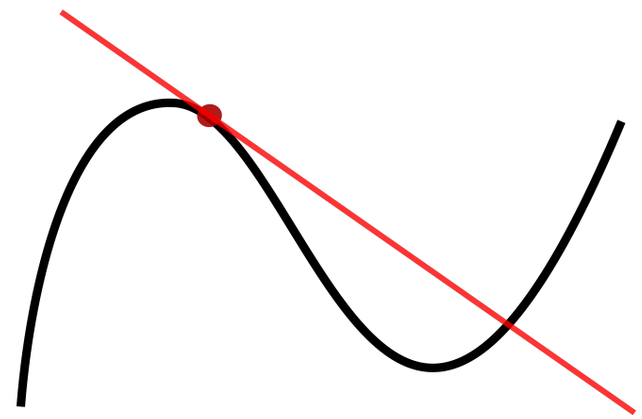
A (quite partial) introduction to differentiable programming

Michele Pagani
(LIP, École Normale Supérieure de Lyon)

Derivatives, gradients, Jacobians

Derivatives, gradients, Jacobians

$$f : \mathbb{R} \to \mathbb{R}$$

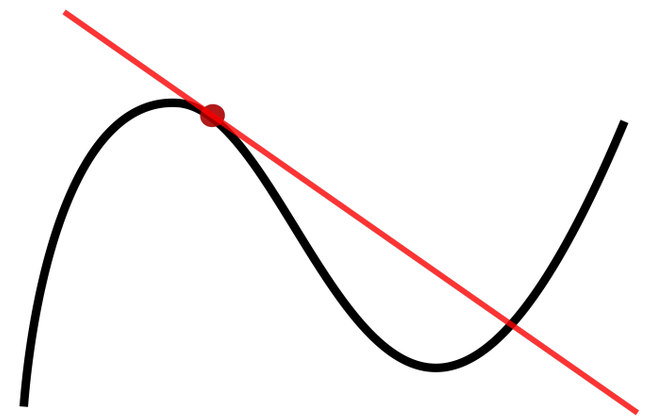


$$\partial f(x)$$

$$:= \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

Derivatives, gradients, Jacobians

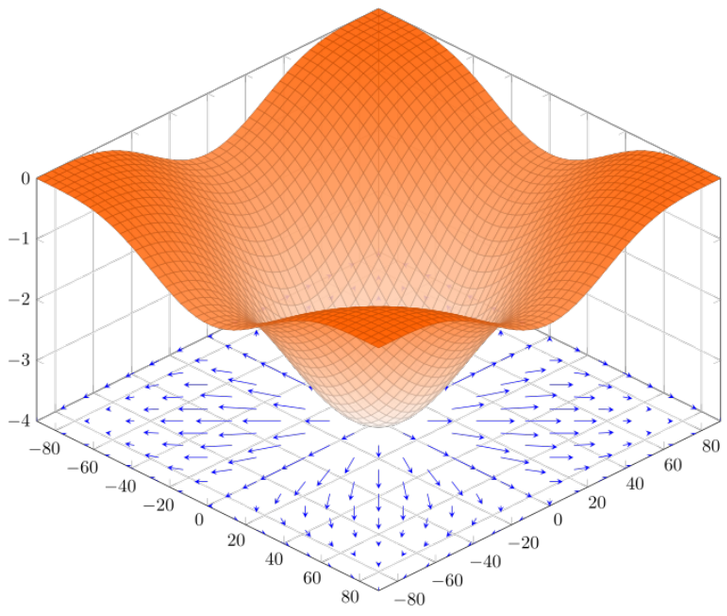
$$f : \mathbb{R} \to \mathbb{R}$$



$$\partial f(x)$$

$$:= \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

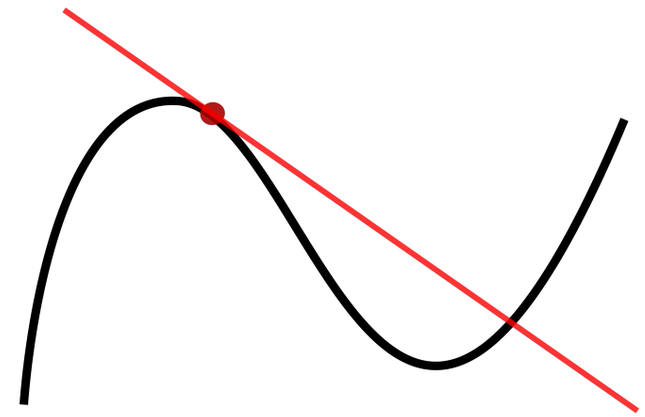
$$f : \mathbb{R}^n \to \mathbb{R}$$



$$\nabla_{\vec{x}}(f) := \begin{pmatrix} \partial_1 f(\vec{x}) \\ \vdots \\ \partial_n f(\vec{x}) \end{pmatrix}$$

Derivatives, gradients, Jacobians

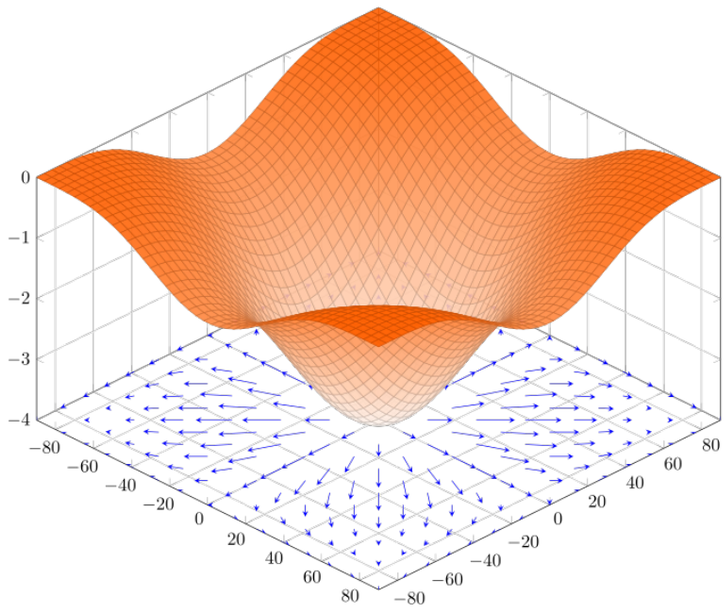
$f : \mathbb{R} \to \mathbb{R}$



$$\partial f(x)$$

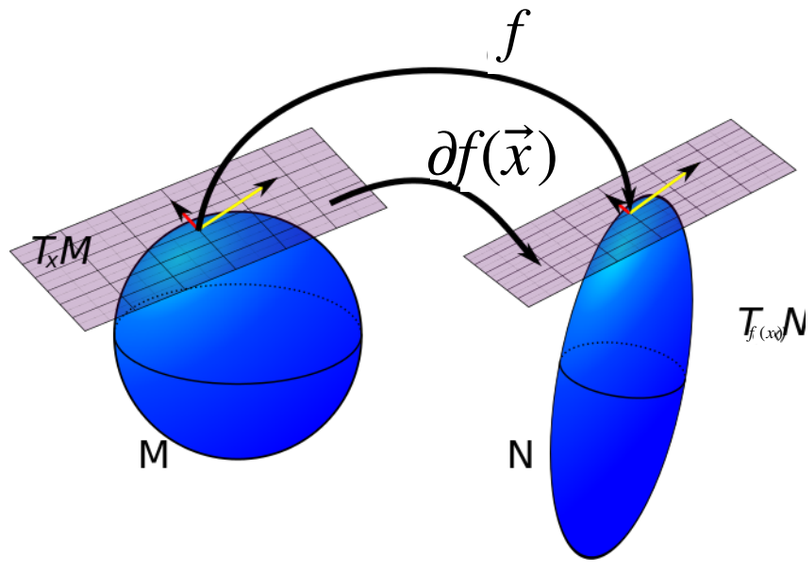
$$:= \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$

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



$$\nabla_{\vec{x}}(f) := \begin{pmatrix} \partial_1 f(\vec{x}) \\ \vdots \\ \partial_n f(\vec{x}) \end{pmatrix}$$

