$$x_{j_i} := u_i\ x_{k_i} \quad \rightsquigarrow \quad \overrightarrow{x_{j_i}} := \overrightarrow{u_i}\ \overrightarrow{x_{k_i}}$$
$$x_{j_i} := b_i\ (x_{k_i}, x_{l_i}) \quad \rightsquigarrow \quad \overrightarrow{x_{j_i}} := \overrightarrow{b_i}\ (\overrightarrow{x_{k_i}}, \overrightarrow{x_{l_i}})$$

$$\overrightarrow{x} = (x, \overline{x'})$$
$$\overrightarrow{u}\ (x, \overline{x'}) = ((u\ x), ((\mathcal{D}\ u\ x) \times \overline{x'}))$$
$$\overrightarrow{b}\ ((x_1, \overline{x_1'}), (x_2, \overline{x_2'})) = ((b\ (x_1, x_2)), (((\mathcal{D}_1\ b\ (x_1, x_2)) \times \overline{x_1'}) + ((\mathcal{D}_2\ b\ (x_1, x_2)) \times \overline{x_2'})))$$

Outline

Straight-Line Code and Jacobians

$$\mathbf{x}_1 = f_1\ \mathbf{x}_0$$
$$\vdots$$
$$\mathbf{x}_n = f_n\ \mathbf{x}_{n-1}$$

$$f = f_1 \circ \cdots \circ f_n$$

$$\mathcal{J}\ f\ \mathbf{x}_0 = (\mathcal{J}\ f_n\ \mathbf{x}_{n-1}) \times \cdots \times (\mathcal{J}\ f_1\ \mathbf{x}_0)$$
$$(\mathcal{J}\ f\ \mathbf{x}_0)^\top = (\mathcal{J}\ f_1\ \mathbf{x}_0)^\top \times \cdots \times (\mathcal{J}\ f_n\ \mathbf{x}_{n-1})^\top$$

Another Way to Compute the Jacobian

$$\overline{\mathbf{X}_{n-1}} = (\mathcal{J}\ f_n\ \mathbf{x}_{n-1})^\top$$
$$\overline{\mathbf{X}_{n-2}} = (\mathcal{J}\ f_{n-1}\ \mathbf{x}_{n-2})^\top \times \overline{\mathbf{X}_{n-1}}$$
$$\vdots$$
$$\overline{\mathbf{X}_0} = (\mathcal{J}\ f_1\ \mathbf{x}_0)^\top \times \overline{\mathbf{X}_1}$$

$$\overline{\mathbf{X}_0} = (\mathcal{J}\ f\ \mathbf{x}_0)^\top$$

Reverse-Mode AD

$$\overline{\mathbf{x}_{n-1}} = (\mathcal{J}\ f_n\ \mathbf{x}_{n-1})^\top \times \overline{\mathbf{x}_n}$$

$$\vdots$$

$$\overline{\mathbf{x}_0} = (\mathcal{J}\ f_1\ \mathbf{x}_0)^\top \times \overline{\mathbf{x}_1}$$

$$\overline{\mathbf{x}_0} = (\mathcal{J}\ f\ \mathbf{x}_0)^\top \times \overline{\mathbf{x}_n}$$

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

Reverse Mode Cannot be Interleaved

$$
\begin{aligned}
\mathbf{x}_1 &= f_1\ \mathbf{x}_0 \\
&\vdots \\
\mathbf{x}_n &= f_n\ \mathbf{x}_{n-1} \\
\overline{\mathbf{x}_{n-1}} &= (\mathcal{J}\ f_n\ \mathbf{x}_{n-1})^\top \times \overline{\mathbf{x}_n} \\
&\vdots \\
\overline{\mathbf{x}_0} &= (\mathcal{J}\ f_1\ \mathbf{x}_0)^\top \times \overline{\mathbf{x}_1}
\end{aligned}
$$

