

**PROVABLY CORRECT,
ASYMPTOTICALLY EFFICIENT,
HIGHER-ORDER,
REVERSE-MODE
AUTOMATIC DIFFERENTIATION**

**ACM
POPL
2022**

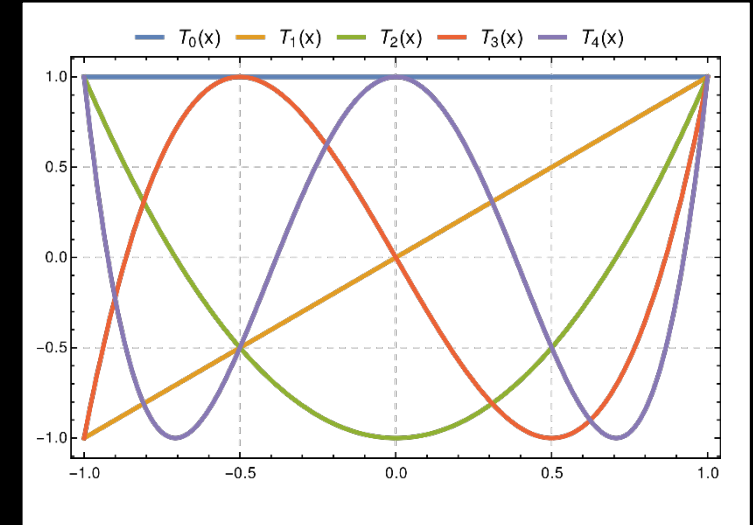
Faustyna Krawiec | University of Cambridge
Simon Peyton Jones | Epic Games/University of Cambridge*
Neel Krishnaswami | University of Cambridge
Tom Ellis | Groq*
Richard Eisenberg | Tweag
Andrew Fitzgibbon | Graphcore*

Why are we here?

■ Before deep learning

$$f(x) \approx \sum_i w_i \phi_i(x)$$

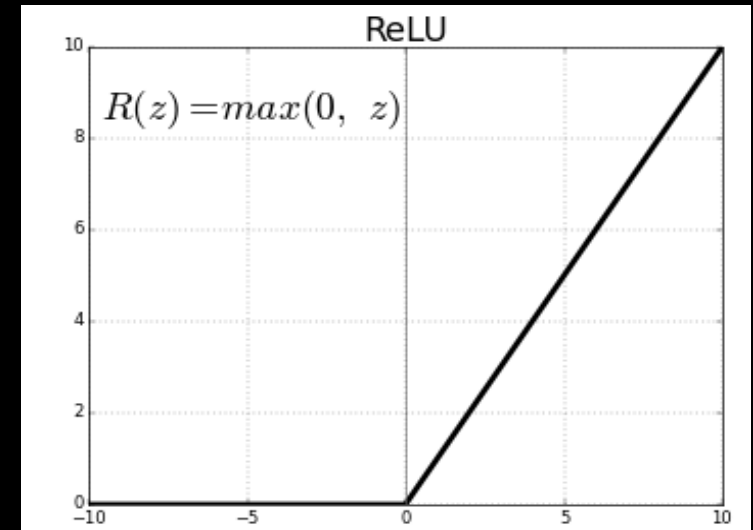
smooth functions, closed-form solutions, grist to analysis



■ After deep learning

$$f(x) \approx f_L \circ f_{L-1} \circ \dots \circ f_1(x) \quad f_i(a) = \sum_j w_{ij} \max(a_j, 0)$$

deep recursion, nonsmooth, grist to gradient descent



Automatic differentiation

Given: computer code for

$$f :: \mathbb{R}^N \rightarrow \mathbb{R}^M$$

Produce: code that computes the Jacobian

$$J_f :: \mathbb{R}^N \rightarrow \mathbb{R}^{M \times N}$$

Or variations thereof:

$$f' :: (\mathbb{R}^N, \mathbb{R}^N) \rightarrow \mathbb{R}^M$$

$$f'(x, dx) = J_f \cdot dx$$

Forward

$$f^{\backslash} :: (\mathbb{R}^N, \mathbb{R}^M) \rightarrow \mathbb{R}^N$$

$$f^{\backslash}(x, df) = J_f^{\top} \cdot df$$

Reverse

And for the important case of $M = 1$

$$\nabla f :: \mathbb{R}^N \rightarrow \mathbb{R}^N$$

$$\nabla f(x) = f^{\backslash}(x, 1)$$

Gradient

Automatic differentiation (arbitrary datatypes)

Given: computer code for

$$f :: S \rightarrow T$$

S and T essentially containers
of “reals + other stuff”

Produce: code that computes the Jacobian

$$J_f :: S \rightarrow (dS \multimap dT)$$

dS the tangent space of S

$A \multimap B$ is the type of linear maps from A to B

Or variations thereof:

Equipped with “apply” \odot and “transpose” \cdot^\top

$$f' :: (S, dS) \rightarrow dT$$

$$f'(x, dx) = J_f \odot dx$$

Forward

$$f^\wedge :: (S, dT) \rightarrow dS$$

$$f^\wedge(x, df) = J_f^\top \odot df$$

Reverse

And for the important case of $M = 1$, the **gradient**

$$\nabla f :: S \rightarrow dS$$

$$\nabla f(x) = f^\wedge(x, 1)$$

Gradient

Automatic differentiation

Given: computer code for

$$f :: \mathbb{R}^N \rightarrow \mathbb{R}^M$$

Produce: code that computes the Jacobian

$$J_f :: \mathbb{R}^N \rightarrow \mathbb{R}^{M \times N}$$

Or variations thereof:

$$f' :: (\mathbb{R}^N, \mathbb{R}^N) \rightarrow \mathbb{R}^M$$

$$f'(x, dx) = J_f \cdot dx$$

Forward

$$f^{\backslash} :: (\mathbb{R}^N, \mathbb{R}^M) \rightarrow \mathbb{R}^N$$

$$f^{\backslash}(x, df) = J_f^{\top} \cdot df$$

Reverse

And for the important case of $M = 1$

$$\nabla f :: \mathbb{R}^N \rightarrow \mathbb{R}^N$$

$$\nabla f(x) = f^{\backslash}(x, 1)$$

Gradient

Automatic differentiation

Given: computer code for

$$f :: \mathbb{R}^N \rightarrow \mathbb{R}^M$$

Produce: code that computes the Jacobian

$$J_f :: \mathbb{R}^N \rightarrow \mathbb{R}^{M \times N}$$

Or variations thereof:

$$f' :: (\mathbb{R}^N, d\mathbb{R}^N) \rightarrow d\mathbb{R}^M$$

$$f'(x, dx) = J_f \cdot dx$$

Forward

$$f^{\backslash} :: (\mathbb{R}^N, d\mathbb{R}^M) \rightarrow d\mathbb{R}^N$$

$$f^{\backslash}(x, df) = J_f^{\top} \cdot df$$

Reverse

And for the important case of $M = 1$

$$\nabla f :: \mathbb{R}^N \rightarrow d\mathbb{R}^N$$

$$\nabla f(x) = f^{\backslash}(x, 1)$$

Gradient

Problem

The (huge, hot)
AD literature

- Has many well-written papers and remarkable implementations
- Implementations tend to be operational and stateful: graph construction and mutation, “tapes”, primal traces (aka Wengert lists), derivative traces, perturbation confusion, call/cc, etc. Accompanying papers (when they exist) tend to be dominated by examples
- Principled theory papers tend to lack implementations, and/or are asymptotically slow
- Is often indirect: you write a program that constructs a graph, that is then run/differentiated