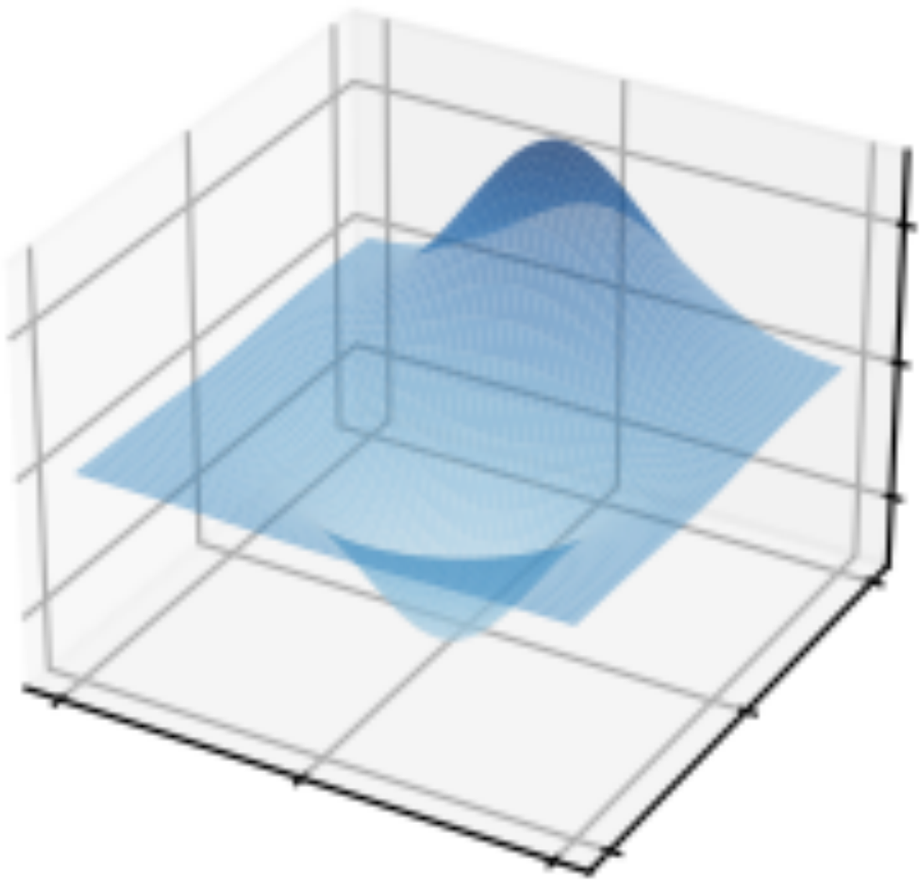
$f : \mathbb{R}^n \to \mathbb{R}^m$

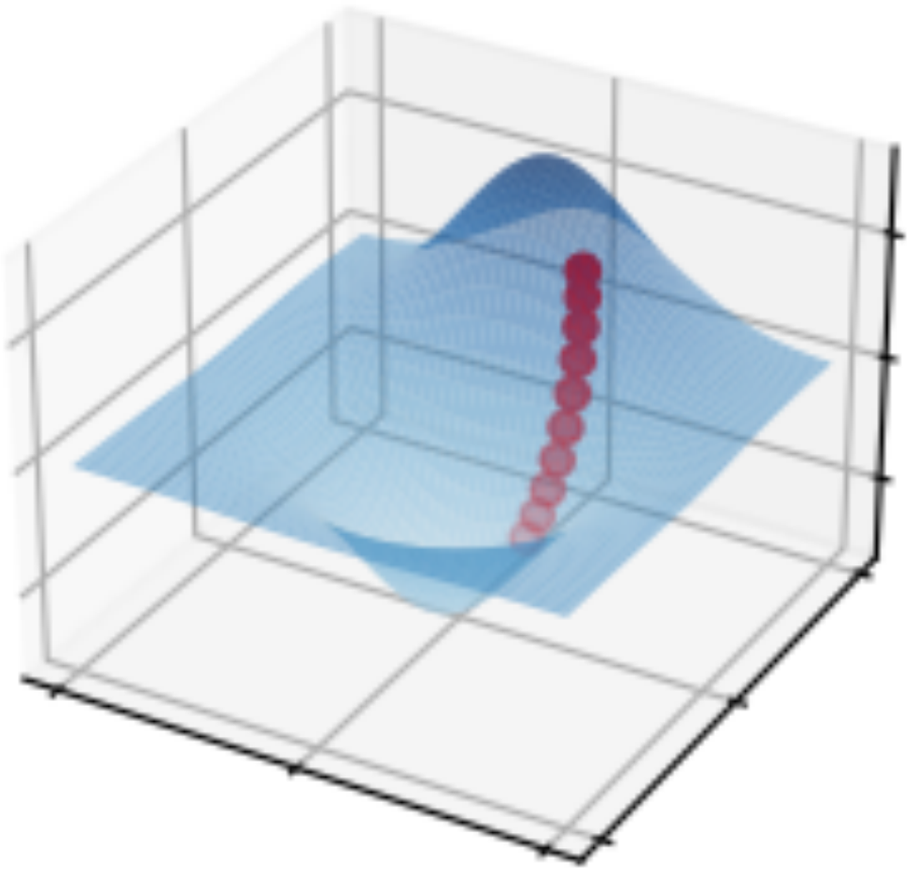


$$\partial f(\vec{x})$$

$$:= \begin{pmatrix} \partial_1 f_1(\vec{x}) & \cdots & \partial_n f_1(\vec{x}) \\ \vdots & \ddots & \vdots \\ \partial_1 f_m(\vec{x}) & \cdots & \partial_n f_m(\vec{x}) \end{pmatrix}$$

An application of AD

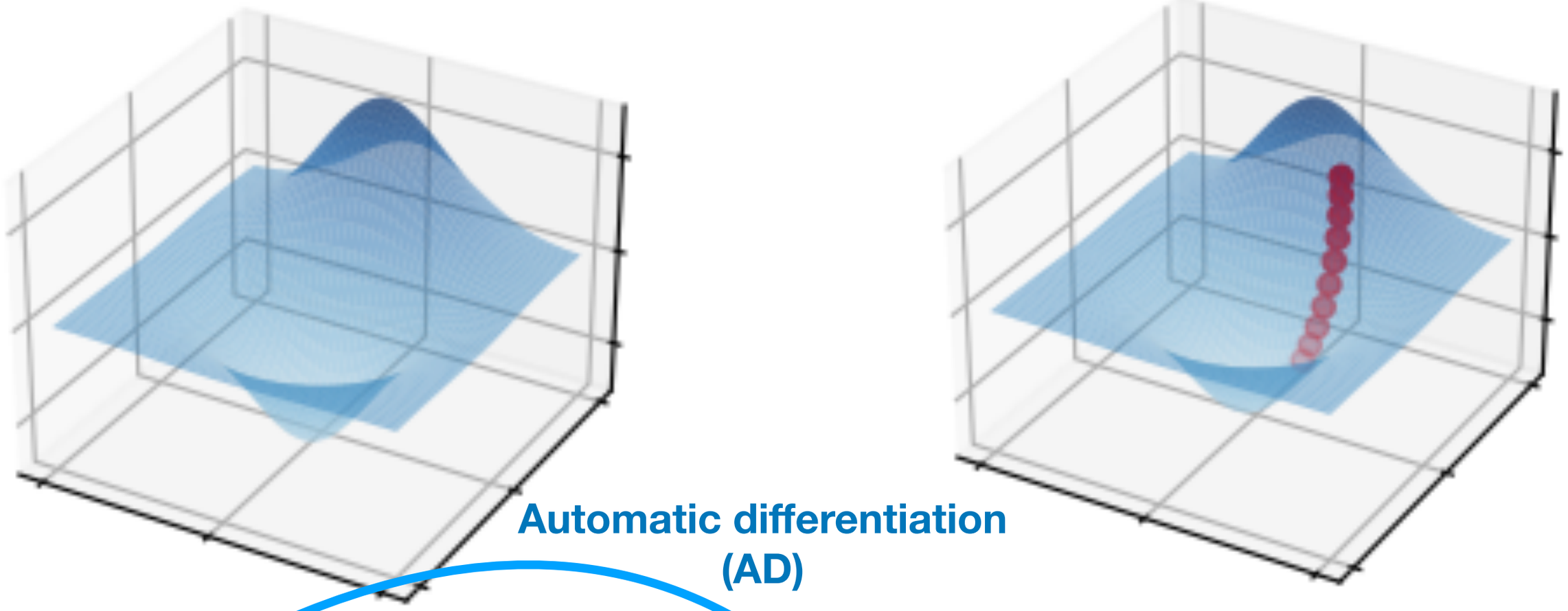




```
def f(x1,x2):
    z1 = x1**2 + x2**2
    z2 = jnp.exp(-z1)
    z3 = x1**2 + (x2-1)**2
    z4 = jnp.exp(-z3)
    z5 = z4 - z2
    return z5
```

```
x1 = jax.random.uniform(key=key1)
x2 = jax.random.uniform(key=key2)
d_f = jax.grad(f, (0, 1))
lr = 0.1
for i in range(10):
    dx1, dx2 = d_f(x1, x2)
    x1 -= lr * dx1
    x2 -= lr * dx2
```

An application of AD



Automatic differentiation (AD)

```
def f(x1,x2):
    z1 = x1**2 + x2**2
    z2 = jnp.exp(-z1)
    z3 = x1**2 + (x2-1)**2
    z4 = jnp.exp(-z3)
    z5 = z4 - z2
    return z5
```

```
x1 = jax.random.uniform(key=key1)
x2 = jax.random.uniform(key=key2)
d_f = jax.grad(f, (0, 1))
lr = 0.1
for i in range(10):
    dx1, dx2 = d_f(x1, x2)
    x1 -= lr * dx1
    x2 -= lr * dx2
```

AD compositional

$$\mathrm{grad}(\texttt{let}\ y = G\ \texttt{in}\ H) \quad \Rightarrow \quad \texttt{let}\ \bar{y} = \mathrm{grad}(G)\ \texttt{in}\ \mathrm{grad}(H)$$