Reverse Mode via Backpropagators

$$\mathbf{x}_1 = f_1\ \mathbf{x}_0$$
$$\overline{\mathbf{x}_1} = \lambda \overline{\mathbf{x}}\ \overline{\mathbf{x}_0}\ ((\mathcal{J}\ f_1\ \mathbf{x}_0)^\top \times \overline{\mathbf{x}})$$
$$\vdots$$
$$\mathbf{x}_n = f_n\ \mathbf{x}_{n-1}$$
$$\overline{\mathbf{x}_n} = \lambda \overline{\mathbf{x}}\ \overline{\mathbf{x}_{n-1}}\ ((\mathcal{J}\ f_n\ \mathbf{x}_{n-1})^\top \times \overline{\mathbf{x}})$$

$$\overline{\mathbf{x}_n}\ \overline{\mathbf{x}_n}$$

Reverse Mode as a Transformation

$$\left.\begin{array}{l} \mathbf{x}_1 = f_1\ \mathbf{x}_0 \\ \quad\vdots \\ \mathbf{x}_n = f_n\ \mathbf{x}_{n-1} \end{array}\right\} \rightsquigarrow \left\{\begin{array}{l} \overleftarrow{\mathbf{x}_1} = \overleftarrow{f_1}\ \overleftarrow{\mathbf{x}_0} \\ \quad\vdots \\ \overleftarrow{\mathbf{x}_n} = \overleftarrow{f_n}\ \overleftarrow{\mathbf{x}_{n-1}} \end{array}\right.$$

$$\begin{aligned} \overleftarrow{\mathbf{x}} &= (\mathbf{x}, \overline{\mathbf{x}}) \\ \overleftarrow{f}\ (\mathbf{x}, \overline{\mathbf{x}}) &= ((f\ \mathbf{x}), (\lambda \overline{\mathbf{x}}\ \overline{\mathbf{x}}\ ((\mathcal{J}\ f\ \mathbf{x})^{\top} \times \overline{\mathbf{x}}))) \end{aligned}$$

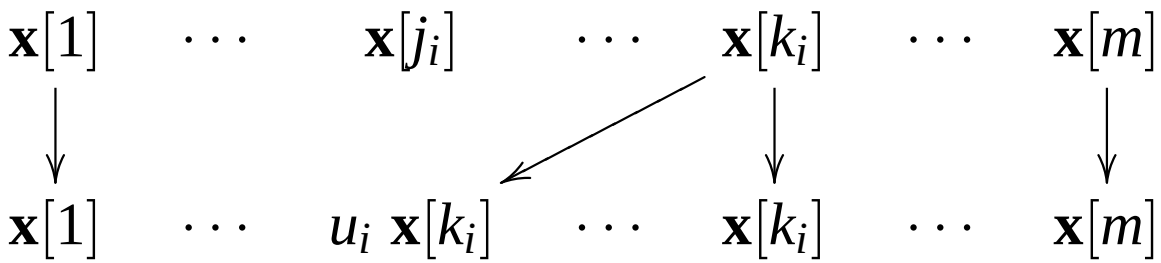
Reverse Mode via a Tape

$$\left.\begin{array}{l} \mathbf{x}_1 = f_1\ \mathbf{x}_0 \\ \quad\vdots \\ \mathbf{x}_n = f_n\ \mathbf{x}_{n-1} \end{array}\right\} \rightsquigarrow \left\{\begin{array}{l} \overleftarrow{\mathbf{x}_1} = \overleftarrow{f_1}\ \overleftarrow{\mathbf{x}_0} \\ \quad\vdots \\ \overleftarrow{\mathbf{x}_n} = \overleftarrow{f_n}\ \overleftarrow{\mathbf{x}_{n-1}} \end{array}\right.$$