Forward Primal Trace

$$\begin{aligned}v_{-1} = x_1 &= 2 \\ v_0 = x_2 &= 5\end{aligned}$$

$$\begin{aligned}v_1 = \ln v_{-1} &= \ln 2 \\ v_2 = v_{-1} \times v_0 &= 2 \times 5\end{aligned}$$

$$\begin{aligned}v_3 = \sin v_0 &= \sin 5 \\ v_4 = v_1 + v_2 &= 0.693 + 10\end{aligned}$$

$$v_5 = v_4 - v_3 = 10.693 - 0.959$$

$$y = v_5 = 11.652$$

Reverse Adjoint (Derivative) Trace

$$\begin{aligned}\bar{x}_1 = \bar{v}_{-1} &= 5.5 \\ \bar{x}_2 = \bar{v}_0 &= 1.716\end{aligned}$$

$$\begin{aligned}\bar{v}_{-1} = \bar{v}_{-1} + \bar{v}_1 \frac{\partial v_1}{\partial v_{-1}} &= \bar{v}_{-1} + \bar{v}_1 / v_{-1} = 5.5 \\ \bar{v}_0 = \bar{v}_0 + \bar{v}_2 \frac{\partial v_2}{\partial v_0} &= \bar{v}_0 + \bar{v}_2 \times v_{-1} = 1.716\end{aligned}$$

$$\bar{v}_{-1} = \bar{v}_2 \frac{\partial v_2}{\partial v_{-1}} = \bar{v}_2 \times v_0 = 5$$

$$\bar{v}_0 = \bar{v}_3 \frac{\partial v_3}{\partial v_0} = \bar{v}_3 \times \cos v_0 = -0.284$$

$$\bar{v}_2 = \bar{v}_4 \frac{\partial v_4}{\partial v_2} = \bar{v}_4 \times 1 = 1$$

$$\bar{v}_1 = \bar{v}_4 \frac{\partial v_4}{\partial v_1} = \bar{v}_4 \times 1 = 1$$

$$\bar{v}_3 = \bar{v}_5 \frac{\partial v_5}{\partial v_3} = \bar{v}_5 \times (-1) = -1$$

$$\bar{v}_4 = \bar{v}_5 \frac{\partial v_5}{\partial v_4} = \bar{v}_5 \times 1 = 1$$

$$\bar{v}_5 = \bar{y} = 1$$

Goals

- Simple, principled
- Works for reverse derivatives
- Works for higher order programs
- Works for functions of type $\mathcal{S} \rightarrow \mathcal{T}$, not just $\mathbb{R}^a \rightarrow \mathbb{R}^b$
- Asymptotically fast
- Provably correct

Dream AD

- Source-to-source
- Optimally efficient for small and large programs
- Supports all language features

```
def f(x: float) -> float:  
    y = x * sin x  
    return y + x*10
```

```
def f'(x: float) -> float:  
    (s,c) = sincos x  
    return x * c + s + 10
```

Dual numbers

The simplest way to do AD

Not very good for reverse AD

But even if you know this, watch keenly...

Forward AD using dual numbers

- Make Float a pair (\mathbb{R} , $d\mathbb{R}$)
- Define operations using simple chain rule

$$(a, da) + (b, db) = (a + b, da + db)$$

$$(a, da) * (b, db) = (a * b, a * db + da * b)$$

$$\sin(a, da) = (\sin(a), da * \cos(a))$$

$$\text{atan2}((a, da), (b, db)) = \left(\text{atan2}(a, b), \frac{b * da - a * db}{a^2 + b^2} \right)$$

- And discard the first part of the result

```
class Float:
    primal: float
    tangent: float

def sin (a: Float) -> Float:
    return Float (
        sin(a.primal),
        a.tangent*cos(a.primal)
    )

def f(x: Float) -> Float:
    y = x * sin x
    return y + x*10

def f' (x: float) -> float:
    xdual = (x, 1.0)
    return snd(f(xdual))
```

Forward AD using dual numbers

- Functions $f :: \mathbb{R}^N \rightarrow \mathbb{R}^M$

- Become $f' :: (\mathbb{R}, d\mathbb{R})^N \rightarrow (\mathbb{R}, d\mathbb{R})^M$

- Easily transformed to

$$f' :: (\mathbb{R}^N, d\mathbb{R}^N) \rightarrow d\mathbb{R}^M$$

- A reasonable way to implement forward derivative, jvp, ...

- ... but notoriously poor for reverse derivatives, and, most importantly, for **gradients**

```
class Float:
    primal: float
    tangent: float

def sin (a: Float) -> Float:
    return Float (
        sin(a.primal),
        a.tangent*cos(a.primal)
    )

def f(x: Float) -> Float:
    y = x * sin x
    return y + x*10

def f' (x: float) -> float:
    xdual = (x, 1.0)
    return f(xdual)2
```

(e)₂ means second element of tuple (e), (e)[1] in Python

Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

- In order to get $\nabla f(x)$, make N calls:

$$\nabla f(x) = \begin{bmatrix} f'(x, [1, \dots, 0]) \\ \vdots \\ f'(x, [0, \dots, 1]) \end{bmatrix}$$

- Hurts if $N = 10^9$

```
def f(x : Float[3]) -> Float:  
  y = x[0] * sin x[1]  
  return y + x[0]*10*x[2]
```

```
def ∇f(x : float[3]) -> float[3]:  
  return [  
    f([(x[0], 1), (x[1], 0), (x[2], 0)])2,  
    f([(x[0], 0), (x[1], 1), (x[2], 0)])2,  
    f([(x[0], 0), (x[1], 0), (x[2], 1)])2  
  ]
```

`(e)2` means second element of tuple `(e)`

And yes... some day we may learn to use fewer probes [Baydin et al], but for now we need gradients

[Aside: can be OK for Jacobians]

- Essentially, forward or reverse derivatives compute a column/row of J .

If you want all of J and

- it's strongly portrait or landscape,
- not sparse

you're probably fine.

Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

- In order to get $\nabla f(x)$, make N calls:

$$\nabla f(x) = \begin{bmatrix} f'(x, [1, \dots, 0]) \\ \vdots \\ f'(x, [0, \dots, 1]) \end{bmatrix}$$

- Hurts if $N = 10^8$

```
def f(x : Float[3]) -> Float:
  y = x[0] * sin x[1]
  return y + x[0]*10*x[2]

def ∇f(x : float[3]) -> float[3]:
  return [
    f([(x[0], 1), (x[1], 0), (x[2], 0)])2
    f([(x[0], 0), (x[1], 1), (x[2], 0)])2
    f([(x[0], 0), (x[1], 0), (x[2], 1)])2
  ]
```

And yes... we may learn to use fewer probes
[Baydin et al]

Fix #1: tupling

Forward AD using dual numbers

- Make Float a pair $(\mathbb{R}, d\mathbb{R})$

- Define operations using simple chain rule

$$(a, da) + (b, db) = (a + b, da + db)$$

$$(a, da) * (b, db) = (a * b, a * db + da * b)$$

$$\sin(a, da) = (\sin(a), da * \cos(a))$$

$$\text{atan2}((a, da), (b, db)) = \left(\text{atan2}(a, b), \frac{b * da - a * db}{a^2 + b^2} \right)$$

- And discard the first part of the result

```
class Float:
    primal: float
    tangent: float

def sin (a: Float) -> Float:
    return Float (
        sin(a.primal),
        a.tangent*cos(a.primal)
    )

def f(x: Float) -> Float:
    y = x * sin x
    return y + x*10

def f' (x: float) -> float:
    xdual = (x, 1.0)
    return snd(f(xdual))
```

Forward AD using dual numbers

- Make Float a pair $(\mathbb{R}, d\mathbb{R})$
- Define operations using simple chain rule

$$+: da + db$$

$$*: a * db + da * b$$

$$\sin: da * \cos(a)$$

```
class Float:
    primal: float
    tangent: float

def sin (a: Float) -> Float:
    return Float (
        sin(a.primal),
        a.tangent*cos(a.primal)
    )
```

$$\text{atan2}: \frac{b}{a^2 + b^2} * da - \frac{a}{a^2 + b^2} * db$$

Forward AD using DualVectors

- Make Float a pair (\mathbb{R}, dS)
- Define operations using simple chain rule

$+$: $dAdd(da, db)$

$*$: $dAdd(dScale(a, db), dScale(b, da))$

\sin : $dScale(\cos(a), da)$

```
interface DualVec:
  dScale: (float, DualVec) -> DualVec
  dAdd: (DualVec, DualVec) -> DualVec
  dZero: DualVec

class Float:
  primal: float
  tangent: DualVec

def sin (a: Float) -> Float:
  return Float (
    sin(a.primal),
    scale(cos(a.primal), a.tangent)
  )
```

atan2 : $dAdd\left(dScale\left(\frac{b}{a^2 + b^2}, da\right), dScale\left(-\frac{a}{a^2 + b^2}, db\right)\right)$

And now the full gradient in one call...