AD compositional

$$\mathrm{grad}(\texttt{let}\ y = G\ \texttt{in}\ H) \quad \Rightarrow \quad \texttt{let}\ \bar{y} = \mathrm{grad}(G)\ \texttt{in}\ \mathrm{grad}(H)$$

- Make the chain rule compositional:

$$\mathbb{R} \xrightarrow{g} \mathbb{R} \xrightarrow{h} \mathbb{R}$$

$$\partial(h \circ g)(x) = \partial h(g(x)) \cdot \partial g(x)$$

AD compositional

$$\mathrm{grad}(\texttt{let}\ y = G\ \texttt{in}\ H) \quad \Rightarrow \quad \texttt{let}\ \bar{y} = \mathrm{grad}(G)\ \texttt{in}\ \mathrm{grad}(H)$$

- Make the chain rule compositional:

$$\mathbb{R} \xrightarrow{g} \mathbb{R} \xrightarrow{h} \mathbb{R}$$

$$\partial(h \circ g)(x) = \partial h(g(x)) \cdot \partial g(x)$$

- Propagating **primals** and **tangents** :

$$\mathscr{D}(F) \simeq (x, \dot{x}) \mapsto (F(x), \partial(F)(x) \cdot \dot{x})$$

$$\mathscr{D}(\texttt{let}\ y = G\ \texttt{in}\ H) \simeq \texttt{let}\ (y, \dot{y}) = \mathscr{D}(G)\ \texttt{in}\ \mathscr{D}(H))$$

$$\mathrm{grad}(F)(x) \simeq \pi_2(\mathscr{D}(F)(x,1))$$

AD forward / backward

$$\mathbb{R}^n \xrightarrow{g} \mathbb{R}^q \xrightarrow{h} \mathbb{R}^p \xrightarrow{k} \mathbb{R}^m$$

$$\mathcal{J}_{\vec{x}}(k \circ h \circ g) =$$

$$\mathcal{J}_{h(g(\vec{x}))}(k) \cdot \mathcal{J}_{g(\vec{x})}(h) \cdot \mathcal{J}_{\vec{x}}(g)$$

AD forward / backward

$$\mathbb{R}^n \xrightarrow{g} \mathbb{R}^q \xrightarrow{h} \mathbb{R}^p \xrightarrow{k} \mathbb{R}^m$$

$$\mathcal{J}_{\vec{x}}(k \circ h \circ g) =$$

$$\mathcal{J}_{h(g(\vec{x}))}(k) \cdot \mathcal{J}_{g(\vec{x})}(h) \cdot \mathcal{J}_{\vec{x}}(g)$$

AD forward / backward

$$\mathbb{R}^n \xrightarrow{g} \mathbb{R}^q \xrightarrow{h} \mathbb{R}^p \xrightarrow{k} \mathbb{R}^m$$

$$\mathcal{J}_{\vec{x}}(k \circ h \circ g) =$$

$$\mathcal{J}_{h(g(\vec{x}))}(k) \cdot \mathcal{J}_{g(\vec{x})}(h) \cdot \mathcal{J}_{\vec{x}}(g)$$

AD forward / backward

$$\mathbb{R}^n \xrightarrow{g} \mathbb{R}^q \xrightarrow{h} \mathbb{R}^p \xrightarrow{k} \mathbb{R}^m$$

$$\begin{pmatrix} \partial_1 g_1 \bullet & \cdots & \partial_n g_1 \bullet \\ \vdots & \ddots & \vdots \\ \partial_1 g_q \bullet & \cdots & \partial_n g_q \bullet \end{pmatrix}$$

$$\begin{pmatrix} \partial_1 h_1 \bullet & \cdots & \partial_q h_1 \bullet \\ \vdots & \ddots & \vdots \\ \partial_1 h_p \bullet & \cdots & \partial_q h_p \bullet \end{pmatrix}$$

$$\mathbb{R}^{p \times \mathbf{n}}$$

$$\begin{pmatrix} \partial_1 k_1 \bullet & \cdots & \partial_p k_1 \bullet \\ \vdots & \ddots & \vdots \\ \partial_1 k_m \bullet & \cdots & \partial_p k_m \bullet \end{pmatrix}$$

$$\mathbb{R}^{m \times \mathbf{n}}$$

$$\mathcal{J}_{\vec{x}}(k \circ h \circ g) =$$

$$\mathcal{J}_{h(g(\vec{x}))}(k) \cdot \mathcal{J}_{g(\vec{x})}(h) \cdot \mathcal{J}_{\vec{x}}(g)$$

AD forward / backward

$$\mathbb{R}^n \xrightarrow{g} \mathbb{R}^q \xrightarrow{h} \mathbb{R}^p \xrightarrow{k} \mathbb{R}^m$$

$$\begin{pmatrix} \partial_1 g_1 \bullet & \cdots & \partial_n g_1 \bullet \\ \vdots & \ddots & \vdots \\ \partial_1 g_q \bullet & \cdots & \partial_n g_q \bullet \end{pmatrix}$$

$$\begin{pmatrix} \partial_1 h_1 \bullet & \cdots & \partial_q h_1 \bullet \\ \vdots & \ddots & \vdots \\ \partial_1 h_p \bullet & \cdots & \partial_q h_p \bullet \end{pmatrix}$$

$$\mathbb{R}^{p \times \mathbf{n}}$$

$$\begin{pmatrix} \partial_1 k_1 \bullet & \cdots & \partial_p k_1 \bullet \\ \vdots & \ddots & \vdots \\ \partial_1 k_m \bullet & \cdots & \partial_p k_m \bullet \end{pmatrix}$$

$$\mathbb{R}^{m \times \mathbf{n}}$$

$$\mathcal{J}_{\vec{x}}(k \circ h \circ g) =$$

$$\mathcal{J}_{h(g(\vec{x}))}(k) \cdot \mathcal{J}_{g(\vec{x})}(h) \cdot \mathcal{J}_{\vec{x}}(g)$$

$$\begin{pmatrix} \partial_1 h_1 \bullet & \cdots & \partial_q h_1 \bullet \\ \vdots & \ddots & \vdots \\ \partial_1 h_p \bullet & \cdots & \partial_q h_p \bullet \end{pmatrix} \begin{pmatrix} \partial_1 g_1 \bullet & \cdots & \partial_n g_1 \bullet \\ \vdots & \ddots & \vdots \\ \partial_1 g_q \bullet & \cdots & \partial_n g_q \bullet \end{pmatrix}$$

$$\begin{pmatrix} \partial_1 k_1 \bullet & \cdots & \partial_p k_1 \bullet \\ \vdots & \ddots & \vdots \\ \partial_1 k_m \bullet & \cdots & \partial_p k_m \bullet \end{pmatrix}$$

$$\mathbb{R}^{\mathbf{m} \times q}$$

$$\mathbb{R}^{\mathbf{m} \times n}$$

AD forward / backward

$$\mathbb{R}^n \xrightarrow{\ g\ } \mathbb{R}^q$$

$$\overrightarrow{\mathscr{D}}(g) \quad \simeq \quad (\mathbf{x}, \dot{\mathbf{x}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g) \cdot \dot{\mathbf{x}})$$

AD forward / backward

$$\mathbb{R}^n \xrightarrow{\ g\ } \mathbb{R}^q$$

$$\overrightarrow{\mathscr{D}}(g) \simeq (\mathbf{x}, \dot{\mathbf{x}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g) \cdot \dot{\mathbf{x}})$$

$\mathbb{R}^n \multimap \mathbb{R}^q$

perturbation in input

perturbation in output

AD forward / backward

$$\mathbb{R}^n \xrightarrow{\ g\ } \mathbb{R}^q$$

$$\overrightarrow{\mathscr{D}}(g) \simeq (\mathbf{x}, \dot{\mathbf{x}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g) \cdot \dot{\mathbf{x}})$$

$\mathbb{R}^n \multimap \mathbb{R}^q$

perturbation in input

perturbation in output

$$\overleftarrow{\mathscr{D}}(g) \simeq (\mathbf{x}, \dot{\mathbf{y}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g)^T \cdot \dot{\mathbf{y}})$$

$\mathbb{R}^q \multimap \mathbb{R}^n$

perturbation in output

perturbation in input

AD forward / backward

$$\mathbb{R}^n \xrightarrow{\ g\ } \mathbb{R}^q \xrightarrow{\ h\ } \mathbb{R}^p$$

$$\overrightarrow{\mathscr{D}}(g) \simeq (\mathbf{x}, \dot{\mathbf{x}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g) \cdot \dot{\mathbf{x}})$$

$\mathbb{R}^n \multimap \mathbb{R}^q$

perturbation in input

perturbation in output

$$\overleftarrow{\mathscr{D}}(g) \simeq (\mathbf{x}, \dot{\mathbf{y}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g)^T \cdot \dot{\mathbf{y}})$$

$\mathbb{R}^q \multimap \mathbb{R}^n$

perturbation in output

perturbation in input

AD forward / backward

$$\mathbb{R}^n \xrightarrow{\ g\ } \mathbb{R}^q \xrightarrow{\ h\ } \mathbb{R}^p$$

$$\overrightarrow{\mathscr{D}}(g) \simeq (\mathbf{x}, \dot{\mathbf{x}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g) \cdot \dot{\mathbf{x}})$$

