Mixed-Mode Automatic Differentiation in julia

Jarrett Revels, Miles Lubin & Juan Pablo Vielma (MIT)

Introduction

Hi, I’m Jarrett

- Started writing Julia code in 2013, working on AD since 2015
- Previously worked in the Julia Group @ CSAIL under Alan Edelman
- Authored Julia's performance regressions testing facilities (*BenchmarkTools*)
- Recently started an engineering position under Juan Pablo Vielma
- Continuing to work on AD, transitioning to direct JuMP development in the fall

My Users Are Smarter Than Me

- No formal optimization background (B.S. in Physics, 2014)
- A lot of my work targets other Julia developers rather than end-users
- Downstream packages: *JuMP*, *Celeste*, *Optim*, *DifferentialEquations*, *RigidBodyDynamics*, *ValidatedNumerics*, etc...

Forward-Mode AD

Multidimensional Dual Numbers

$$f(x + y\epsilon) = f(x) + f'(x)y\epsilon \text{ where } \epsilon \neq 0, \epsilon^2 = 0$$

Multidimensional Dual Numbers

$$f(x + y\epsilon) = f(x) + f'(x)y\epsilon \text{ where } \epsilon \neq 0, \epsilon^2 = 0$$

$$f(x+\sum_{i=1}^{n} y_i\epsilon_i) = f(x)+f'(x)\sum_{i=1}^{n} y_i\epsilon_i \text{ where } \epsilon_i \neq 0, \epsilon_i\epsilon_j = 0$$

Multidimensional Dual Numbers

$$f(x + y\epsilon) = f(x) + f'(x)y\epsilon \text{ where } \epsilon \neq 0, \epsilon^2 = 0$$

$$f(x + \sum_{i=1}^{n} y_i\epsilon_i) = f(x) + f'(x)\sum_{i=1}^{n} y_i\epsilon_i \text{ where } \epsilon_i \neq 0, \epsilon_i\epsilon_j = 0$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} \rightarrow \mathbf{x}_\epsilon = \begin{bmatrix} x_1 + \epsilon_1 \\ \vdots \\ x_i + \epsilon_i \\ \vdots \\ x_n + \epsilon_n \end{bmatrix}$$

Multidimensional Dual Numbers

$$f(x + y\epsilon) = f(x) + f'(x)y\epsilon \text{ where } \epsilon \neq 0, \epsilon^2 = 0$$

$$f(x+\sum_{i=1}^{n} y_i\epsilon_i) = f(x)+f'(x)\sum_{i=1}^{n} y_i\epsilon_i \text{ where } \epsilon_i \neq 0, \epsilon_i\epsilon_j = 0$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix} \rightarrow \mathbf{x}_\epsilon = \begin{bmatrix} x_1 + \epsilon_1 \\ \vdots \\ x_i + \epsilon_i \\ \vdots \\ x_n + \epsilon_n \end{bmatrix}$$

$$g(\mathbf{x}_\epsilon) = g(\mathbf{x}) + \sum_{i=1}^{n} \frac{\partial g(\mathbf{x})}{\partial x_i}\epsilon_i$$

The `Dual` Type

```julia
# stack-allocated vector of partial derivatives
using StaticArrays.SVector


# N-dimensional dual number type
struct Dual{N,T<:Real} <: Real
    value::T
    partials::SVector{N,T}
end
```

The `Dual` Type

```julia
# stack-allocated vector of partial derivatives
using StaticArrays.SVector


# N-dimensional dual number type
struct Dual{N,T<:Real} <: Real
    value::T
    partials::SVector{N,T}
end


# overload various math operations
import Base: sin, cos, -, +, *

sin(d::Dual) = Dual(sin(d.value), cos(d.value) * d.partials)
cos(d::Dual) = Dual(cos(d.value), -(sin(d.value)) * d.partials)
(-)(d::Dual) = Dual(-(d.value), -(d.partials))
(+)(a::Dual, b::Dual) = Dual(a.value + b.value, a.partials + b.partials)
(*)(a::Dual, b::Dual) = Dual(a.value * b.value,
                             b.value * a.partials + a.value * b.partials)
```