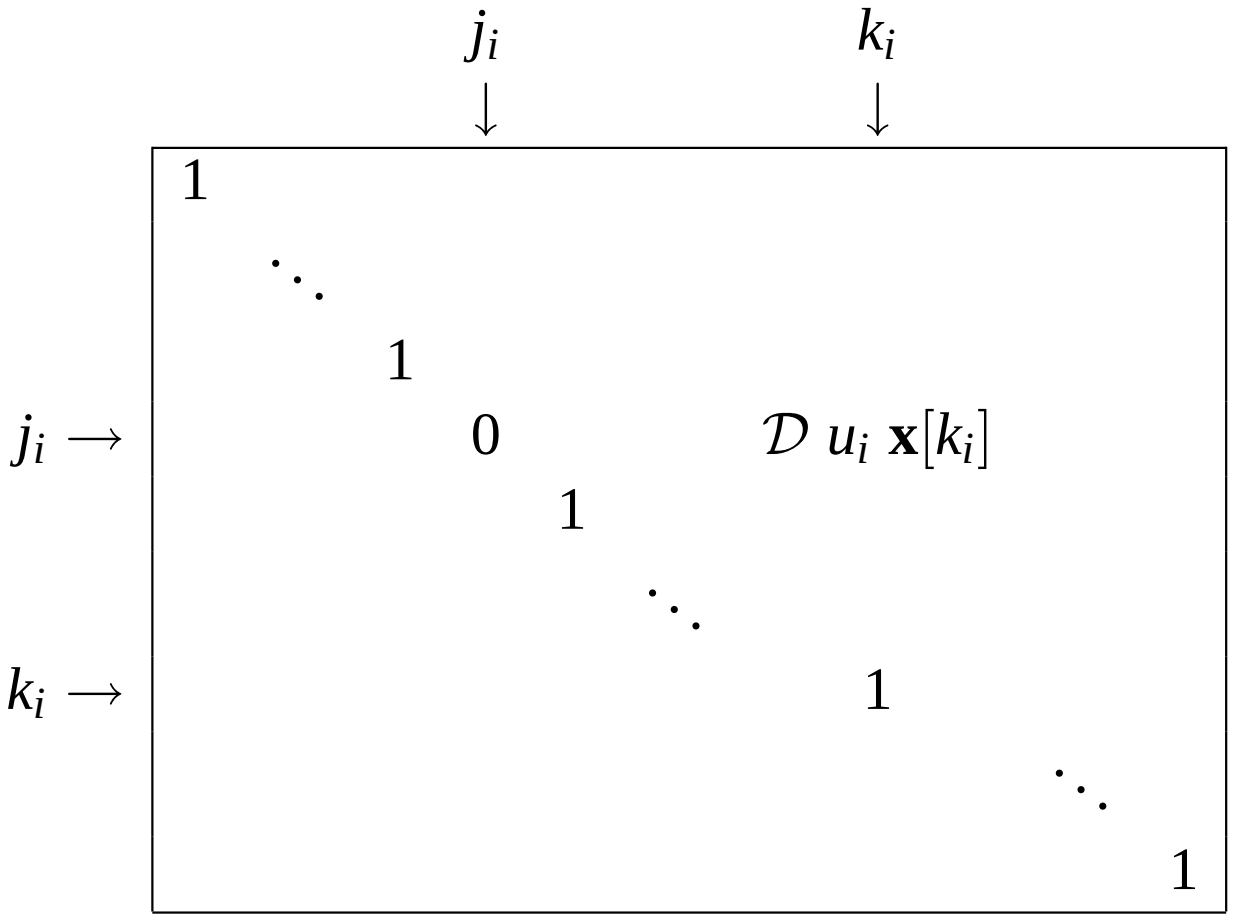
$$\begin{array}{rcl} \overleftarrow{\mathbf{x}} & = & \mathbf{x} \\ \overleftarrow{f}\ \mathbf{x} & = & \textbf{begin } \overline{\mathbf{x}} := \lambda \overline{\mathbf{x}}\ \overline{\mathbf{x}}\ ((\mathcal{J}\ f\ \mathbf{x})^{\top} \times \overline{\mathbf{x}}); \\ & & (f\ \mathbf{x})\ \textbf{end} \end{array}$$

A Unary Sparse Function

$$(f_i\ \mathbf{x})[j_i] = u_i\ \mathbf{x}[k_i]$$
$$(f_i\ \mathbf{x})[j'] = \mathbf{x}[j'] \qquad j' \neq j_i$$