$\overrightarrow{\mathscr{D}}(h)$ composes

$\mathbb{R}^n \multimap \mathbb{R}^q$

perturbation in input

perturbation in output

$$\overleftarrow{\mathscr{D}}(g) \simeq (\mathbf{x}, \dot{\mathbf{y}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g)^T \cdot \dot{\mathbf{y}})$$

$\mathbb{R}^q \multimap \mathbb{R}^n$

perturbation in output

perturbation in input

AD forward / backward

$$\mathbb{R}^n \xrightarrow{\ g\ } \mathbb{R}^q \xrightarrow{\ h\ } \mathbb{R}^p$$

$$\overrightarrow{\mathscr{D}}(g) \simeq (\mathbf{x}, \dot{\mathbf{x}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g) \cdot \dot{\mathbf{x}})$$

$\overrightarrow{\mathscr{D}}(h)$ composes

$\mathbb{R}^n \multimap \mathbb{R}^q$

perturbation in input

perturbation in output

$$\overleftarrow{\mathscr{D}}(g) \simeq (\mathbf{x}, \dot{\mathbf{y}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g)^T \cdot \dot{\mathbf{y}})$$

$\overleftarrow{\mathscr{D}}(h)$ does not compose

$\mathbb{R}^q \multimap \mathbb{R}^n$

perturbation in output

perturbation in input

How to compose AD backward ?
some approaches

How to compose AD backward ? some approaches

- **“Tape” (as in e.g. JAX)**
  - Wengert, *A simple automatic derivative evaluation program*, Comm. ACM 1964
  - Frostig, Johnson, MacLaurin, Paszke, Radul, *You Only Linearise Once: Tangents Transpose to Gradients*, POPL 2023

How to compose AD backward ? some approaches

- **“Tape” (as in e.g. JAX)**
  - Wengert, *A simple automatic derivative evaluation program*, Comm. ACM 1964
  - Frostig, Johnson, MacLaurin, Paszke, Radul, *You Only Linearise Once: Tangents Transpose to Gradients*, POPL 2023
- **Back-propagator (+ linear factoring)**
  - Pearlmutter, Siskind, *Reverse-mode AD in a functional framework: lambda the ultimate backpropagator,* ACM Trans. 2008
  - Brunel, Mazza, P., *Backpropagation in simply typed lambda-calculus with linear negation,* POPL 2020
  - Smedig, Vákár, *Efficient Dual-Numbers Reverse AD via Well-Known Program Transformations.* POPL 2023

How to compose AD backward ? some approaches

- **“Tape” (as in e.g. JAX)**
  - Wengert, *A simple automatic derivative evaluation program*, Comm. ACM 1964
  - Frostig, Johnson, MacLaurin, Paszke, Radul, *You Only Linearise Once: Tangents Transpose to Gradients*, POPL 2023
- **Back-propagator (+ linear factoring)**
  - Pearlmutter, Siskind, *Reverse-mode AD in a functional framework: lambda the ultimate backpropagator,* ACM Trans. 2008
  - Brunel, Mazza, P., *Backpropagation in simply typed lambda-calculus with linear negation,* POPL 2020
  - Smedig, Vákár, *Efficient Dual-Numbers Reverse AD via Well-Known Program Transformations.* POPL 2023
- **CHAD (and/or) Dialectica**
  - Elliott, *The simple essence of automatic differentiation*, ICFP 2018
  - Vákár, *Reverse AD at Higher Types: Pure, Principled and Denotationally Correct.* ESOP 2021
  - Kerjean, Pédrot, *∂ for Dialectica*. LICS 2024

How to compose AD backward ? some approaches

- **“Tape” (as in e.g. JAX)**
  - Wengert, *A simple automatic derivative evaluation program*, Comm. ACM 1964
  - Frostig, Johnson, MacLaurin, Paszke, Radul, *You Only Linearise Once: Tangents Transpose to Gradients*, POPL 2023
- **Back-propagator (+ linear factoring)**
  - Pearlmutter, Siskind, *Reverse-mode AD in a functional framework: lambda the ultimate backpropagator,* ACM Trans. 2008
  - Brunel, Mazza, P., *Backpropagation in simply typed lambda-calculus with linear negation,* POPL 2020
  - Smedig, Vákár, *Efficient Dual-Numbers Reverse AD via Well-Known Program Transformations.* POPL 2023
- **CHAD (and/or) Dialectica**
  - Elliott, *The simple essence of automatic differentiation*, ICFP 2018
  - Vákár, *Reverse AD at Higher Types: Pure, Principled and Denotationally Correct.* ESOP 2021
  - Kerjean, Pédrot, *∂ for Dialectica*. LICS 2024

“Tape” (as e.g. in Jax)

$$\overleftarrow{\mathscr{D}}(g) \simeq (\mathbf{x}, \dot{\mathbf{y}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g)^T \cdot \dot{\mathbf{y}})$$

perturbation in output

$\mathbb{R}^q \multimap \mathbb{R}^n$

perturbation input

$$\overleftarrow{\mathscr{D}}(g) : \mathbb{R}^n \times \mathbb{L} \to \mathbb{R}^q \times \mathbb{L}$$

$$\overleftarrow{\mathscr{D}}(g)(\mathbf{x}, \ell) := (f(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g)^T :: \ell)$$

$$F : \mathbb{R}^d \to \cdots \to \mathbb{R}^n \xrightarrow{g} \mathbb{R}^q \to \cdots \to \mathbb{R}$$

$$\texttt{grad}(F)(\mathbf{x}) := \texttt{let } (\_, \ell) = \overleftarrow{\mathscr{D}}(F)(\mathbf{x}, [1]) \texttt{ in } \prod \ell$$

“Tape” (as e.g. in Jax)

```
# function to differentiate
def F(x1, x2):
    z1 = x1 * x2
    z2 = z1 ** z1
    return z2

# jacobians
def J_prod(x1,x2):
    return np.array([[x2,x1]])

def J_exp(y1,y2):
    return np.array([[y2 * (y1 ** (y2-1)),(y1 ** y2) * np.log(y1)]])

def J_diag_t(a):    #trasponse diagonal gives sum
    return np.array([a[0]+a[1]])
```

“Tape” (as e.g. in Jax)

```
# tape-based transformation (jax)
def DF_nonlin(x1, x2):
    z1 = x1 * x2
    jz1 = J_prod(x1,x2)
    z2 = z1 ** z1
    jz2 = J_exp(z1,z1)
    return (z2, jz1, jz2)
```

```
def DF_lin(dz2, jz1, jz2):
    dz1 = jz2.transpose() @ dz2
    dz1_dup = np.array([dz1[0]+dz1[1]]) #trasponse diagonal gives sum
    dx = jz1.transpose() @ dz1_dup
    return dx

def jax_grad_F(x1, x2):
    (_, jz1, jz2) = DF_nonlin(x1, x2)
    return DF_lin(np.array([[1]]), jz1, jz2)
```

AD backward: some approaches

- **“Tape” (as in e.g. JAX)**
  - Wengert, *A simple automatic derivative evaluation program*, Comm. ACM 1964
  - Frostig, Johnson, MacLaurin, Paszke, Radul, *You Only Linearise Once: Tangents Transpose to Gradients*, POPL 2023

- **Back-propagator (+ linear factoring)**
  - Pearlmutter, Siskind, *Reverse-mode AD in a functional framework: lambda the ultimate backpropagator,* ACM Trans. 2008
  - Brunel, Mazza, P., *Backpropagation in simply typed lambda-calculus with linear negation,* POPL 2020
  - Smedig, Vákár, *Efficient Dual-Numbers Reverse AD via Well-Known Program Transformations.* POPL 2023

- **CHAD (and/or) Dialectica**
  - Elliott, *The simple essence of automatic differentiation*, ICFP 2018
  - Vákár, *Reverse AD at Higher Types: Pure, Principled and Denotationally Correct.* ESOP 2021
  - Kerjean, Pédrot, *∂ for Dialectica*. LICS 2024

Back-propagators (+ linear factoring)

$$\overleftarrow{\mathscr{D}}(g) \simeq (\mathbf{x}, \dot{\mathbf{y}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g)^T \cdot \dot{\mathbf{y}})$$

perturbation in output

$\mathbb{R}^q \multimap \mathbb{R}^n$

perturbation in input

$$F : \mathbb{R}^d \to \cdots \to \mathbb{R}^n \xrightarrow{g} \mathbb{R}^q \to \cdots \to \mathbb{R}$$

$$\overleftarrow{\mathscr{D}}(g) : \mathbb{R}^n \times (\mathbb{R}^n \multimap \mathbb{R}^d) \to \mathbb{R}^q \times (\mathbb{R}^q \multimap \mathbb{R}^d)$$

$$\overleftarrow{\mathscr{D}}(g)(\mathbf{x}, b) := (g(\mathbf{x}), \lambda \dot{\mathbf{y}} \,.\, b(\mathscr{J}_{\mathbf{x}}(g)^T \cdot \dot{\mathbf{y}}))$$

$$\texttt{grad}(F)(\mathbf{x}) := \texttt{let}\ (\_, b) = \overleftarrow{\mathscr{D}}(F)(\mathbf{x}, \lambda \dot{\mathbf{x}} \,.\, \dot{\mathbf{x}})\ \texttt{in}\ b(\mathbf{1})$$