The `Dual` Type

```
# stack-allocated vector of partial derivatives
using StaticArrays.SVector


# N-dimensional dual number type
struct Dual{N,T<:Real} <: Real
    value::T
    partials::SVector{N,T}
end


# overload various math operations
import Base: sin, cos, -, +, *

sin(d::Dual) = Dual(sin(d.value), cos(d.value) * d.partials)
cos(d::Dual) = Dual(cos(d.value), -(sin(d.value)) * d.partials)
(-)(d::Dual) = Dual(-(d.value), -(d.partials))
(+)(a::Dual, b::Dual) = Dual(a.value + b.value, a.partials + b.partials)
(*)(a::Dual, b::Dual) = Dual(a.value * b.value,
                             b.value * a.partials + a.value * b.partials)
```

This code enables:

- `sin` and `cos` derivatives to **arbitrary order** (e.g. `Dual{M,Dual{N,T}}`)
- `sin` and `cos` derivatives over **complex number types** (e.g. `Complex{Dual{N,T}}`)
- `sin` and `cos` derivatives over **custom number types** (e.g. `Custom{Dual{N,T}}`)

Taking Jacobians using `Duals`

$$\text{cumprod}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}\right) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \dots x_1 \end{bmatrix}$$

```
function cumprod(x)
    y = similar(x)
    if length(x) < 1
        return y
    end
    y[1] = x[1]
    for i in 2:length(y)
        y[i] = y[i-1]*x[i]
    end
    return y
end
```

Taking Jacobians using `Duals`

$$\mathbf{J}(\mathbf{g})(\mathbf{x}) = \begin{bmatrix} \dfrac{\partial g_1(\mathbf{x})}{\partial x_1} & \cdots & \dfrac{\partial g_1(\mathbf{x})}{\partial x_i} & \cdots & \dfrac{\partial g_1(\mathbf{x})}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \\ \dfrac{\partial g_j(\mathbf{x})}{\partial x_1} & \cdots & \dfrac{\partial g_j(\mathbf{x})}{\partial x_i} & \cdots & \dfrac{\partial g_j(\mathbf{x})}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \\ \dfrac{\partial g_m(\mathbf{x})}{\partial x_1} & \cdots & \dfrac{\partial g_m(\mathbf{x})}{\partial x_i} & \cdots & \dfrac{\partial g_m(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

Taking Jacobians using `Duals`

$$\mathbf{J}(\mathbf{g})(\mathbf{x}) = \begin{bmatrix} \frac{\partial g_1(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial g_1(\mathbf{x})}{\partial x_i} & \cdots & \frac{\partial g_1(\mathbf{x})}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \\ \frac{\partial g_j(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial g_j(\mathbf{x})}{\partial x_i} & \cdots & \frac{\partial g_j(\mathbf{x})}{\partial x_n} \\ \vdots & \ddots & \vdots & \ddots & \\ \frac{\partial g_m(\mathbf{x})}{\partial x_1} & \cdots & \frac{\partial g_m(\mathbf{x})}{\partial x_i} & \cdots & \frac{\partial g_m(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

$$\mathbf{g}(\mathbf{x}_\epsilon) = \begin{bmatrix} g_1(\mathbf{x}_\epsilon) \\ \vdots \\ g_j(\mathbf{x}_\epsilon) \\ \vdots \\ g_m(\mathbf{x}_\epsilon) \end{bmatrix} = \begin{bmatrix} g_1(\mathbf{x}) + \sum_{i=1}^n \frac{\partial g_1(\mathbf{x})}{\partial x_i}\epsilon_i \\ \vdots \\ g_j(\mathbf{x}) + \sum_{i=1}^n \frac{\partial g_j(\mathbf{x})}{\partial x_i}\epsilon_i \\ \vdots \\ g_m(\mathbf{x}) + \sum_{i=1}^n \frac{\partial g_m(\mathbf{x})}{\partial x_i}\epsilon_i \end{bmatrix}$$