The Jacobian of a Unary Sparse Function



The Transpose of the Jacobian of a Unary Sparse Function

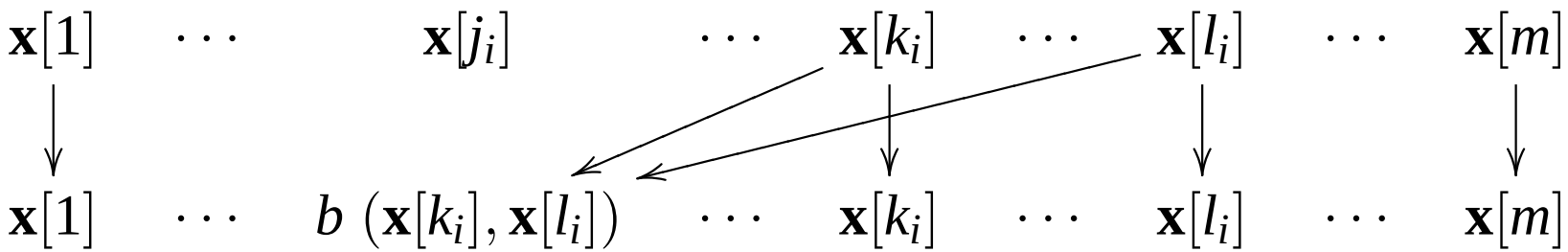
$$\begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & 0 & & \mathcal{D}\ u_i\ \mathbf{x}[k_i] & & \\ & & & & 1 & & & \\ & & & & & \ddots & & \\ & & & & & & 1 & \\ & & & & & & & \ddots \\ & & & & & & & & 1 \end{pmatrix}^{\top} = \begin{pmatrix} 1 & & & & & & & \\ & \ddots & & & & & & \\ & & 1 & & & & & \\ & & & 0 & & & & \\ & & & & 1 & & & \\ & & & & & \ddots & & \\ & & & \mathcal{D}\ u_i\ \mathbf{x}[k_i] & & & 1 & \\ & & & & & & & \ddots \\ & & & & & & & & 1 \end{pmatrix}$$

Computing $(\mathcal{J}\ f_i\ \mathbf{x}_{i-1})^\top \times \overline{\mathbf{x}_i}$ for a Unary Sparse Function

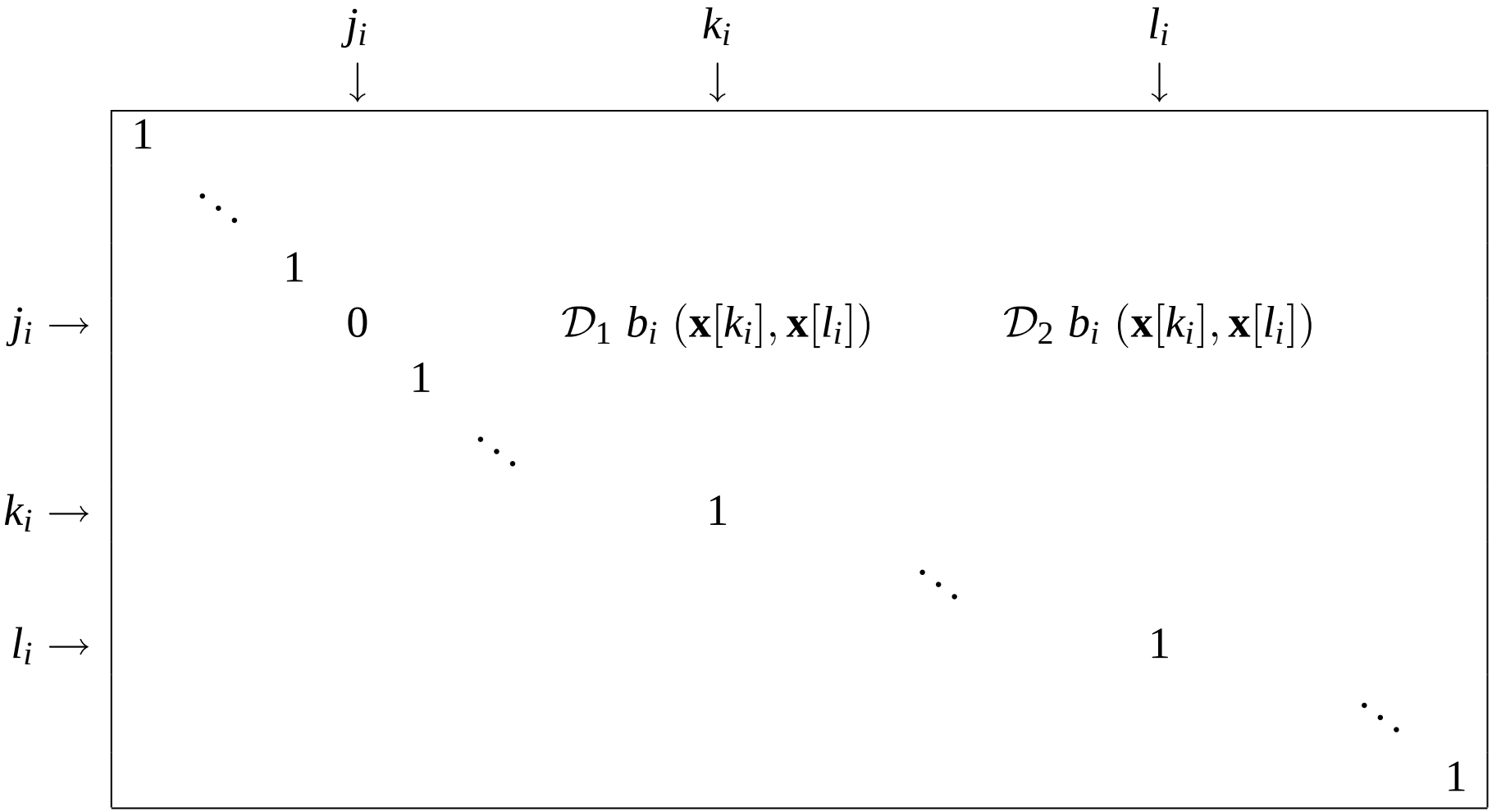
$$\begin{pmatrix} \overline{\mathbf{x}}[1] \\ \vdots \\ \overline{\mathbf{x}}[j_i - 1] \\ 0 \\ \overline{\mathbf{x}}[j_i + 1] \\ \vdots \\ ((\mathcal{D}\ u_i\ \mathbf{x}[k_i]) \times \overline{\mathbf{x}}[j_i]) + \overline{\mathbf{x}}[k_i] \\ \vdots \\ \overline{\mathbf{x}}[m] \end{pmatrix} = \begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & 0 & & & & & \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & \mathcal{D}\ u_i\ \mathbf{x}[k_i] & & & 1 & & \\ & & & & & & & \ddots & \\ & & & & & & & & 1 \end{pmatrix} \begin{pmatrix} \overline{\mathbf{x}}[1] \\ \vdots \\ \overline{\mathbf{x}}[j_i - 1] \\ \overline{\mathbf{x}}[j_i] \\ \overline{\mathbf{x}}[j_i + 1] \\ \vdots \\ \overline{\mathbf{x}}[k_i] \\ \vdots \\ \overline{\mathbf{x}}[m] \end{pmatrix}$$