Recall: Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

$$f'(x, dx) = f_2 \left(\begin{bmatrix} x_0, dx_0 \\ \vdots \\ x_n, dx_n \end{bmatrix} \right)$$

- In order to get $\nabla f(x)$, make N calls:

$$\nabla f(x) = \begin{bmatrix} f'(x, [1, \dots, 0]) \\ \vdots \\ f'(x, [0, \dots, 1]) \end{bmatrix}$$

```
def f(x : Float[3]) -> Float:
  y = x[0] * sin x[1]
  return y + x[0]*10*x[2]

def Vf(x : float[3]) -> float[3]:
  return [
    f([(x[0], 1), (x[1], 0), x[2], 0)])_2
    f([(x[0], 0), (x[1], 1), x[2], 0)])_2
    f([(x[0], 0), (x[1], 0), x[2], 1)])_2 ]
```

$$f_2(x) = f(x)[1] = \text{snd}(f(x))$$

Recall: Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

$$f'(x, dx) = f_2 \left(\begin{bmatrix} x_0, dx_0 \\ \vdots \\ x_n, dx_n \end{bmatrix} \right)$$

- In order to get $\nabla f(x)$, make N calls:

$$\nabla f(x) = \left[f_2 \left(\begin{bmatrix} x_0, 1 \\ \vdots \\ x_N, 0 \end{bmatrix} \right), \dots, f_2 \left(\begin{bmatrix} x_0, 0 \\ \vdots \\ x_N, 1 \end{bmatrix} \right) \right]$$

```
def f(x : Float[3]) -> Float:
  y = x[0] * sin x[1]
  return y + x[0]*10*x[2]

def Vf(x : float[3]) -> float[3]:
  return [
    f([(x[0], 1), (x[1], 0), (x[2], 0)])2
    f([(x[0], 0), (x[1], 1), (x[2], 0)])2
    f([(x[0], 0), (x[1], 0), (x[2], 1)])2 ]
```

Recall: Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

$$f'(x, dx) = f_2 \left(\begin{bmatrix} x_0, dx_0 \\ \vdots \\ x_n, dx_n \end{bmatrix} \right)$$

- In order to get $\nabla f(x)$, make 1 call:

$$\nabla f(x) = f_2 \left(\begin{bmatrix} x_0, [1, \dots, 0] \\ \vdots \\ x_N, [0, \dots, 1] \end{bmatrix} \right)$$

- Hooray! $1 \ll N!$ But still obviously $O(N^2)$...

```
def f(x : Float[3]) -> Float:
  y = x[0] * sin x[1]
  return y + x[0]*10*x[2]

def Vf(x : float[3]) -> float[3]:
  return
    f((x[0], [1], [0]), (x[1], [0], [0]), (x[2], [0], [1]))_2
    [0] [0] [1]
```

[Not real Python syntax ☺]

Recall: Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

$$f'(x, dx) = f_2 \left(\begin{bmatrix} x_0, dx_0 \\ \vdots \\ x_n, dx_n \end{bmatrix} \right)$$

- In order to get $\nabla f(x)$, make 1 call:

$$\nabla f(x) = f_2 \left(\begin{bmatrix} x_0, [1, \dots, 0] \\ \vdots \\ x_N, [0, \dots, 1] \end{bmatrix} \right)$$

- Hooray! $1 \ll N!$ But still obviously $O(N^2)$...

```
def f(x : Float[3]) -> Float:  
  y = x[0] * sin x[1]  
  return y + x[0]*10*x[2]
```

```
def
```

So use sparse
matrices...
problem solved?

```
[0]  
[2], [0])])2  
[1]
```



Recall: Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

$$f'(x, dx) = f_2 \left(\begin{bmatrix} x_0, dx_0 \\ \vdots \\ x_n, dx_n \end{bmatrix} \right)$$

- In order to get $\nabla f(x)$, make 1 call:

$$\nabla f(x) = f_2 \left(\begin{bmatrix} x_0, \text{sparseOneHot}(1, N) \\ \vdots \\ x_N, \text{sparseOneHot}(N, N) \end{bmatrix} \right)$$

- Hooray! $1 \ll N!$ But still obviously $O(N^2)$...

```
def f(x : Float[3]) -> Float:  
  y = x[0] * sin x[1]  
  return y + x[0]*10*x[2]
```

```
def
```

So use sparse
matrices...
problem solved?

```
[0]  
[2], [0])])2  
[1]
```



Recall: Terrible for gradients

- We only have

$$f'(x, dx) = \nabla f(x) \cdot dx$$

Not quite... they start sparse,
but end up full*.

But... bear with us while we
introduce a funny kind of sparse
vector...

- In

$$\nabla f(x) = f_2 \left(\begin{bmatrix} x_0, \text{sparseOneHot}(1, N) \\ \vdots \\ x_N, \text{sparseOneHot}(N, N) \end{bmatrix} \right)$$

- Hooray! $1 \ll N!$ But still obviously $O(N^2)$...

* and if there's one thing worse than a $10^8 \times 10^8$ dense matrix,
it's a $10^8 \times 10^8$ sparse matrix which is mostly full.

```
def f(x : Float[3]) -> Float:  
  y = x[0] * sin x[1]  
  return y + x[0]*10*x[2]
```

```
def
```

So use sparse
matrices...
problem solved?

```
[0]  
[2], [0])])2  
[1]
```



Fix #2: Symbolic sparse vectors

Record constructors, rather than actually constructing

Forward AD using DualVectors

- Make Float a pair (\mathbb{R}, dS)
- Define operations using simple chain rule

$+$: $dAdd(da, db)$

$*$: $dAdd(dScale(a, db), dScale(b, da))$

\sin : $dScale(\cos(a), da)$

```
interface DualVec:
  dScale: (float, DualVec) -> DualVec
  dAdd: (DualVec, DualVec) -> DualVec
  dZero: DualVec
  dDot: (DualVec, DualVec) -> float

class Float:
  primal: float
  tangent: DualVec

def sin (a: Float) -> Float:
  return Float (
    sin(a.primal),
    scale(cos(a.primal), a.tangent)
  )
```

atan2 : $dAdd\left(dScale\left(\frac{b}{a^2 + b^2}, da\right), dScale\left(-\frac{a}{a^2 + b^2}, db\right)\right)$

Forward AD using DualVectors

- Make Float a pair (\mathbb{R}, dS)
- Define operations using simple chain rule

$+$: $dAdd(da, db)$

$*$: $dAdd(dScale(a, db), dScale(b, da))$

\sin : $dScale(\cos(a), da)$

```
interface DualVec:
  dScale: (float, DualVec) -> DualVec
  dAdd: (DualVec, DualVec) -> DualVec
  dZero: DualVec
  dOneHot: (i:int, n:int) -> DualVec
  dDot: (DualVec, DualVec) -> float

class Float:
  primal: float
  tangent: DualVec

def sin (a: Float) -> Float:
  return Float (
    sin(a.primal),
    scale(cos(a.primal), a.tangent)
  )
```

atan2 : $dAdd\left(dScale\left(\frac{b}{a^2 + b^2}, da\right), dScale\left(-\frac{a}{a^2 + b^2}, db\right)\right)$

Lazy DualVecs

- In forward mode, start with dx vector in $f'(x, dx)$ and DualVec interfaces just operate directly
- In Lazy mode, DualVec instances are just constructors, so after running the program, we get not a vector but a tree dv
- Need a function $eval(dv)$ to turn it into a dense Vec
- All ops constant time

```
interface DualVec:
  dScale: (float, DualVec) -> DualVec
  dAdd: (DualVec, DualVec) -> DualVec
  dZero: DualVec
  dOneHot: (i:int, n:int) -> DualVec
  dDot: (DualVec, DualVec) -> float

class Scale(DualVec):
  s: float
  dv: DualVec

class Add(DualVec):
  da: DualVec
  db: DualVec

class OneHot(DualVec):
  i:int
  n:int
```

- This is just a symbolic sparseVector representation.