Back-propagators (+ linear factoring)

$$\overleftarrow{\mathscr{D}}(g) \simeq (\mathbf{x}, \dot{\mathbf{y}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g)^T \cdot \dot{\mathbf{y}})$$

perturbation in output

$\mathbb{R}^q \multimap \mathbb{R}^n$

$\mathbb{R}^d$

perturbation in input

perturbation global input

$$F : \mathbb{R}^d \to \cdots \to \mathbb{R}^n \xrightarrow{g} \mathbb{R}^q \to \cdots \to \mathbb{R}$$

$$\overleftarrow{\mathscr{D}}(g) : \mathbb{R}^n \times (\mathbb{R}^n \multimap \mathbb{R}^d) \to \mathbb{R}^q \times (\mathbb{R}^q \multimap \mathbb{R}^d)$$

$$\overleftarrow{\mathscr{D}}(g)(\mathbf{x}, b) := (g(\mathbf{x}), \lambda \dot{\mathbf{y}} \,.\, b(\mathscr{J}_{\mathbf{x}}(g)^T \cdot \dot{\mathbf{y}}))$$

$$\texttt{grad}(F)(\mathbf{x}) := \texttt{let}\ (\_, b) = \overleftarrow{\mathscr{D}}(F)(\mathbf{x}, \lambda \dot{\mathbf{x}} \,.\, \dot{\mathbf{x}})\ \texttt{in}\ b(\mathbf{1})$$

Back-propagators (+ linear factoring)

```
# backpropagators-based transformation
def DF(x1, x2, xb):
    z1 = x1 * x2
    z1b = lambda dz1: xb(J_prod(x1,x2).transpose() @ dz1)
    z2 = z1 ** z1
    z2b = lambda dz2: J_diag_t(z1b(J_exp(z1,z1).transpose() @ dz2)) # sum over R^d
    return (z2, z2b)
```

```
# after factor rule optimization
def DF_opt(x1, x2, xb):
    z1 = x1 * x2
    z1b = lambda dz1: xb(J_prod(x1,x2).transpose() @ dz1)
    z2 = z1 ** z1
    z2b = lambda dz2: z1b(J_diag_t(J_exp(z1,z1).transpose() @ dz2))  # sum over R
    return (z2, z2b)

def bp_grad_F(x1, x2):
    _, b = DF_opt(x1, x2, lambda dx:dx)
    return b (np.array([[1]]))
```

AD backward: some approaches

- **“Tape” (as in e.g. JAX)**
  - Wengert, *A simple automatic derivative evaluation program*, Comm. ACM 1964
  - Frostig, Johnson, MacLaurin, Paszke, Radul, *You Only Linearise Once: Tangents Transpose to Gradients*, POPL 2023

- **Back-propagator (+ linear factoring)**
  - Pearlmutter, Siskind, *Reverse-mode AD in a functional framework: lambda the ultimate backpropagator,* ACM Trans. 2008
  - Brunel, Mazza, P., *Backpropagation in simply typed lambda-calculus with linear negation,* POPL 2020
  - Smedig, Vákár, *Efficient Dual-Numbers Reverse AD via Well-Known Program Transformations.* POPL 2023

- **CHAD (and/or) Dialectica**
  - Elliott, *The simple essence of automatic differentiation*, ICFP 2018
  - Vákár, *Reverse AD at Higher Types: Pure, Principled and Denotationally Correct.* ESOP 2021
  - Kerjean, Pédrot, *∂ for Dialectica*. LICS 2024

CHAD

$$\overleftarrow{\mathscr{D}}(g) \quad \simeq \quad (\mathbf{x}, \dot{\mathbf{y}}) \mapsto (g(\mathbf{x}), \mathscr{J}_{\mathbf{x}}(g)^T \cdot \dot{\mathbf{y}})$$

$$\simeq \quad (\mathbf{x} \mapsto g(\mathbf{x}), (\mathbf{x}, \dot{\mathbf{y}}) \mapsto \mathscr{J}_{\mathbf{x}}(g)^T \cdot \dot{\mathbf{y}})$$

$$F : \mathbb{R}^d \to \cdots \to \mathbb{R}^n \xrightarrow{g} \mathbb{R}^m \to \cdots \to \mathbb{R}$$

$$\overleftarrow{\mathscr{D}}(g) : (\mathbb{R}^n \to \mathbb{R}^m, !\mathbb{R}^n \otimes \mathbb{R}^m \multimap \mathbb{R}^n)$$

$$\overleftarrow{\mathscr{D}}_1(g)(\mathbf{x}) := g(\mathbf{x}), \quad \overleftarrow{\mathscr{D}}_2(g)(\mathbf{x}, \dot{\mathbf{y}}) := \mathscr{J}_{\mathbf{x}}^T(g) \cdot \dot{\mathbf{y}}$$

$$\texttt{grad}(F)(\mathbf{x}) := \texttt{let } (\_, d) = \overleftarrow{\mathscr{D}}(F)(\mathbf{x}) \texttt{ in } d(\mathbf{x},1)$$

AD on more complex data types

$\texttt{Bool}, A \times B, A \to B, \ldots$

AD on more complex data types

$\texttt{Bool}, A \times B, A \to B, \ldots$

- **“Tape” (as in e.g. JAX)**
  - Considered just basic control flows, work-in-progress…
  - Specific constructions, not completely understood from a theoretical point of view
  - eg. jax.cond, jax.switch, jax.loop,…

AD on more complex data types

$$\texttt{Bool}, A \times B, A \to B, \ldots$$

- **"Tape" (as in e.g. JAX)**
  - Considered just basic control flows, work-in-progress…
  - Specific constructions, not completely understood from a theoretical point of view
  - eg. jax.cond, jax.switch, jax.loop,…

- **Back-propagator (+ linear factoring)**
  - Simple "operator overloading"

$$\overleftarrow{\mathscr{D}}(A \times B) := \overleftarrow{\mathscr{D}}(A) \times \overleftarrow{\mathscr{D}}(B)$$

$$\overleftarrow{\mathscr{D}}(A \Rightarrow B) := \overleftarrow{\mathscr{D}}(A) \Rightarrow \overleftarrow{\mathscr{D}}(B)$$

$$\overleftarrow{\mathscr{D}}(x^A) := \begin{cases} (x, \dot{x}) & \textbf{if } A = \mathbb{R}^n \\ x^{\overleftarrow{\mathscr{D}}(A)} & \textbf{otherwise} \end{cases}$$

$$\overleftarrow{\mathscr{D}}(\lambda x^A \,.\, G) := \lambda \overleftarrow{\mathscr{D}}(x) \,.\, \overleftarrow{\mathscr{D}}(G)$$

$$\overleftarrow{\mathscr{D}}(GH) := \overleftarrow{\mathscr{D}}(G)\overleftarrow{\mathscr{D}}(H)$$

AD on more complex data types

$$\texttt{Bool}, A \times B, A \to B, \ldots$$

- **"Tape" (as in e.g. JAX)**
  - Considered just basic control flows, work-in-progress…
  - Specific constructions, not completely understood from a theoretical point of view
  - eg. jax.cond, jax.switch, jax.loop,…

- **Back-propagator (+ linear factoring)**
  - Simple "operator overloading"

$$\overleftarrow{\mathscr{D}}(A \times B) := \overleftarrow{\mathscr{D}}(A) \times \overleftarrow{\mathscr{D}}(B)$$

$$\overleftarrow{\mathscr{D}}(A \Rightarrow B) := \overleftarrow{\mathscr{D}}(A) \Rightarrow \overleftarrow{\mathscr{D}}(B)$$

$$\overleftarrow{\mathscr{D}}(x^A) := \begin{cases} (x, \dot{x}) & \textbf{if } A = \mathbb{R}^n \\ x^{\overleftarrow{\mathscr{D}}(A)} & \textbf{otherwise} \end{cases}$$

$$\overleftarrow{\mathscr{D}}(\lambda x^A \,.\, G) := \lambda \overleftarrow{\mathscr{D}}(x) \,.\, \overleftarrow{\mathscr{D}}(G)$$

$$\overleftarrow{\mathscr{D}}(GH) := \overleftarrow{\mathscr{D}}(G)\overleftarrow{\mathscr{D}}(H)$$

- **CHAD (and/or) Dialectica**
  - "Grothendieck construction" (or Dialectica interpretation) on a linear logic

$$\overleftarrow{\mathscr{D}}_1(A \times B) := \overleftarrow{\mathscr{D}}_1(A) \times \overleftarrow{\mathscr{D}}_2(B)$$

$$\overleftarrow{\mathscr{D}}_1(A \Rightarrow B) := \overleftarrow{\mathscr{D}}_1(A) \Rightarrow \overleftarrow{\mathscr{D}}_2(B) \times (\overleftarrow{\mathscr{D}}_2(B) \multimap \overleftarrow{\mathscr{D}}_1(A))$$

$$\overleftarrow{\mathscr{D}}_2(A \times B) := !\overleftarrow{\mathscr{D}}_1(A) \times \overleftarrow{\mathscr{D}}_2(B)$$