A Binary Sparse Function

$$(f_i\ \mathbf{x})[j_i] = b_i\ (\mathbf{x}[k_i], \mathbf{x}[l_i])$$
$$(f_i\ \mathbf{x})[j'] = \mathbf{x}[j'] \qquad j' \neq j_i$$



The Jacobian of a Binary Sparse Function



The Transpose of the Jacobian of a Binary Sparse Function

$$\begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & 0 & \mathcal{D}_1\, b_i\, (\mathbf{x}[k_i], \mathbf{x}[l_i]) & \mathcal{D}_2\, b_i\, (\mathbf{x}[k_i], \mathbf{x}[l_i]) & & & \\ & & & 1 & & & & & \\ & & & & \ddots & & & & \\ & & & & & 1 & & & \\ & & & & & & \ddots & & \\ & & & & & & & 1 & \\ & & & & & & & & \ddots \\ & & & & & & & & & 1 \end{pmatrix}^{\top} = \begin{pmatrix} 1 & & & & & & & & \\ & \ddots & & & & & & & \\ & & 1 & & & & & & \\ & & & 0 & & & & & \\ & & & & 1 & & & & \\ & & & & & \ddots & & & \\ & & & \mathcal{D}_1\, b_i\, (\mathbf{x}[k_i], \mathbf{x}[l_i]) & & & 1 & & \\ & & & & & & & \ddots & \\ & & & \mathcal{D}_2\, b_i\, (\mathbf{x}[k_i], \mathbf{x}[l_i]) & & & & & 1 \\ & & & & & & & & \ddots \\ & & & & & & & & & 1 \end{pmatrix}$$

Computing $(\mathcal{J}\ f_i\ \mathbf{x}_{i-1})^\top \times \overleftarrow{\mathbf{x}_i}$ for a Binary Sparse Function