Taking Jacobians using `Duals`

$$\mathbf{J}(\mathbf{g})(\mathbf{x}) = \begin{bmatrix} \boxed{\frac{\partial g_1(\mathbf{x})}{\partial x_1} \quad \cdots \quad \frac{\partial g_1(\mathbf{x})}{\partial x_i} \quad \cdots \quad \frac{\partial g_1(\mathbf{x})}{\partial x_n}} \\ \vdots \quad \ddots \quad \vdots \quad \ddots \\ \boxed{\frac{\partial g_j(\mathbf{x})}{\partial x_1} \quad \cdots \quad \frac{\partial g_j(\mathbf{x})}{\partial x_i} \quad \cdots \quad \frac{\partial g_j(\mathbf{x})}{\partial x_n}} \\ \vdots \quad \ddots \quad \vdots \quad \ddots \\ \boxed{\frac{\partial g_m(\mathbf{x})}{\partial x_1} \quad \cdots \quad \frac{\partial g_m(\mathbf{x})}{\partial x_i} \quad \cdots \quad \frac{\partial g_m(\mathbf{x})}{\partial x_n}} \end{bmatrix}$$

$$\mathbf{g}(\mathbf{x}_\epsilon) = \begin{bmatrix} g_1(\mathbf{x}_\epsilon) \\ \vdots \\ g_j(\mathbf{x}_\epsilon) \\ \vdots \\ g_m(\mathbf{x}_\epsilon) \end{bmatrix} = \begin{bmatrix} g_1(\mathbf{x}) + \sum_{i=1}^n \boxed{\frac{\partial g_1(\mathbf{x})}{\partial x_i}} \epsilon_i \\ \vdots \\ g_j(\mathbf{x}) + \sum_{i=1}^n \boxed{\frac{\partial g_j(\mathbf{x})}{\partial x_i}} \epsilon_i \\ \vdots \\ g_m(\mathbf{x}) + \sum_{i=1}^n \boxed{\frac{\partial g_m(\mathbf{x})}{\partial x_i}} \epsilon_i \end{bmatrix}$$

Taking Jacobians using `Duals`

$$\text{cumprod}(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \dots x_1 \end{bmatrix}$$

Taking Jacobians using `Duals`

$$\text{cumprod}(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \dots x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}) = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 x_1 & x_1 & 0 \\ x_3 x_2 & x_3 x_1 & x_2 x_1 \end{bmatrix}$$

Taking Jacobians using `Duals`

$$\text{cumprod}(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \ldots x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}) = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 x_1 & x_1 & 0 \\ x_3 x_2 & x_3 x_1 & x_2 x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})(\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}) = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 6 & 3 & 2 \end{bmatrix}$$

Taking Jacobians using `Duals`

$$\text{cumprod}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}\right) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \dots x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 x_1 & x_1 & 0 \\ x_3 x_2 & x_3 x_1 & x_2 x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})\left(\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}\right) = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 6 & 3 & 2 \end{bmatrix}$$

```julia
julia> x = [Dual(1, 1, 0, 0),  # 1 + ϵ₁
            Dual(2, 0, 1, 0),  # 2 + ϵ₂
            Dual(3, 0, 0, 1)]; # 3 + ϵ₃
```

Taking Jacobians using `Duals`

$$\mathrm{cumprod}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}\right) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \ldots x_1 \end{bmatrix}$$

$$\downarrow$$

$$\mathbf{J}(\mathrm{cumprod})\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 x_1 & x_1 & 0 \\ x_3 x_2 & x_3 x_1 & x_2 x_1 \end{bmatrix}$$

$$\downarrow$$

$$\mathbf{J}(\mathrm{cumprod})\left(\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}\right) = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 6 & 3 & 2 \end{bmatrix}$$

```
julia> x = [Dual(1, 1, 0, 0),  # 1 + ϵ₁
            Dual(2, 0, 1, 0),  # 2 + ϵ₂
            Dual(3, 0, 0, 1)]; # 3 + ϵ₃
```

```
julia> cumprod(x)
3-element Array{Dual{3,Int64},1}:
 Dual(1,1,0,0)
 Dual(2,2,1,0)
 Dual(6,6,3,2)
```