- How might you represent a sparse vector?

Type SparseVec = List[int,float]

- SparseVec([2,1.1],[17,2.2]) == Add(Scale(1.1,OneHot(2)), Scale(2.2, OneHot(17)))

interface DualVec:	Storage
dScale: (float, DualVec) -> DualVec	4 bytes
dAdd: (DualVec, DualVec) -> DualVec	0 bytes
dZero: DualVec	0 bytes
dOneHot: (i:int, n:int) -> DualVec	8 bytes
dDot: (DualVec, DualVec) -> float	

- Essentially equally efficient

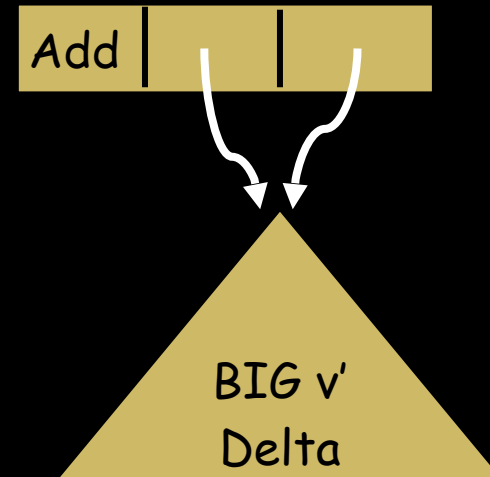
- But now, get lazy...

Fix #3: sharing

Job done? Not so fast...

```
def f(x):  
    y = <very-big-expr> # Returns a big LazyDV  
    return y + y
```

- We will call f @ (Dual Delta)
- So $y::\text{Dual Delta} = D v v'$ where $v': \text{Delta}$ may be VERY BIG
- Then $(y+y) = D (v + v)$ (*Add v'v'*)
NB: v' is shared
- BUT **ALAS** eval can't see the sharing and so will traverse BIG Delta twice



Catastrophic loss of sharing

- BUT ALAS eval can't see the sharing and so will traverse BIG Delta twice
- This can be *asymptotically* bad

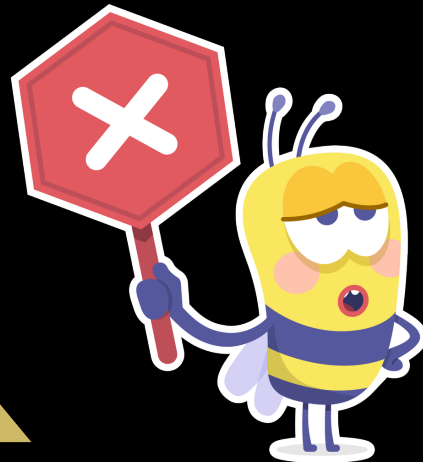
```
f :: Num a => Vec a -> a
f x = let y1 = <rhs>
        y2 = y1+y1
        y3 = y2+y2
        ...
        y100 = y99+y99
    in y100
```

- A linear sized graph unravels to an *exponentially* larger tree

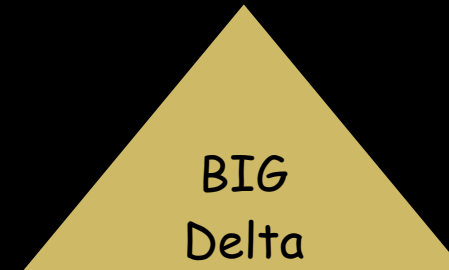
The fix: make sharing explicit

```
data Delta = Zero
           | OneHot Int
           | Scale Float Delta
           | Add Delta Delta
           | Var DeltaId
           | Let DeltaId Delta Delta
type DeltaId = Int
eval :: (Int,Int) -> Delta -> Vec Float'
```

Add | |

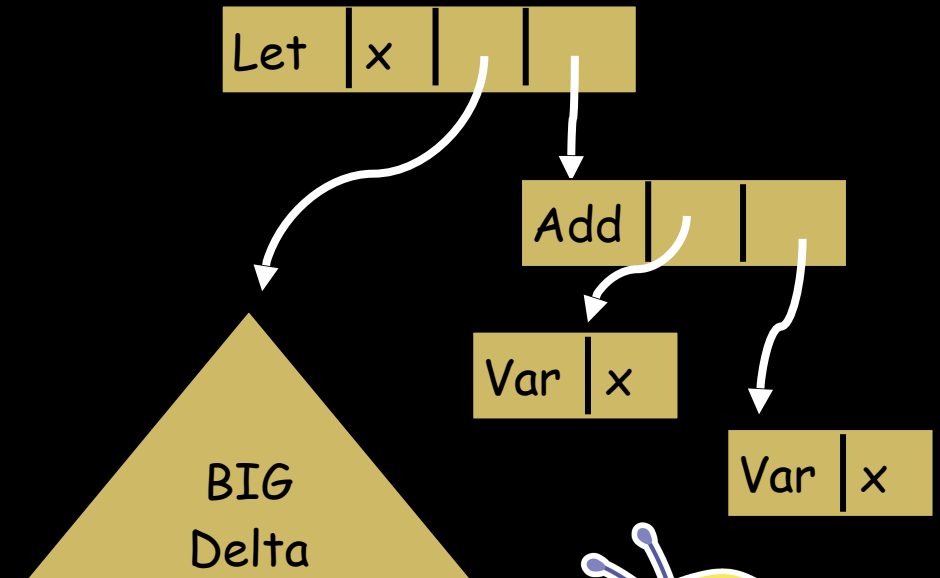


Let | x | |



The fix: make sharing explicit

```
data Delta = Zero
           | OneHot Int
           | Scale Float Delta
           | Add Delta Delta
           | Var DeltaId
           | Let DeltaId Delta Delta
type DeltaId = Int
eval :: (Int,Int) -> Delta -> Vec Float'
```



- And eval is now efficient
- Evaluates the RHS of a let just once!
- Can still use update-in-place
- Side note: evaluate Let “backwards” [see paper]



Does it work?

...i.e. does it correctly compute derivatives?

Yes: we have a proof (see the paper)

7 CORRECTNESS

The property we would like to establish about our reverse mode translation is given in Figure 2. Specializing it to the case of $\mathbb{R}^a \rightarrow \mathbb{R}$, we get the following statement:

THEOREM 6 (CORRECTNESS OF $\mathcal{R}\{e\}$).

If e is a closed term of type $\mathbb{R}^a \rightarrow \mathbb{R}$ then for all $s : \mathbb{R}^a$, and $\delta t : \mathbb{R}$, $[[\mathcal{R}\{e\}]](s, \delta t) = \delta t^\top \bullet \mathcal{J}[[e]](s)$.

Does it work fast?