$$\begin{pmatrix} \overleftarrow{\mathbf{x}}[1] \\ \vdots \\ \overleftarrow{\mathbf{x}}[j_i - 1] \\ 0 \\ \overleftarrow{\mathbf{x}}[j_i + 1] \\ \vdots \\ ((\mathcal{D}_1\ b_i\ (\mathbf{x}[k_i], \mathbf{x}[l_i])) \times \overleftarrow{\mathbf{x}}[j_i]) + \overleftarrow{\mathbf{x}}[k_i] \\ \vdots \\ ((\mathcal{D}_2\ b_i\ (\mathbf{x}[k_i], \mathbf{x}[l_i])) \times \overleftarrow{\mathbf{x}}[j_i]) + \overleftarrow{\mathbf{x}}[l_i] \\ \vdots \\ \overleftarrow{\mathbf{x}}[m] \end{pmatrix} = \begin{pmatrix} 1 & & & & & & & & & & \\ & \ddots & & & & & & & & & \\ & & 1 & & & & & & & & \\ & & & 0 & & & & & & & \\ & & & & 1 & & & & & & \\ & & & & & \ddots & & & & & \\ & & & \mathcal{D}_1\ b_i\ (\mathbf{x}[k_i], \mathbf{x}[l_i]) & & & 1 & & & & \\ & & & & & & & \ddots & & & \\ & & & \mathcal{D}_2\ b_i\ (\mathbf{x}[k_i], \mathbf{x}[l_i]) & & & & & 1 & & \\ & & & & & & & & & \ddots & \\ & & & & & & & & & & 1 \end{pmatrix} \begin{pmatrix} \overleftarrow{\mathbf{x}}[1] \\ \vdots \\ \overleftarrow{\mathbf{x}}[j_i - 1] \\ \overleftarrow{\mathbf{x}}[j_i] \\ \overleftarrow{\mathbf{x}}[j_i + 1] \\ \vdots \\ \overleftarrow{\mathbf{x}}[k_i] \\ \vdots \\ \overleftarrow{\mathbf{x}}[l_i] \\ \vdots \\ \overleftarrow{\mathbf{x}}[m] \end{pmatrix}$$

Sparse Reverse Mode via a Tape

$$x_{j_i} := u_i\ x_{k_i} \quad \rightsquigarrow \quad \begin{array}{l} \bar{x} := \lambda[\,]\ \textbf{begin}\ \overline{x_{k_i}} +:= (\mathcal{D}\ u_i\ x_{k_i}) \times \overline{x_{j_i}}; \\ \qquad\qquad\qquad\qquad\quad \overline{x_{j_i}} := 0; \\ \qquad\qquad\qquad\qquad\quad \bar{x}\ [\,]\ \textbf{end}; \\ x_{j_i} := u_i\ x_{k_i} \end{array}$$

$$x_{j_i} := b_i\ (x_{k_i}, x_{l_i}) \quad \rightsquigarrow \quad \begin{array}{l} \bar{x} := \lambda[\,]\ \textbf{begin}\ \overline{x_{k_i}} +:= (\mathcal{D}_1\ b_i\ (x_{k_i}, x_{l_i})) \times \overline{x_{j_i}}; \\ \qquad\qquad\qquad\qquad\quad \overline{x_{l_i}} +:= (\mathcal{D}_2\ b_i\ (x_{k_i}, x_{l_i})) \times \overline{x_{j_i}}; \\ \qquad\qquad\qquad\qquad\quad \overline{x_{j_i}} := 0; \\ \qquad\qquad\qquad\qquad\quad \bar{x}\ [\,]\ \textbf{end}; \\ x_{j_i} := b_i\ (x_{k_i}, x_{l_i}) \end{array}$$

Traditional AD

Forward Mode: $\mathbb{R}^n \to \mathbb{R}^m \rightsquigarrow (\mathbb{R}^n \times \mathbb{R}^n) \to (\mathbb{R}^m \times \mathbb{R}^m)$

Reverse Mode: $\mathbb{R}^n \to \mathbb{R}^m \rightsquigarrow \mathbb{R}^n \to (\mathbb{R}^m \times (\mathbb{R}^m \to \mathbb{R}^n))$