Taking Jacobians using `Duals`

$$\text{cumprod}\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}\right) = \begin{bmatrix} x_1 \\ x_2 x_1 \\ x_3 x_2 x_1 \\ \vdots \\ x_n x_{n-1} x_{n-2} \ldots x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})\left(\begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}\right) = \begin{bmatrix} x_1 & 0 & 0 \\ x_2 x_1 & x_1 & 0 \\ x_3 x_2 & x_3 x_1 & x_2 x_1 \end{bmatrix}$$

$$\mathbf{J}(\text{cumprod})\left(\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}\right) = \boxed{\begin{bmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ 6 & 3 & 2 \end{bmatrix}}$$

```
julia> x = [Dual(1, 1, 0, 0),   # 1 + ϵ₁
            Dual(2, 0, 1, 0),   # 2 + ϵ₂
            Dual(3, 0, 0, 1)];  # 3 + ϵ₃
```

```
julia> cumprod(x)
3-element Array{Dual{3,Int64},1}:
 Dual(1,1,0,0)
 Dual(2,2,1,0)
 Dual(6,6,3,2)
```

Perturbation Confusion

```
D = (f, x_0) -> df/dx evaluated at x_0

# nested, closed over differentiation
D(x -> x * D(y -> x + y, 1), 1)

# correct answer
d1 = D(x -> x * D(y -> x + y, 1), 1)
d1 = D(x -> x * (y -> 1)(1), 1)
d1 = D(x -> x, 1)
d1 = (x -> 1)(1)
d1 = 1
```

Perturbation Confusion

```
D = (f, x_0) -> df/dx evaluated at x_0

# nested, closed over differentiation
D(x -> x * D(y -> x + y, 1), 1)

# correct answer
d1 = D(x -> x * D(y -> x + y, 1), 1)
d1 = D(x -> x * (y -> 1)(1), 1)
d1 = D(x -> x, 1)
d1 = (x -> 1)(1)
d1 = 1
```

```
epsilon(d::Dual) = d.partials[1]

# what AD using naive dual numbers will compute
d2 = D(x -> x * D(y -> x + y, 1), 1)
d2 = D(x -> x * epsilon(x + Dual(1,1)), 1)
d2 = epsilon(Dual(1,1) *
     epsilon(Dual(1,1) + Dual(1,1)))
d2 = epsilon(Dual(1,1) * epsilon(Dual(2,2)))
d2 = epsilon(Dual(1,1) * 2)
d2 = epsilon(Dual(2,2))
d2 = 2 # != d1
```

ForwardDiff.jl

- Implements multidimensional dual numbers
- Fully stack-allocated and aggressively inlined, plays well with SIMD
- Tagging system prevents perturbation confusion and resolves nested differentiation ambiguities
- Provides an API instead of exposing dual numbers directly (e.g. `ForwardDiff.jacobian(f,x)`)
- Used by JuMP for calculating Hessian-vector products

Reverse-Mode AD

Compared to Forward-Mode AD

- Instead of propagating an input perturbation forward, we propagate an output sensitivity backwards
  - Forward-mode AD evaluates chain rule from right (inner function) to left (outer function)
  - Reverse-mode AD evaluates chain rule from left (outer function) to right (inner function)

- Main hurdle: requires a reverse-traversable computation graph
  - Graph can be defined declaratively via special objects/syntax (JuMP, TensorFlow)
  - ...or dynamically by tracking arguments/intercepting function calls (ReverseDiff, Autograd, PyTorch)

- Which mode should I use?
  - Output dimension < input dimension && input dimension >> code size? Use reverse mode
  - Output dimension > input dimension && input dimension << code size? Use forward mode
  - Output dimension = input dimension? That’s tough