- Every Float turns into a (D Float Delta)
- Each primop allocates a constant amount of Delta stuff – and nothing else does.
- eval runs in $O(\text{size Delta})$

Bottom line: runtime is only a **constant factor** worse than the original program – and that too is easy to prove

```

module Program (R : Real) = struct
  open R -- access real from the module R
          -- along with arithmetic operations

  type vec3 = { x : real; y : real; z : real }

  type quaternion =
    { x : real; y : real; z : real; w : real }

  let q_to_vec (q : quaternion) : vec3 =
    { x = q.x; y = q.y; z = q.z }

  let dot (p : vec3) (q : vec3) : real =
    p.x × q.x + p.y × q.y + p.z × q.z
    -- Vector addition

  let (++) (p : vec3) (q : vec3) : vec3 =
    { x = p.x + q.x; y = p.y + q.y; z = p.z + q.z }

  let scale k (v : vec3) : vec3 =
    { x = k × v.x; y = k × v.y; z = k × v.z }

  let cross (a : vec3) (b : vec3) : vec3 =
    { x = a.y × b.z - a.z × b.y;
      y = a.z × b.x - a.x × b.z;
      z = a.x × b.y - a.y × b.x }

  let norm (x : vec3) : real = sqrt (dot x x)

  let rotate_vec_by_quat (v : vec3)
                        (q : quaternion) : vec3 =

    let u = q_to_vec q in
    let s = q.w in
    scale (from_float 2.0 × dot u v) u
    ++
    scale (s × s - dot u u) v
    ++
    scale (from_float 2.0 × s) (cross u v)

```

end

```

-- Result of runDelta 8 (D{e} s0), where
-- e = λq v. (rotate_vec_by_quat v q).x
-- s0 = (q0; v0)
-- q0 = { (1.1; Var qx); (2.2; Var qy); (3.3; Var qz); (4.4; Var qw) }
-- v0 = { (5.5; Var vx); (6.6; Var vy); (7.7; Var vz) }

-- We informally use a LetIn notation for the constructor Let, and
-- use variable names instead of numbers, so xN stands for variable N+8

Let x1 = Add (Scale 5.5 (Var qy)) (Scale 2.2 (Var vx)) in
Let x2 = Add (Scale 6.6 (Var qx)) (Scale 1.1 (Var vy)) in
Let x3 = Add (Var x2) (Scale (-1.0) (Var x1)) in
Let x4 = Add (Scale 7.7 (Var qx)) (Scale 1.1 (Var vz)) in
Let x5 = Add (Scale 5.5 (Var qz)) (Scale 3.3 (Var vx)) in
Let x6 = Add (Var x5) (Scale (-1.0) (Var x4)) in
Let x7 = Add (Scale 6.6 (Var qz)) (Scale 3.3 (Var vy)) in
Let x8 = Add (Scale 7.7 (Var qy)) (Scale 2.2 (Var vz)) in
Let x9 = Add (Var x8) (Scale (-1.0) (Var x7)) in
Let x10 = Zero in
Let x11 = Add (Scale 4.4 (Var x10)) (Scale 2 (Var qw)) in
Let x12 = Add (Scale (-4.84) (Var x11)) (Scale 8.8 (Var x3)) in
Let x13 = Add (Scale 9.68 (Var x11)) (Scale 8.8 (Var x6)) in
Let x14 = Add (Scale (-4.84) (Var x11)) (Scale 8.8 (Var x9)) in
Let x15 = Add (Scale 3.3 (Var qz)) (Scale 3.3 (Var qz)) in
Let x16 = Add (Scale 2.2 (Var qy)) (Scale 2.2 (Var qy)) in
Let x17 = Add (Scale 1.1 (Var qx)) (Scale 1.1 (Var qx)) in
Let x18 = Add (Var x17) (Var x16) in
Let x19 = Add (Var x18) (Var x15) in
Let x20 = Add (Scale 4.4 (Var qw)) (Scale 4.4 (Var qw)) in
Let x21 = Add (Var x20) (Scale (-1.0) (Var x19)) in
Let x22 = Add (Scale 7.7 (Var x21)) (Scale 2.42 (Var vz)) in
Let x23 = Add (Scale 6.6 (Var x21)) (Scale 2.42 (Var vy)) in
Let x24 = Add (Scale 5.5 (Var x21)) (Scale 2.42 (Var vx)) in
Let x25 = Add (Scale 7.7 (Var qz)) (Scale 3.3 (Var vz)) in
Let x26 = Add (Scale 6.6 (Var qy)) (Scale 2.2 (Var vy)) in
Let x27 = Add (Scale 5.5 (Var qx)) (Scale 1.1 (Var vx)) in
Let x28 = Add (Var x27) (Var x26) in
Let x29 = Add (Var x28) (Var x25) in
Let x30 = Zero in
Let x31 = Add (Scale 45.98 (Var x30)) (Scale 2 (Var x29)) in
Let x32 = Add (Scale 3.3 (Var x31)) (Scale 91.96 (Var qz)) in
Let x33 = Add (Scale 2.2 (Var x31)) (Scale 91.96 (Var qy)) in
Let x34 = Add (Scale 1.1 (Var x31)) (Scale 91.96 (Var qx)) in
Let x35 = Add (Var x32) (Var x22) in
Let x36 = Add (Var x33) (Var x23) in
Let x37 = Add (Var x34) (Var x24) in
Let x38 = Add (Var x35) (Var x12) in
Let x39 = Add (Var x36) (Var x13) in
Let x40 = Add (Var x37) (Var x14) in
Var x40

```

This is the Delta we get

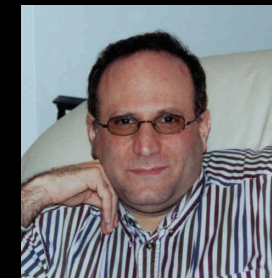
Is it “new”? No...



Edward Kmett



Barak Pearlmutter



Jeffrey Siskind

Hackage :: [Package] Search · Browse

ad: Automatic Differentiation

[[bsd3](#), [library](#), [math](#)] [[Propose Tags](#)]

Forward-, reverse- and mixed- mode automatic differentiation combinators with a common API.

Type-level "branding" is used to both prevent the end user from confusing infinitesimals and to limit unsafe access to the implementation details of each Mode.

Each mode has a separate module full of combinators.

- `Numeric.AD.Mode.Forward` provides basic forward-mode AD. It is good for computing simple derivatives.
- `Numeric.AD.Mode.Reverse` uses benign side-effects to compute reverse-mode AD. It is good for computing gradients in one pass. It generates a Wengert list (linear tape) using `Data.Reflection`.