Functional AD

$$
\begin{aligned}
\overline{\mathbf{null}} &= \mathbf{null} \\
\overline{\mathbb{R}} &= \mathbb{R} \\
\overline{\tau_1 \times \tau_2} &= \overline{\tau_1} \times \overline{\tau_2} \\
\overline{\tau_1 \xrightarrow{\tau_1',\ldots,\tau_n'} \tau_2} &= \overline{\tau_1'} \times \cdots \times \overline{\tau_n'}
\end{aligned}
$$

$$
\begin{aligned}
\overrightarrow{\mathbf{null}} &= \mathbf{null} \times \overline{\mathbf{null}} \\
\overrightarrow{\mathbb{R}} &= \mathbb{R} \times \overline{\mathbb{R}} \\
\overrightarrow{\tau_1 \times \tau_2} &= \overrightarrow{\tau_1} \times \overrightarrow{\tau_2} \\
\overrightarrow{\tau_1 \xrightarrow{\tau_1',\ldots,\tau_n'} \tau_2} &= \overrightarrow{\tau_1} \xrightarrow{\overrightarrow{\tau_1'},\ldots,\overrightarrow{\tau_n'}} \overrightarrow{\tau_2}
\end{aligned}
$$

$$
\begin{aligned}
\overleftarrow{\mathbf{null}} &= \mathbf{null} \\
\overleftarrow{\mathbb{R}} &= \mathbb{R} \\
\overleftarrow{\tau_1 \times \tau_2} &= \overleftarrow{\tau_1} \times \overleftarrow{\tau_2} \\
\overleftarrow{\tau_1 \xrightarrow{\tau_1',\ldots,\tau_n'} \tau_2} &= \\
&\overleftarrow{\tau_1} \xrightarrow{\overleftarrow{\tau_1'},\ldots,\overleftarrow{\tau_n'}} (\overleftarrow{\tau_2} \times (\overleftarrow{\tau_2} \rightarrow (\overleftarrow{\tau_1'} \times \cdots \times \overleftarrow{\tau_n'}) \times \overleftarrow{\tau_1}))
\end{aligned}
$$

Forward Mode: $\overrightarrow{\mathcal{J}} : \tau \rightarrow \overrightarrow{\tau}$

Reverse Mode: $\overleftarrow{\mathcal{J}} : \tau \rightarrow \overleftarrow{\tau}$

Outline

Derivatives

$$\mathcal{D}\, f\, x \triangleq \textsc{Tangent}\ ((\overrightarrow{\mathcal{J}}\ f)\ (x \blacktriangleright 1))$$

$$\mathcal{D}\, f\, x \triangleq \textsc{Cdr}\ ((\textsc{Cdr}\ ((\overleftarrow{\mathcal{J}}\ f)\ (\overleftarrow{\mathcal{J}}\ x)))\ 1)$$

Roots using Newton-Raphson

$$\textsc{Root}\ (f, x_0, \epsilon) \stackrel{\triangle}{=} \textbf{let}\ x' \stackrel{\triangle}{=} x_0 - \frac{f\ x_0}{\mathcal{D}\ f\ x_0}$$
$$\textbf{in if}\ |x_0 - x'| \leq \epsilon\ \textbf{then}\ x_0\ \textbf{else}\ \textsc{Root}\ (f, x', \epsilon)$$

Univariate Minimizer

Line Search

$$\textsc{LineSearch}\ (f, x_0, \epsilon) \stackrel{\triangle}{=} \textsc{Root}\ ((\mathcal{D}\ f), x_0, \epsilon)$$

Gradients

$$\nabla f\ x \stackrel{\triangle}{=} \textbf{let}\ n \stackrel{\triangle}{=} \textsc{Length}\ x\ \textbf{in}\ \textsc{Map}\ ((\lambda i\ \textsc{Tangent}\ ((\overrightarrow{\mathcal{J}}\ f)\ (x \blacktriangleright e_{i,n}))), (\iota\ n))$$

$$\nabla f\ x \stackrel{\triangle}{=} \textsc{Cdr}\ ((\textsc{Cdr}\ ((\overleftarrow{\mathcal{J}}\ f)\ (\overleftarrow{\mathcal{J}}\ x)))\ 1)$$

Multivariate Minimizer

Gradient Descent

$$\begin{array}{l}\textsc{GradientDescent}\ (f, x_0, \epsilon) \stackrel{\triangle}{=} \\ \quad \textbf{let}\ g \stackrel{\triangle}{=} \nabla f\ x_0 \\ \quad \textbf{in if}\ \|g\| \leq \epsilon \\ \qquad \textbf{then}\ x_0 \\ \qquad \textbf{else}\ \textsc{GradientDescent} \\ \qquad\quad (f, (x_0 + ((\textsc{LineSearch}\ ((\lambda k\ f\ (x_0 + (k \times g))), \epsilon)) \times g)), \epsilon)\end{array}$$