Code Representation

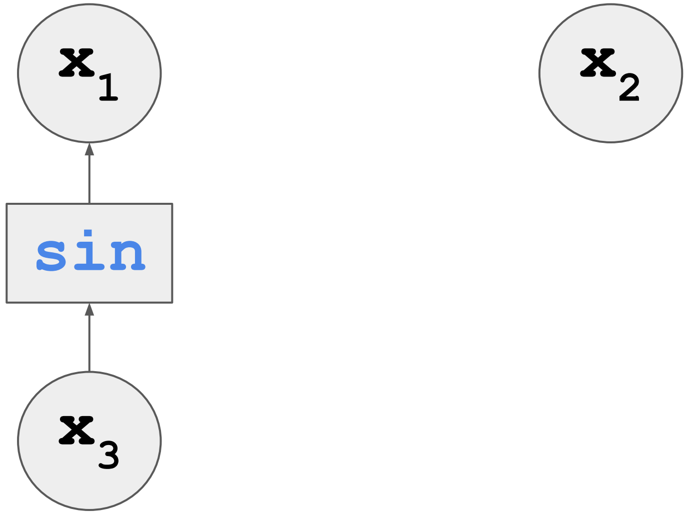
```
function f(x1, x2)
    # ?
end
```

Graph Representation

Code Representation

```
function f(x₁, x₂)
    x₃ = sin(x₁)
    # ?
end
```

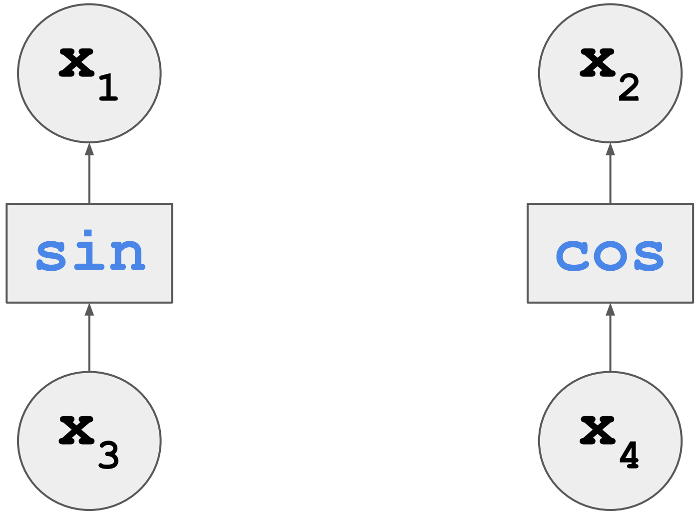
Graph Representation



Code Representation

```
function f(x₁, x₂)
    x₃ = sin(x₁)
    x₄ = cos(x₂)
    # ?
end
```

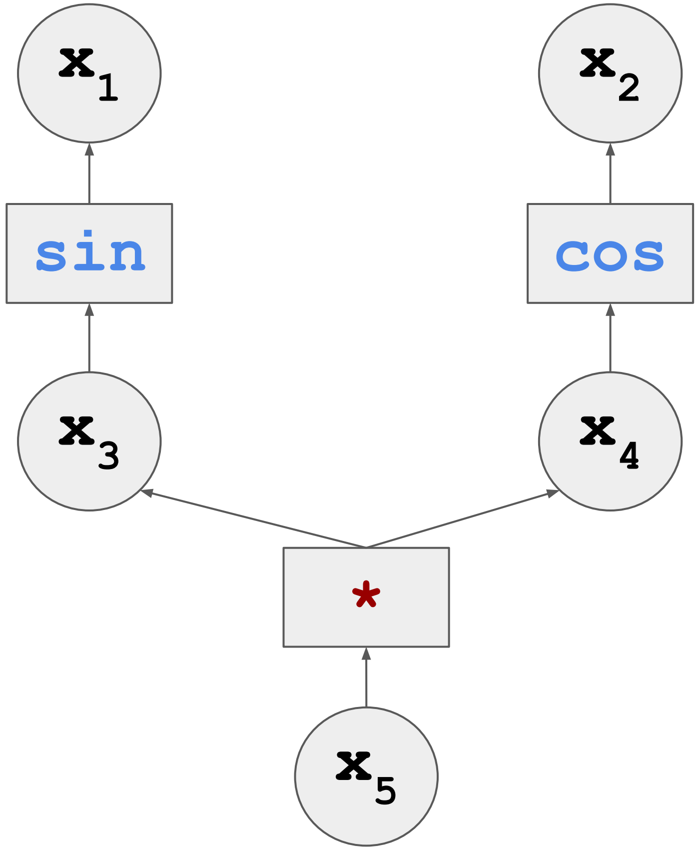
Graph Representation



Code Representation

```
function f(x1, x2)
    x3 = sin(x1)
    x4 = cos(x2)
    x5 = x3 * x4
    # ?
end
```