Versions

[\[RSS\]](#) [\[faq\]](#)
0.12, 0.13, 0.15, 0.16, 0.17, 0.18, 0.19, 0.20, 0.21, 0.22, 0.23, 0.24, 0.25, 0.26, 0.27, 0.28, 0.29, 0.30, 0.31, 0.32.0, 0.33.0, 0.40.0, 0.41.0, 0.42.0, 0.43.0, 0.44.0, 0.44.1, 0.44.2, 0.44.3, 0.44.4, 0.45.0, 0.46.0, 0.46.1, 0.46.2, 0.47.0, 1.0.0, 1.0.1, 1.0.2, 1.0.3, 1.0.4, 1.0.5, 1.0.6, 1.1.0, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.1.6, 1.1.7, 1.1.8, 1.1.9, 1.2.0, 1.2.0.1, 1.2.0.2, 1.2.0.3, 1.2.0.4, 1.2.0.5, 1.2.0.6, 1.2.0.7, 1.2.0.8, 1.2.0.9, 1.2.0.10, 1.2.0.11, 1.2.0.12, 1.2.0.13, 1.2.0.14, 1.2.0.15, 1.2.0.16, 1.2.0.17, 1.2.0.18, 1.2.0.19, 1.2.0.20, 1.2.0.21, 1.2.0.22, 1.2.0.23, 1.2.0.24, 1.2.0.25, 1.2.0.26, 1.2.0.27, 1.2.0.28, 1.2.0.29, 1.2.0.30, 1.2.0.31, 1.2.0.32, 1.2.0.33, 1.2.0.34, 1.2.0.35, 1.2.0.36, 1.2.0.37, 1.2.0.38, 1.2.0.39, 1.2.0.40, 1.2.0.41, 1.2.0.42, 1.2.0.43, 1.2.0.44, 1.2.0.45, 1.2.0.46, 1.2.0.47, 1.2.0.48, 1.2.0.49, 1.2.0.50, 1.2.0.51, 1.2.0.52, 1.2.0.53, 1.2.0.54, 1.2.0.55, 1.2.0.56, 1.2.0.57, 1.2.0.58, 1.2.0.59, 1.2.0.60, 1.2.0.61, 1.2.0.62, 1.2.0.63, 1.2.0.64, 1.2.0.65, 1.2.0.66, 1.2.0.67, 1.2.0.68, 1.2.0.69, 1.2.0.70, 1.2.0.71, 1.2.0.72, 1.2.0.73, 1.2.0.74, 1.2.0.75, 1.2.0.76, 1.2.0.77, 1.2.0.78, 1.2.0.79, 1.2.0.80, 1.2.0.81, 1.2.0.82, 1.2.0.83, 1.2.0.84, 1.2.0.85, 1.2.0.86, 1.2.0.87, 1.2.0.88, 1.2.0.89, 1.2.0.90, 1.2.0.91, 1.2.0.92, 1.2.0.93, 1.2.0.94, 1.2.0.95, 1.2.0.96, 1.2.0.97, 1.2.0.98, 1.2.0.99, 1.2.0.100, 1.2.0.101, 1.2.0.102, 1.2.0.103, 1.2.0.104, 1.2.0.105, 1.2.0.106, 1.2.0.107, 1.2.0.108, 1.2.0.109, 1.2.0.110, 1.2.0.111, 1.2.0.112, 1.2.0.113, 1.2.0.114, 1.2.0.115, 1.2.0.116, 1.2.0.117, 1.2.0.118, 1.2.0.119, 1.2.0.120, 1.2.0.121, 1.2.0.122, 1.2.0.123, 1.2.0.124, 1.2.0.125, 1.2.0.126, 1.2.0.127, 1.2.0.128, 1.2.0.129, 1.2.0.130, 1.2.0.131, 1.2.0.132, 1.2.0.133, 1.2.0.134, 1.2.0.135, 1.2.0.136, 1.2.0.137, 1.2.0.138, 1.2.0.139, 1.2.0.140, 1.2.0.141, 1.2.0.142, 1.2.0.143, 1.2.0.144, 1.2.0.145, 1.2.0.146, 1.2.0.147, 1.2.0.148, 1.2.0.149, 1.2.0.150, 1.2.0.151, 1.2.0.152, 1.2.0.153, 1.2.0.154, 1.2.0.155, 1.2.0.156, 1.2.0.157, 1.2.0.158, 1.2.0.159, 1.2.0.160, 1.2.0.161, 1.2.0.162, 1.2.0.163, 1.2.0.164, 1.2.0.165, 1.2.0.166, 1.2.0.167, 1.2.0.168, 1.2.0.169, 1.2.0.170, 1.2.0.171, 1.2.0.172, 1.2.0.173, 1.2.0.174, 1.2.0.175, 1.2.0.176, 1.2.0.177, 1.2.0.178, 1.2.0.179, 1.2.0.180, 1.2.0.181, 1.2.0.182, 1.2.0.183, 1.2.0.184, 1.2.0.185, 1.2.0.186, 1.2.0.187, 1.2.0.188, 1.2.0.189, 1.2.0.190, 1.2.0.191, 1.2.0.192, 1.2.0.193, 1.2.0.194, 1.2.0.195, 1.2.0.196, 1.2.0.197, 1.2.0.198, 1.2.0.199, 1.2.0.200, 1.2.0.201, 1.2.0.202, 1.2.0.203, 1.2.0.204, 1.2.0.205, 1.2.0.206, 1.2.0.207, 1.2.0.208, 1.2.0.209, 1.2.0.210, 1.2.0.211, 1.2.0.212, 1.2.0.213, 1.2.0.214, 1.2.0.215, 1.2.0.216, 1.2.0.217, 1.2.0.218, 1.2.0.219, 1.2.0.220, 1.2.0.221, 1.2.0.222, 1.2.0.223, 1.2.0.224, 1.2.0.225, 1.2.0.226, 1.2.0.227, 1.2.0.228, 1.2.0.229, 1.2.0.230, 1.2.0.231, 1.2.0.232, 1.2.0.233, 1.2.0.234, 1.2.0.235, 1.2.0.236, 1.2.0.237, 1.2.0.238, 1.2.0.239, 1.2.0.240, 1.2.0.241, 1.2.0.242, 1.2.0.243, 1.2.0.244, 1.2.0.245, 1.2.0.246, 1.2.0.247, 1.2.0.248, 1.2.0.249, 1.2.0.250, 1.2.0.251, 1.2.0.252, 1.2.0.253, 1.2.0.254, 1.2.0.255, 1.2.0.256, 1.2.0.257, 1.2.0.258, 1.2.0.259, 1.2.0.260, 1.2.0.261, 1.2.0.262, 1.2.0.263, 1.2.0.264, 1.2.0.265, 1.2.0.266, 1.2.0.267, 1.2.0.268, 1.2.0.269, 1.2.0.270, 1.2.0.271, 1.2.0.272, 1.2.0.273, 1.2.0.274, 1.2.0.275, 1.2.0.276, 1.2.0.277, 1.2.0.278, 1.2.0.279, 1.2.0.280, 1.2.0.281, 1.2.0.282, 1.2.0.283, 1.2.0.284, 1.2.0.285, 1.2.0.286, 1.2.0.287, 1.2.0.288, 1.2.0.289, 1.2.0.290, 1.2.0.291, 1.2.0.292, 1.2.0.293, 1.2.0.294, 1.2.0.295, 1.2.0.296, 1.2.0.297, 1.2.0.298, 1.2.0.299, 1.2.0.300, 1.2.0.301, 1.2.0.302, 1.2.0.303, 1.2.0.304, 1.2.0.305, 1.2.0.306, 1.2.0.307, 1.2.0.308, 1.2.0.309, 1.2.0.310, 1.2.0.311, 1.2.0.312, 1.2.0.313, 1.2.0.314, 1.2.0.315, 1.2.0.316, 1.2.0.317, 1.2.0.318, 1.2.0.319, 1.2.0.320, 1.2.0.321, 1.2.0.322, 1.2.0.323, 1.2.0.324, 1.2.0.325, 1.2.0.326, 1.2.0.327, 1.2.0.328, 1.2.0.329, 1.2.0.330, 1.2.0.331, 1.2.0.332, 1.2.0.333, 1.2.0.334, 1.2.0.335, 1.2.0.336, 1.2.0.337, 1.2.0.338, 1.2.0.339, 1.2.0.340, 1.2.0.341, 1.2.0.342, 1.2.0.343, 1.2.0.344, 1.2.0.345, 1.2.0.346, 1.2.0.347, 1.2.0.348, 