Saddle Points

Continuous Two-Person Zero Sum Games

$$\mathbf{x} : \mathbb{R}^m$$
$$\mathbf{y} : \mathbb{R}^n$$
$$\textsc{Payoff} : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}$$
$$\min_{\mathbf{x}} \max_{\mathbf{y}} \textsc{Payoff}\ (\mathbf{x}, \mathbf{y})$$

$$(\mathbf{x}^*, \mathbf{y}^*) = \textbf{let}\ \mathbf{x}^* \triangleq \textsc{GradientDescent}\ ((\lambda \mathbf{x}\ \textsc{GradientDescent}\ ((\lambda \mathbf{y}\ (-\textsc{Payoff}\ (\mathbf{x}, \mathbf{y}))), \mathbf{y}_0, \epsilon)), \mathbf{x}_0, \epsilon)\ \textbf{in}\ (\mathbf{x}^*, (\textsc{GradientDescent}\ ((\lambda \mathbf{y}\ (-\textsc{Payoff}\ (\mathbf{x}^*, \mathbf{y}))), \mathbf{y}_0, \epsilon)))$$

von Neumann, J. and Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press, Princeton, NJ.

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

Function Inversion

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

Neural Nets

$$\textsc{Neuron}\ (\mathbf{w}, \mathbf{x}) \stackrel{\triangle}{=} \textsc{Sigmoid}\ (\mathbf{w} \cdot \mathbf{x})$$

$$\textsc{NeuralNet}\ ([\mathbf{w}''; \mathbf{w}'_1; \ldots; \mathbf{w}'_m], \mathbf{x}) \stackrel{\triangle}{=}$$
$$\quad \textsc{Neuron}\ (\mathbf{w}'', [\textsc{Neuron}\ (\mathbf{w}'_1, \mathbf{x}); \ldots; \textsc{Neuron}\ (\mathbf{w}'_m, \mathbf{x})])$$

$$\textsc{Error}\ \mathbf{w} \stackrel{\triangle}{=}$$
$$\quad \|[y_1; \ldots; y_n] - [\textsc{NeuralNet}\ (\mathbf{w}, \mathbf{x}_1); \ldots; \textsc{NeuralNet}\ (\mathbf{w}, \mathbf{x}_n)]\|$$

$$\textsc{GradientDescent}\ (\textsc{Error}, \mathbf{w}_0, \epsilon)$$

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, **323**:533–6.

Supervised Machine Learning

Function Approximation

$$\textsc{Error}\ \theta \stackrel{\triangle}{=} \|[y_1; \ldots; y_n] - [f\ (\theta, \mathbf{x}_1); \ldots; f\ (\theta, \mathbf{x}_n)]\|$$

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

Maximum Likelihood Estimation

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

Fisher, R. A. (1922). On the mathematical foundations of theoretical statistics. *Philos. Trans. Roy. Soc. London Ser. A*, **222**:309–68.

Engineering Design

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

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

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - mathematical: the $\lambda\nabla$-calculus

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - mathematical: the $\lambda\nabla$-calculus
  - programming language: Voice for FP + AD

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - mathematical: the $\lambda\nabla$-calculus
  - programming language: VLAD

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - mathematical: the $\lambda\nabla$-calculus
  - programming language: VLAD
  - implementation: high-performance SCHEME + AD

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - mathematical: the $\lambda\nabla$-calculus
  - programming language: VLAD
  - implementation: STALIN + AD

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - mathematical: the $\lambda\nabla$-calculus
  - programming language: VLAD
  - implementation: STALIN $+$ gradients

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - mathematical: the $\lambda\nabla$-calculus
  - programming language: VLAD
  - implementation: STALIN $+\nabla$

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - mathematical: the $\lambda\nabla$-calculus
  - programming language: VLAD
  - implementation: STALIN∇

Work in Progress

Lambda: the ultimate calculus

Part of larger project to combine λ-calculus and AD.

- include both forward-mode and reverse-mode operators
- get numerically correct answers
- via correct transformation (time/space complexity)
- make $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$ first class
- make them apply to *all* (differentiable) programs
- handle nested application (self-application of $\overrightarrow{\mathcal{J}}$ and $\overleftarrow{\mathcal{J}}$)
- good puns
  - mathematical: the $\lambda\nabla$-calculus
  - programming language: VLAD
  - implementation: STALIN∇
- manuscripts and code:
  `http://www-bcl.cs.nuim.ie/~qobi/stalingrad/`