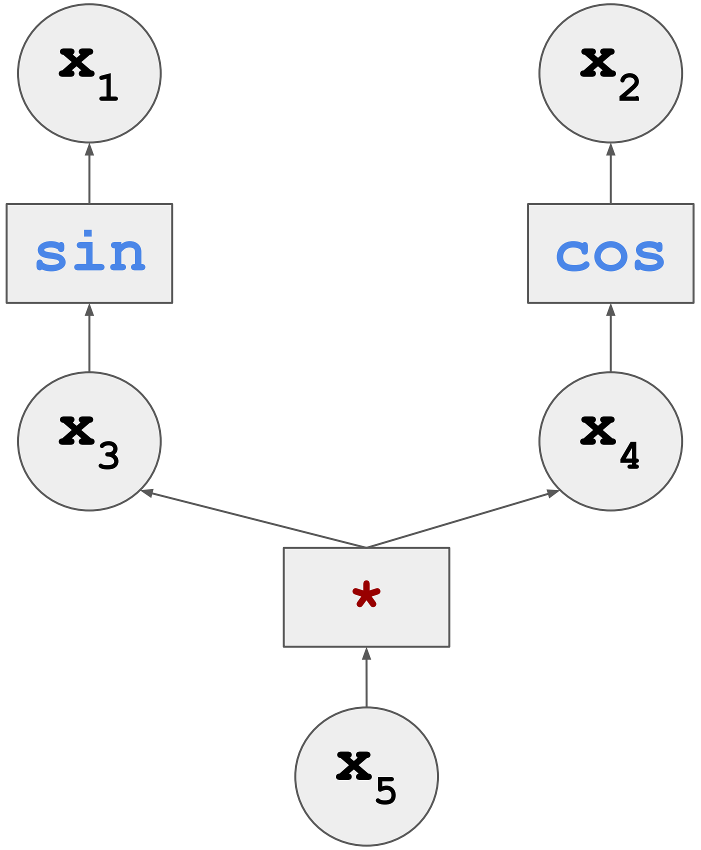
Graph Representation

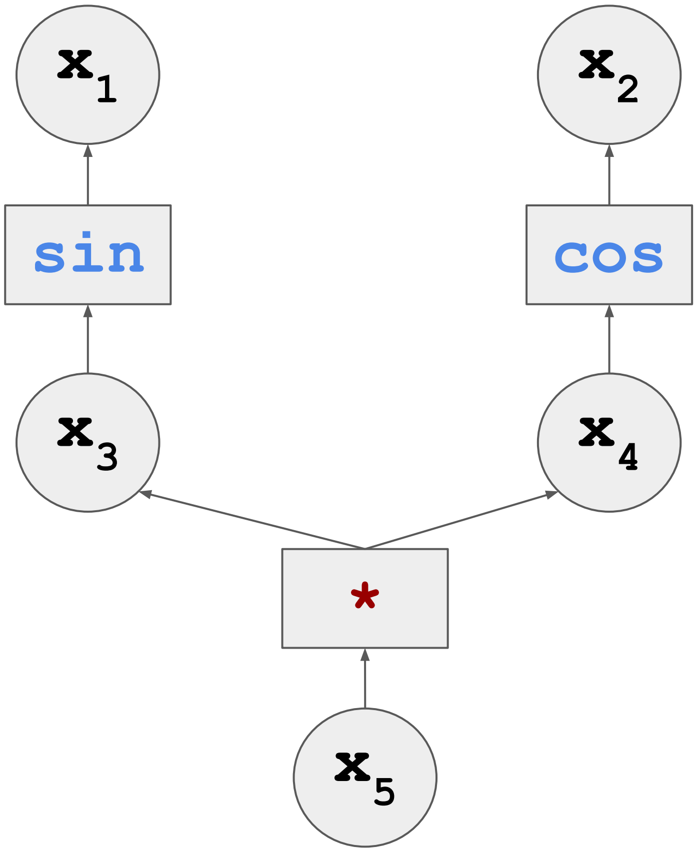


Code Representation

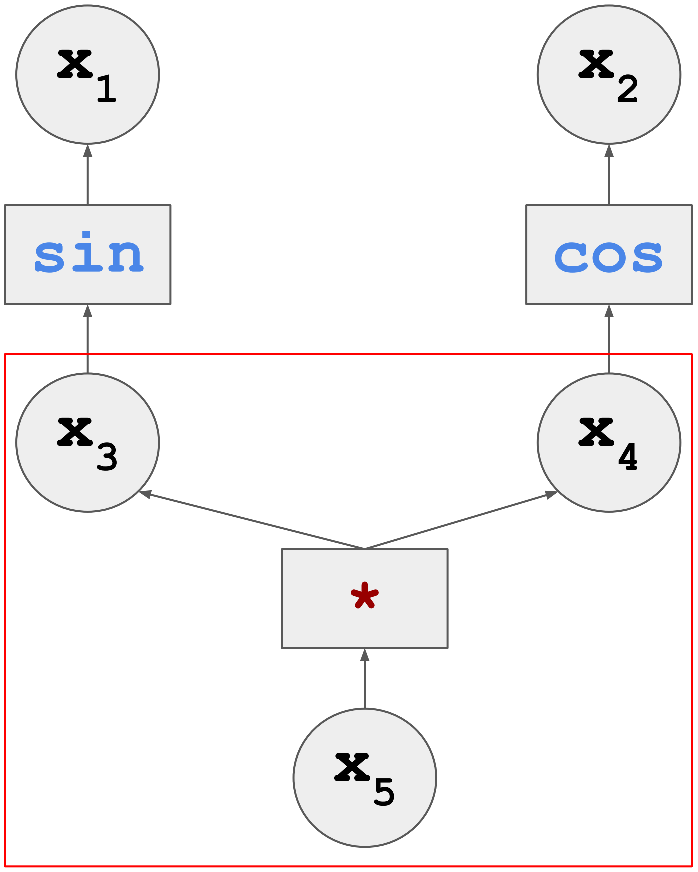
```
function f(x_1, x_2)