1.2.0.349, 1.2.0.350, 1.2.0.351, 1.2.0.352, 1.2.0.353, 1.2.0.354, 1.2.0.355, 1.2.0.356, 1.2.0.357, 1.2.0.358, 1.2.0.359, 1.2.0.360, 1.2.0.361, 1.2.0.362, 1.2.0.363, 1.2.0.364, 1.2.0.365, 1.2.0.366, 1.2.0.367, 1.2.0.368, 1.2.0.369, 1.2.0.370, 1.2.0.371, 1.2.0.372, 1.2.0.373, 1.2.0.374, 1.2.0.375, 1.2.0.376, 1.2.0.377, 1.2.0.378, 1.2.0.379, 1.2.0.380, 1.2.0.381, 1.2.0.382, 1.2.0.383, 1.2.0.384, 1.2.0.385, 1.2.0.386, 1.2.0.387, 1.2.0.388, 1.2.0.389, 1.2.0.390, 1.2.0.391, 1.2.0.392, 1.2.0.393, 1.2.0.394, 1.2.0.395, 1.2.0.396, 1.2.0.397, 1.2.0.398, 1.2.0.399, 1.2.0.400, 1.2.0.401, 1.2.0.402, 1.2.0.403, 1.2.0.404, 1.2.0.405, 1.2.0.406, 1.2.0.407, 1.2.0.408, 1.2.0.409, 1.2.0.410, 1.2.0.411, 1.2.0.412, 1.2.0.413, 1.2.0.414, 1.2.0.415, 1.2.0.416, 1.2.0.417, 1.2.0.418, 1.2.0.419, 1.2.0.420, 1.2.0.421, 1.2.0.422, 1.2.0.423, 1.2.0.424, 1.2.0.425, 1.2.0.426, 1.2.0.427, 1.2.0.428, 1.2.0.429, 1.2.0.430, 1.2.0.431, 1.2.0.432, 1.2.0.433, 1.2.0.434, 1.2.0.435, 1.2.0.436, 1.2.0.437, 1.2.0.438, 1.2.0.439, 1.2.0.440, 1.2.0.441, 1.2.0.442, 1.2.0.443, 1.2.0.444, 1.2.0.445, 1.2.0.446, 1.2.0.447, 1.2.0.448, 1.2.0.449, 1.2.0.450, 1.2.0.451, 1.2.0.452, 1.2.0.453, 1.2.0.454, 1.2.0.455, 1.2.0.456, 1.2.0.457, 1.2.0.458, 1.2.0.459, 1.2.0.460, 1.2.0.461, 1.2.0.462, 1.2.0.463, 1.2.0.464, 1.2.0.465, 1.2.0.466, 1.2.0.467, 1.2.0.468, 1.2.0.469, 1.2.0.470, 1.2.0.471, 1.2.0.472, 1.2.0.473, 1.2.0.474, 1.2.0.475, 1.2.0.476, 1.2.0.477, 1.2.0.478, 1.2.0.479, 1.2.0.480, 1.2.0.481, 1.2.0.482, 1.2.0.483, 1.2.0.484, 1.2.0.485, 1.2.0.486, 1.2.0.487, 1.2.0.488, 1.2.0.489, 1.2.0.490, 1.2.0.491, 1.2.0.492, 1.2.0.493, 1.2.0.494, 1.2.0.495, 1.2.0.496, 1.2.0.497, 1.2.0.498, 1.2.0.499, 1.2.0.500, 1.2.0.501, 1.2.0.502, 1.2.0.503, 1.2.0.504, 1.2.0.505, 1.2.0.506, 1.2.0.507, 1.2.0.508, 1.2.0.509, 1.2.0.510, 1.2.0.511, 1.2.0.512, 1.2.0.513, 1.2.0.514, 1.2.0.515, 1.2.0.516, 1.2.0.517, 1.2.0.518, 1.2.0.519, 1.2.0.520, 1.2.0.521, 1.2.0.522, 1.2.0.523, 1.2.0.524, 1.2.0.525, 1.2.0.526, 1.2.0.527, 1.2.0.528, 1.2.0.529, 1.2.0.530, 1.2.0.531, 1.2.0.532, 1.2.0.533, 1.2.0.534, 1.2.0.535, 1.2.0.536, 1.2.0.537, 1.2.0.538, 1.2.0.539, 1.2.0.540, 1.2.0.541, 1.2.0.542, 1.2.0.543, 1.2.0.544, 1.2.0.545, 1.2.0.546, 1.2.0.547, 1.2.0.548, 1.2.0.549, 1.2.0.550, 1.2.0.551, 1.2.0.552, 1.2.0.553, 1.2.0.554, 1.2.0.555, 1.2.0.556, 1.2.0.557, 1.2.0.558, 1.2.0.559, 1.2.0.560, 1.2.0.561, 1.2.0.562, 1.2.0.563, 1.2.0.564, 1.2.0.565, 1.2.0.566, 1.2.0.567, 1.2.0.568, 1.2.0.569, 1.2.0.570, 1.2.0.571, 1.2.0.572, 1.2.0.573, 1.2.0.574, 1.2.0.575, 1.2.0.576, 1.2.0.577, 1.2.0.578, 1.2.0.579, 1.2.0.580, 1.2.0.581, 1.2.0.582, 1.2.0.583, 1.2.0.584, 1.2.0.585, 1.2.0.586, 1.2.0.587, 1.2.0.588, 1.2.0.589, 1.2.0.590, 1.2.0.591, 1.2.0.592, 1.2.0.593, 1.2.0.594, 1.2.0.595, 1.2.0.596, 1.2.0.597, 1.2.0.598, 1.2.0.599, 1.2.0.600, 1.2.0.601, 1.2.0.602, 1.2.0.603, 1.2.0.604, 1.2.0.605, 1.2.0.606, 1.2.0.607, 1.2.0.608, 1.2.0.609, 1.2.0.610, 1.2.0.611, 1.2.0.612, 1.2.0.613, 1.2.0.614, 1.2.0.615, 1.2.0.616, 1.2.0.617, 1.2.0.618, 1.2.0.619, 1.2.0.620, 1.2.0.621, 1.2.0.622, 1.2.0.623, 1.2.0.624, 1.2.0.625, 1.2.0.626, 1.2.0.627, 1.2.0.628, 1.2.0.629, 1.2.0.630, 1.2.0.631, 1.2.0.632, 1.2.0.633, 1.2.0.634, 1.2.0.635, 1.2.0.636, 1.2.0.637, 1.2.0.638, 1.2.0.639, 1.2.0.640, 1.2.0.641, 1.2.0.642, 1.2.0.643, 1.2.0.644, 1.2.0.645, 1.2.0.646, 1.2.0.647, 1.2.0.648, 1.2.0.649, 1.2.0.650, 1.2.0.651, 1.2.0.652, 1.2.0.653, 1.2.0.654, 1.2.0.655, 1.2.0.656, 1.2.0.657, 1.2.0.658, 1.2.0.659, 1.2.0.660, 1.2.0.661, 1.2.0.662, 1.2.0.663, 1.2.0.664, 1.2.0.665, 1.2.0.666, 1.2.0.667, 1.2.0.668, 1.2.0.669, 1.2.0.670, 1.2.0.671, 1.2.0.672, 1.2.0.673, 1.2.0.674, 1.2.0.675, 1.2.0.676, 1.2.0.677, 1.2.0.678, 1.2.0.679, 1.2.0.680, 1.2.0.681, 1.2.0.682, 1.2.0.683, 1.2.0.684, 1.2.0.685, 1.2.0.686, 1.2.0.687, 1.2.0.688, 1.2.0.689, 1.2.0.690, 1.2.0.691, 1.2.0.692, 1.2.0.693, 1.2.0.694, 1.2.0.695, 1.2.0.696, 1.2.0.697, 1.2.0.698, 1.2.0.699, 1.2.0.700, 1.2.0.701, 1.2.0.702, 1.2.0.703, 1.2.0.704, 1.2.0.705, 1.2.0.706, 1.2.0.707, 1.2.0.708, 1.2.0.709, 1.2.0.710, 1.2.0.711, 1.2.0.712, 1.2.0.713, 1.2.0.714, 1.2.0.715, 1.2.0.716, 1.2.0.717, 1.2.0.718, 1.2.0.719, 1.2.0.720, 1.2.0.721, 1.2.0.722, 1.2.0.723, 1.2.0.724, 1.2.0.725, 