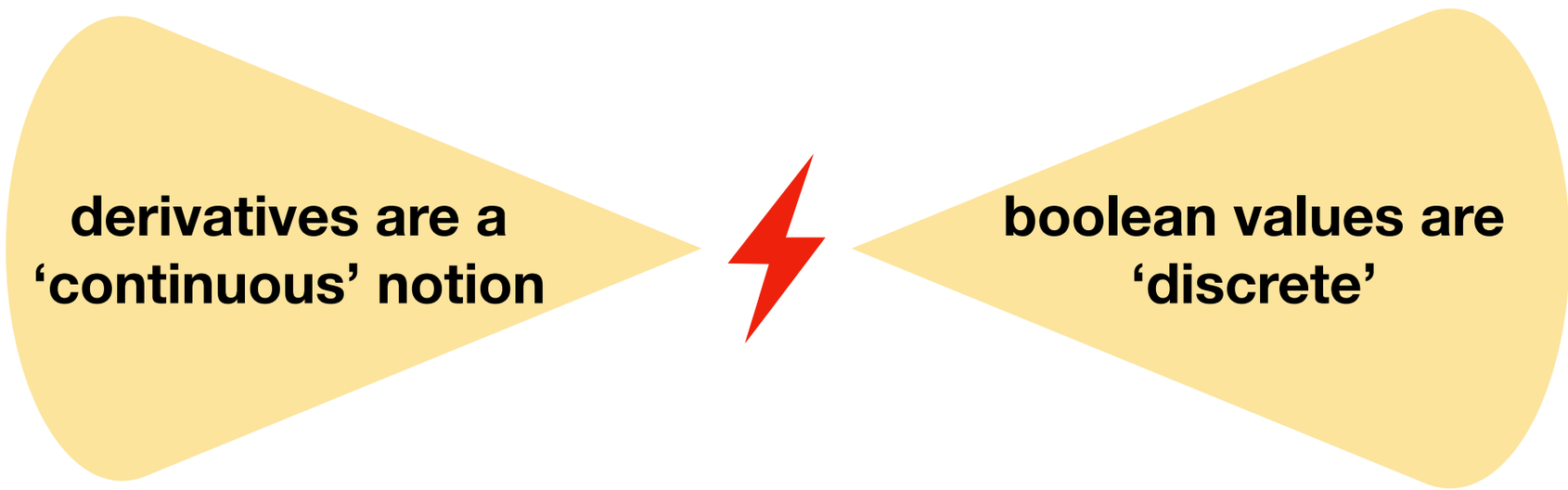
$$\overleftarrow{\mathscr{D}}_2(A \Rightarrow B) := !\overleftarrow{\mathscr{D}}_1(A) \otimes \overleftarrow{\mathscr{D}}_2(B)$$

AD may go wrong

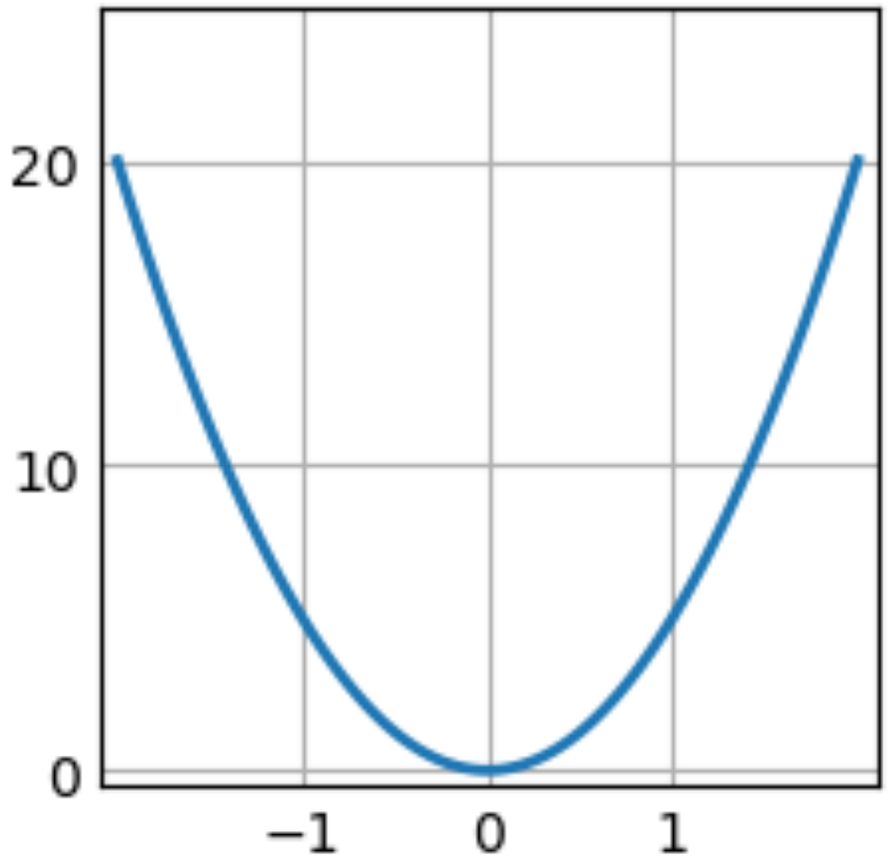
$$\mathscr{D}(\texttt{if } B \texttt{ then } G \texttt{ else } H) := \texttt{if } B \texttt{ then } \mathscr{D}(G) \texttt{ else } \mathscr{D}(H)$$

AD may go wrong

$$\mathscr{D}(\texttt{if } B \texttt{ then } G \texttt{ else } H) := \texttt{if } B \texttt{ then } \mathscr{D}(G) \texttt{ else } \mathscr{D}(H)$$



AD may go wrong



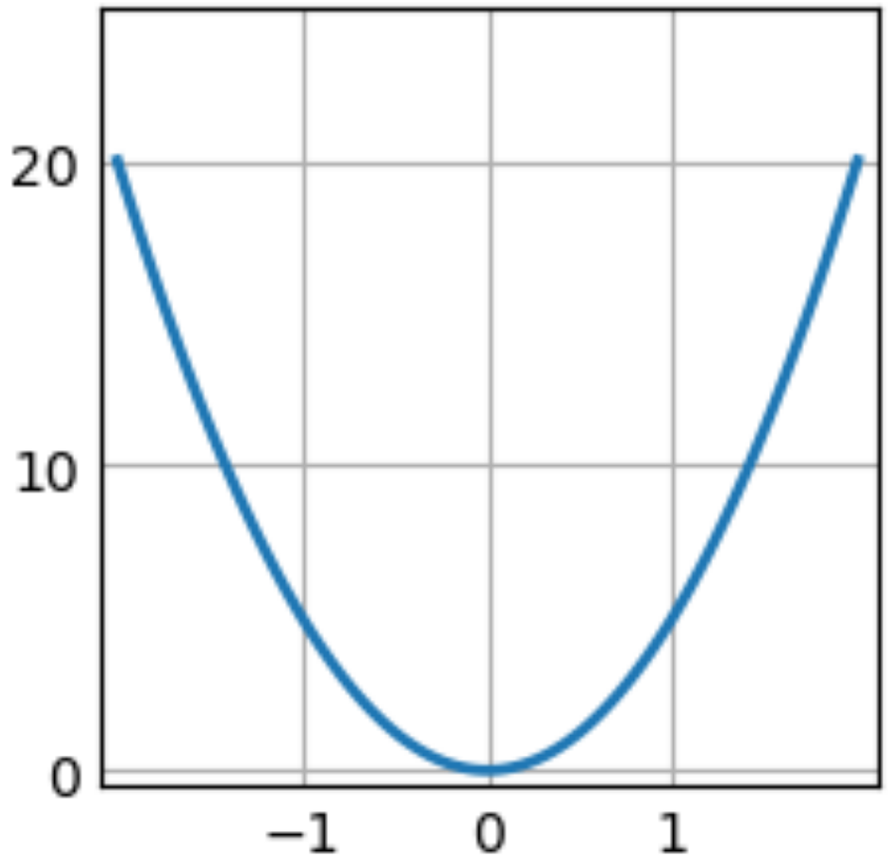

```
epoch 0, x = 0.0 y = 0.00, dx = 4.18
epoch 1, x = -1.0 y = 5.00, dx = 10.00
epoch 2, x = -2.0 y = 20.00, dx = 10.00
epoch 3, x = -3.0 y = 45.00, dx = 10.00
epoch 4, x = -4.0 y = 80.00, dx = 10.00
epoch 5, x = -5.0 y = 125.00, dx = 10.00
epoch 6, x = -6.0 y = 180.00, dx = 10.00
epoch 7, x = -7.0 y = 245.00, dx = 10.00
epoch 8, x = -8.0 y = 320.00, dx = 10.00
epoch 9, x = -9.0 y = 405.00, dx = 10.00
```

```python
def fake_p(x):
    def _g(x, n):
        if x > 0:
            return ((x-n)**2)/(2*lr)
        elif x == 0:
            return x/lr+(n**2)/(2*lr)
        else:
            return _g(x+1, n+1)
    return _g(x,0)
```

```python
x = jax.random.uniform(jax.random.PRNGKey(1))
d_fake_p = jax.grad(fake_p)
lr = 0.1
for i in range(10):
    dx_fake = d_fake_p(x)
    x -= lr * dx_fake
```