1.2.0.726, 1.2.0.727, 1.2.0.728, 1.2.0.729, 1.2.0.730, 1.2.0.731, 1.2.0.732, 1.2.0.733, 1.2.0.734, 1.2.0.735, 1.2.0.736, 1.2.0.737, 1.2.0.738, 1.2.0.739, 1.2.0.740, 1.2.0.741, 1.2.0.742, 1.2.0.743, 1.2.0.744, 1.2.0.745, 1.2.0.746, 1.2.0.747, 1.2.0.748, 1.2.0.749, 1.2.0.750, 1.2.0.751, 1.2.0.752, 1.2.0.753, 1.2.0.754, 1.2.0.755, 1.2.0.756, 1.2.0.757, 1.2.0.758, 1.2.0.759, 1.2.0.760, 1.2.0.761, 1.2.0.762, 1.2.0.763, 1.2.0.764, 1.2.0.765, 1.2.0.766, 1.2.0.767, 1.2.0.768, 1.2.0.769, 1.2.0.770, 1.2.0.771, 1.2.0.772, 1.2.0.773, 1.2.0.774, 1.2.0.775, 1.2.0.776, 1.2.0.777, 1.2.0.778, 1.2.0.779, 1.2.0.780, 1.2.0.781, 1.2.0.782, 1.2.0.783, 1.2.0.784, 1.2.0.785, 1.2.0.786, 1.2.0.787, 1.2.0.788, 1.2.0.789, 1.2.0.790, 1.2.0.791, 1.2.0.792, 1.2.0.793, 1.2.0.794, 1.2.0.795, 1.2.0.796, 1.2.0.797, 1.2.0.798, 1.2.0.799, 1.2.0.800, 1.2.0.801, 1.2.0.802, 1.2.0.803, 1.2.0.804, 1.2.0.805, 1.2.0.806, 1.2.0.807, 1.2.0.808, 1.2.0.809, 1.2.0.810, 1.2.0.811, 1.2.0.812, 1.2.0.813, 1.2.0.814, 1.2.0.815, 1.2.0.816, 1.2.0.817, 1.2.0.818, 1.2.0.819, 1.2.0.820, 1.2.0.821, 1.2.0.822, 1.2.0.823, 1.2.0.824, 1.2.0.825, 1.2.0.826, 1.2.0.827, 1.2.0.828, 1.2.0.829, 1.2.0.830, 1.2.0.831, 1.2.0.832, 1.2.0.833, 1.2.0.834, 1.2.0.835, 1.2.0.836, 1.2.0.837, 1.2.0.838, 1.2.0.839, 1.2.0.840, 1.2.0.841, 1.2.0.842, 1.2.0.843, 1.2.0.844, 1.2.0.845, 1.2.0.846, 1.2.0.847, 1.2.0.848, 1.2.0.849, 1.2.0.850, 1.2.0.851, 1.2.0.852, 1.2.0.853, 1.2.0.854, 1.2.0.855, 1.2.0.856, 1.2.0.857, 1.2.0.858, 1.2.0.859, 1.2.0.860, 1.2.0.861, 1.2.0.862, 1.2.0.863, 1.2.0.864, 1.2.0.865, 1.2.0.866, 1.2.0.867, 1.2.0.868, 1.2.0.869, 1.2.0.870, 1.2.0.871, 1.2.0.872, 1.2.0.873, 1.2.0.874, 1.2.0.875, 1.2.0.876, 1.2.0.877, 1.2.0.878, 1.2.0.879, 1.2.0.880, 1.2.0.881, 1.2.0.882, 1.2.0.883, 1.2.0.884, 1.2.0.885, 1.2.0.886, 1.2.0.887, 1.2.0.888, 1.2.0.889, 1.2.0.890, 1.2.0.891, 1.2.0.892, 1.2.0.893, 1.2.0.894, 1.2.0.895, 1.2.0.896, 1.2.0.897, 1.2.0.898, 1.2.0.899, 1.2.0.900, 1.2.0.901, 1.2.0.902, 1.2.0.903, 1.2.0.904, 1.2.0.905, 1.2.0.906, 1.2.0.907, 1.2.0.908, 1.2.0.909, 1.2.0.910, 1.2.0.911, 1.2.0.912, 1.2.0.913, 1.2.0.914, 1.2.0.915, 1.2.0.916, 1.2.0.917, 1.2.0.918, 1.2.0.919, 1.2.0.920, 1.2.0.921, 1.2.0.922, 1.2.0.923, 1.2.0.924, 1.2.0.925, 1.2.0.926, 1.2.0.927, 1.2.0.928, 1.2.0.929, 1.2.0.930, 1.2.0.931, 1.2.0.932, 1.2.0.933, 1.2.0.934, 1.2.0.935, 1.2.0.936, 1.2.0.937, 1.2.0.938, 1.2.0.939, 1.2.0.940, 1.2.0.941, 1.2.0.942, 1.2.0.943, 1.2.0.944, 1.2.0.945, 1.2.0.946, 1.2.0.947, 1.2.0.948, 1.2.0.949, 1.2.0.950, 1.2.0.951, 1.2.0.952, 1.2.0.953, 1.2.0.954, 1.2.0.955, 1.2.0.956, 1.2.0.957, 1.2.0.958, 1.2.0.959, 1.2.0.960, 1.2.0.961, 1.2.0.962, 1.2.0.963, 1.2.0.964, 1.2.0.965, 1.2.0.966, 1.2.0.967, 1.2.0.968, 1.2.0.969, 1.2.0.970, 1.2.0.971, 1.2.0.972, 1.2.0.973, 1.2.0.974, 1.2.0.975, 1.2.0.976, 1.2.0.977, 1.2.0.978, 1.2.0.979, 1.2.0.980, 1.2.0.981, 1.2.0.982, 1.2.0.983, 1.2.0.984, 1.2.0.985, 1.2.0.986, 1.2.0.987, 1.2.0.988, 1.2.0.989, 1.2.0.990, 1.2.0.991, 1.2.0.992, 1.2.0.993, 1.2.0.994, 1.2.0.995, 1.2.0.996, 1.2.0.997, 1.2.0.998, 1.2.0.999, 1.2.0.1000, 1.2.0.1001, 1.2.0.1002, 1.2.0.1003, 1.2.0.1004, 1.2.0.1005, 1.2.0.1006, 1.2.0.1007, 1.2.0.1008, 1.2.0.1009, 1.2.0.1010, 1.2.0.1011, 1.2.0.1012, 1.2.0.1013, 1.2.0.1014, 1.2.0.1015, 1.2.0.1016, 1.2.0.1017, 1.2.0.1018, 1.2.0.1019, 1.2.0.1020, 1.2.0.1021, 1.2.0.1022, 1.2.0.1023, 1.2.0.1024, 1.2.0.1025, 1.2.0.1026, 1.2.0.1027, 1.2.0.1028, 1.2.0.1029, 1.2.0.1030, 1.2.0.1031, 1.2.0.1032, 1.2.0.1033, 1.2.0.1034, 1.2.0

Start
+



This paper's route



Finish



Reverse
mode
AD



Start
+



This paper's route



Finish

Reverse
mode
AD

Edward Kmett route



Conclusions

Take away ideas

- A series of simple steps got us
 - From a simple, “obviously correct” forward mode solution
 - To a subtle, but still correct, reverse mode solution
 - Using the dual-number idea all the way
- Simple enough to have a correctness proof

Plenty to do

- More general argument and result types

- e.g $\left(\text{Array of } \mathbb{R}^3, \mathbb{R}^{3 \times 4}\right) \rightarrow \mathbb{R}^3$

- Nested arrays

- Recursion and recursive types

- Higher order derivatives

- Fusion and checkpointing

- ~~A decent implementation~~