© Alex Lew & Mathieu Huot (NeurIPS 2021)

AD may go wrong



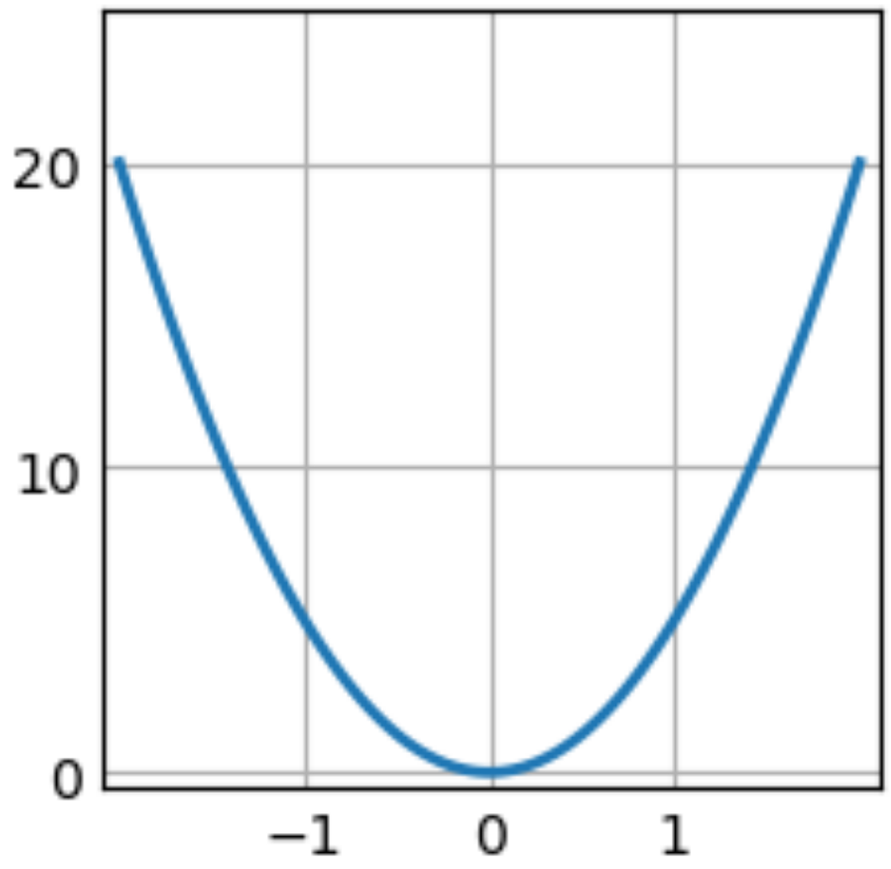

```
epoch 0, x = 0.0 y = 0.00, dx = 4.18
epoch 1, x = -1.0 y = 5.00, dx = 10.00
epoch 2, x = -2.0 y = 20.00, dx = 10.00
epoch 3, x = -3.0 y = 45.00, dx = 10.00
epoch 4, x = -4.0 y = 80.00, dx = 10.00
epoch 5, x = -5.0 y = 125.00, dx = 10.00
epoch 6, x = -6.0 y = 180.00, dx = 10.00
epoch 7, x = -7.0 y = 245.00, dx = 10.00
epoch 8, x = -8.0 y = 320.00, dx = 10.00
epoch 9, x = -9.0 y = 405.00, dx = 10.00
```

```python
def fake_p(x):
    def _g(x, n):
        if x > 0:
            return ((x-n)**2)/(2*lr)
        elif x == 0:
            return x/lr+(n**2)/(2*lr)
        else:
            return _g(x+1, n+1)
    return _g(x,0)
```

```python
x = jax.random.uniform(jax.random.PRNGKey(1))
d_fake_p = jax.grad(fake_p)
lr = 0.1
for i in range(10):
    dx_fake = d_fake_p(x)
    x -= lr * dx_fake
```

© Alex Lew & Mathieu Huot (NeurIPS 2021)

AD may go wrong




```
epoch    x = 0.0 y = 0.00, dx = 4.18
epoch
epoch
epoch
epoch
epoch
epoch
epoch
epoch
epoch
```

Indeed, this is a well-known problem:

Doctoral Thesis

Algorithmisches Differenzieren

Author(s):
Joss, Johann

Publication Date:
1976

Permanent Link:

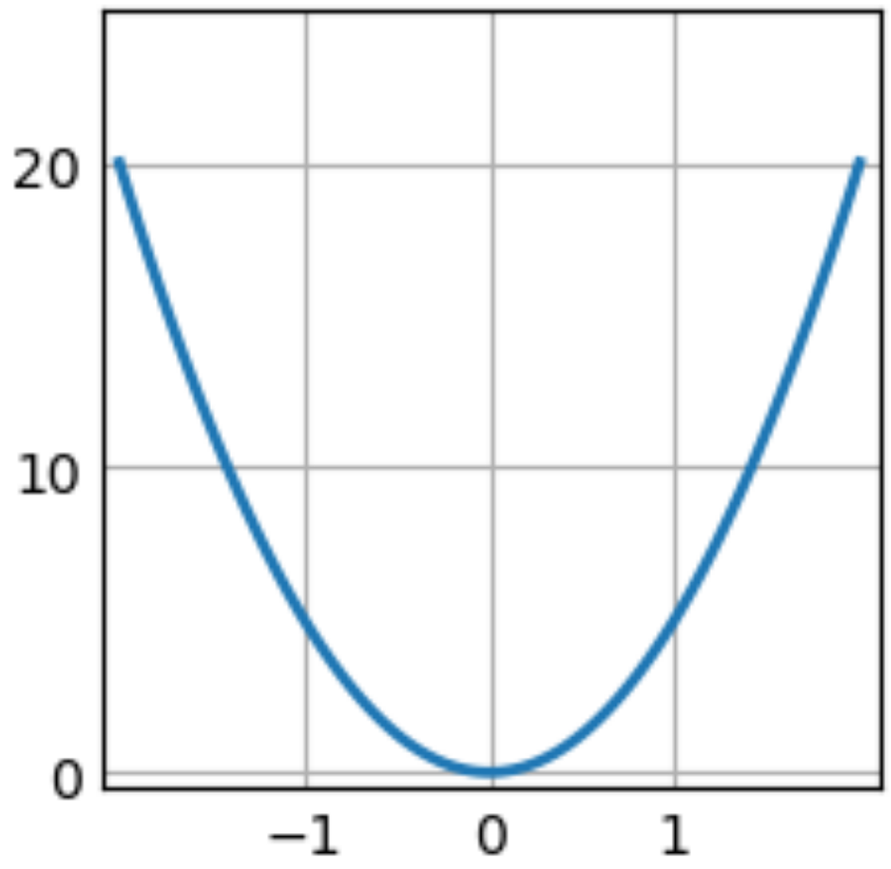
https://doi.org/10.3929/ethz-a-000098736 →

```python
def fake_p(x):
    def _g(x, n):
        if x > 0:
            return ((x-n)**2)/(2*lr)
        elif x == 0:
            return x/lr+(n**2)/(2*lr)
        else:
            return _g(x+1, n+1)
    return _g(x,0)
```

```
x = jax.
d_fake_p
lr = 0.1
for i in
  dx_fak
  x -= l
```

© Alex Lew & Mathieu Huot (NeurIPS 2021)

AD may go wrong




```
def fake_p(x):
    def _g(x, n):
        if x > 0:
            return ((x-n)**2)/(2*lr)
        elif x == 0:
            return x/lr+(n**2)/(2*lr)
        else:
            return _g(x+1, n+1)
    return _g(x,0)
```

```
epoch    x = 0.0 y = 0.00, dx = 4.18
epoch
epoch
epoch
epoch
epoch
epoch
epoch
epoch
epoch
```

```
x = jax.
d_fake_p
lr = 0.1
for i in
  dx_fak
  x -= l
```

Indeed, this is a well-known problem:

Doctoral Thesis

Algorithmisches Differenzieren

Author(s):
Joss, Johann

The set of errors is negligible

Publication Date:
1976

Permanent Link:
https://doi.org/10.3929/ethz-a-000098736 →

© **Alex Lew & Mathieu Huot (NeurIPS 2021)**

Joss Theorem on PCF

Theorem (Mazza, P., 2021)

Suppose the functional symbols of PCF form an **admissible clone**.
Given $x_1 : \mathbb{R}, \ldots, x_n : \mathbb{R} \vdash F : \mathbb{R}$ **PCF term**, then the set :

$$\mathrm{Fail}(F) := \left\{ \mathbf{r} \in \mathrm{diff}([\![F]\!]) \; ; \; \mathrm{grad}(F)(\mathbf{r}) \not\mapsto \nabla [\![F]\!](\mathbf{r}) \right\}$$

is a **quasivariety**, hence negligible.

Joss Theorem on PCF

Theorem (Mazza, P., 2021)

Suppose the functional symbols of PCF form an **admissible clone**.
Given $x_1 : \mathbb{R}, \ldots, x_n : \mathbb{R} \vdash F : \mathbb{R}$ **PCF term**, then the set :

$$\mathrm{Fail}(F) := \left\{ \mathbf{r} \in \mathrm{diff}([\![F]\!]) \; ; \; \mathrm{grad}(F)(\mathbf{r}) \not\mapsto \nabla [\![F]\!](\mathbf{r}) \right\}$$

is a **quasivariety**, hence negligible.

There exists a family of not-identically zero maps $\{g_i\}_{i \in \mathbb{N}}$, composition of functional symbols of PCF and projections, s.t.:

$$\mathrm{Fail}(F) \subseteq \bigcup_{i \in \mathbb{N}} g_i^{-1}(0)$$

Joss Theorem on PCF

Theorem (Mazza, P., 2021)

Suppose the functional symbols of PCF form an **admissible clone**.
Given $x_1 : \mathbb{R}, \ldots, x_n : \mathbb{R} \vdash F : \mathbb{R}$ **PCF term**, then the set :

$$\mathrm{Fail}(F) := \left\{ \mathbf{r} \in \mathrm{diff}([\![F]\!]) \; ; \; \mathrm{grad}(F)(\mathbf{r}) \not\mapsto \nabla [\![F]\!](\mathbf{r}) \right\}$$

is a **quasivariety**, hence negligible.

There exists a family of not-identically zero maps $\{g_i\}_{i \in \mathbb{N}}$, composition of functional symbols of PCF and projections, s.t.:

$$\mathrm{Fail}(F) \subseteq \bigcup_{i \in \mathbb{N}} g_i^{-1}(0)$$

All functions g are:

- continuous on their domain
- if not-identically zero, $g^{-1}(0)$ is negligible

Joss Theorem on PCF

Theorem (Mazza, P., 2021)

Suppose the functional symbols of PCF form an **admissible clone**.
Given $x_1 : \mathbb{R}, \ldots, x_n : \mathbb{R} \vdash F : \mathbb{R}$ **PCF term**, then the set :

$$\mathrm{Fail}(F) := \left\{ \mathbf{r} \in \mathrm{diff}([\![F]\!]) \; ; \; \mathrm{grad}(F)(\mathbf{r}) \not\mapsto \nabla [\![F]\!](\mathbf{r}) \right\}$$

is a **quasivariety**, hence negligible.

There exists a family of not-identically zero maps $\{g_i\}_{i \in \mathbb{N}}$, composition of functional symbols of PCF and projections, s.t.:

$$\mathrm{Fail}(F) \subseteq \bigcup_{i \in \mathbb{N}} g_i^{-1}(0)$$

All functions g are:

- continuous on their domain
- if not-identically zero, $g^{-1}(0)$ is negligible

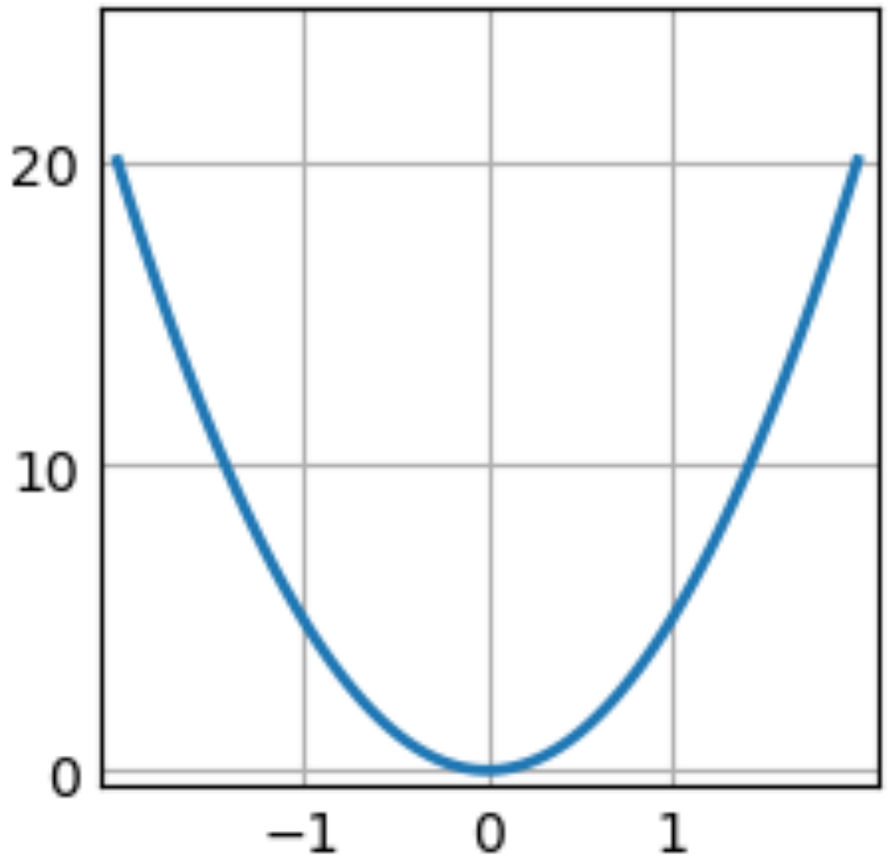
Not too restrictive:

- satisfied by any analytic function
- (non-differentiable) functions definable within PCF, e.g.:

$$\max(x, y) := \texttt{if } x \leq y \texttt{ then } x \texttt{ else } y$$

$$\mathrm{reLU}(x) := \texttt{if } x \leq 0 \texttt{ then } 0 \texttt{ else } x$$

AD may go wrong




```
epoch 0, x = 0.0 y = 0.00, dx = 4.18
epoch 1, x = -1.0 y = 5.00, dx = 10.00
epoch 2, x = -2.0 y = 20.00, dx = 10.00
epoch 3, x = -3.0 y = 45.00, dx = 10.00
epoch 4, x = -4.0 y = 80.00, dx = 10.00
epoch 5, x = -5.0 y = 125.00, dx = 10.00
epoch 6, x = -6.0 y = 180.00, dx = 10.00
epoch 7, x = -7.0 y = 245.00, dx = 10.00
epoch 8, x = -8.0 y = 320.00, dx = 10.00
epoch 9, x = -9.0 y = 405.00, dx = 10.00
```

```python
def fake_p(x):
    def _g(x, n):
        if x > 0:
            return ((x-n)**2)/(2*lr)
        elif x == 0:
            return x/lr+(n**2)/(2*lr)
        else:
            return _g(x+1, n+1)
    return _g(x,0)
```

```python
x = jax.random.uniform(jax.random.PRNGKey(1))
d_fake_p = jax.grad(fake_p)
lr = 0.1
for i in range(10):
    dx_fake = d_fake_p(x)
    x -= lr * dx_fake
```

© Alex Lew & Mathieu Huot (NeurIPS 2021)

- Pearlmutter, Siskind, *Reverse-mode AD in a functional framework: lambda the ultimate backpropagator,* ACM Trans. 2008
- Elliott, *The simple essence of automatic differentiation*, ICFP 2018
- Wang, Zheng, Decker, Wu, Essertel, Rompf, *Demystifying differentiable programming: shift/reset the penultimate backpropagator,* ICFP 2019
- Abadi, Plotkin, *A simple differentiable programming language,* POPL 2020
- Barthe, Crubillé, Dal Lago, Gavazzo, *On the versatility of open logical relations,* ESOP 2020
- Brunel, Mazza, P., *Backpropagation in simply typed lambda-calculus with linear negation,* POPL 2020
- Huot, Staton, Vákár, *Correctness of automatic differentiation via diffeologiens and categorical gluing,* FOSSACS 2020
- Lee, Yu, Rival, Yang, *On correctness of Automatic Differentiation for non-differentiable functions,* NEURIPS 2020
- Mazza, P., *Automatic Differentiation in PCF*. POPL 2021
- Vákár, *Reverse AD at Higher Types: Pure, Principled and Denotationally Correct.* ESOP 2021
- Lew, Huot, Mansinghka, *Towards Denotational Semantics of AD for Higher-Order, Recursive, Probabilistic Languages,* NEURIPS 2021
- Krawiec, Jones, Krishnaswami, Ellis, Eisenberg, Fitzgibbon, *Provably Correct, Asymptotically Efficient, Higher-Order Reverse-Mode Automatic Differentiation.* POPL 2022
- Smedig, Vákár, *Efficient Dual-Numbers Reverse AD via Well-Known Program Transformations.* POPL 2023
- Frostig, Johnson, MacLaurin, Paszke, Radul, *You Only Linearise Once: Tangents Transpose to Gradients*, POPL 2023