    x_3 = sin(x_1)
    x_4 = cos(x_2)
    x_5 = x_3 * x_4
    return x_5
end
```

Graph Representation

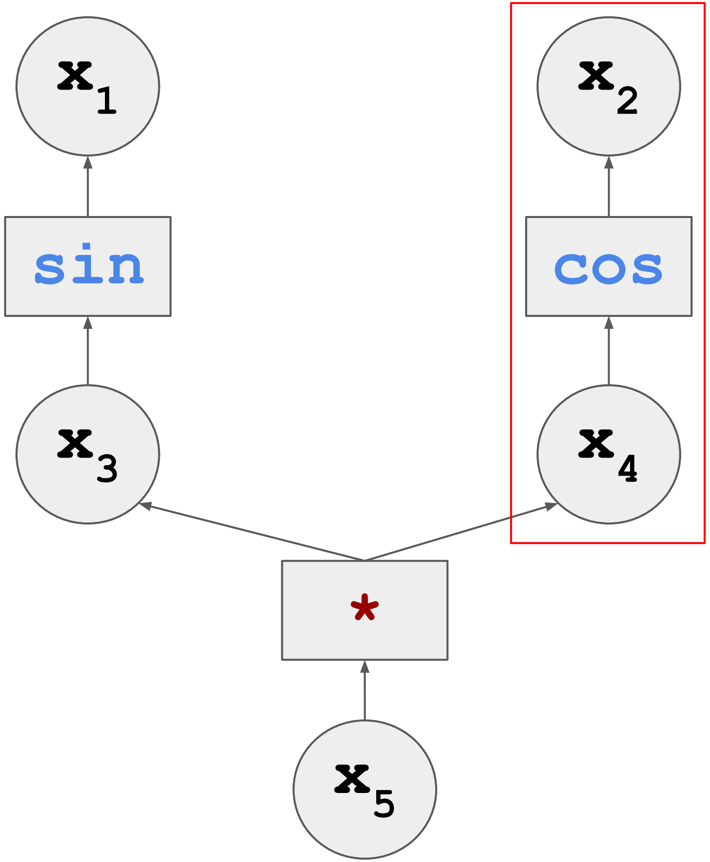




x1
sin
x2
cos
x3
x4
*
x5

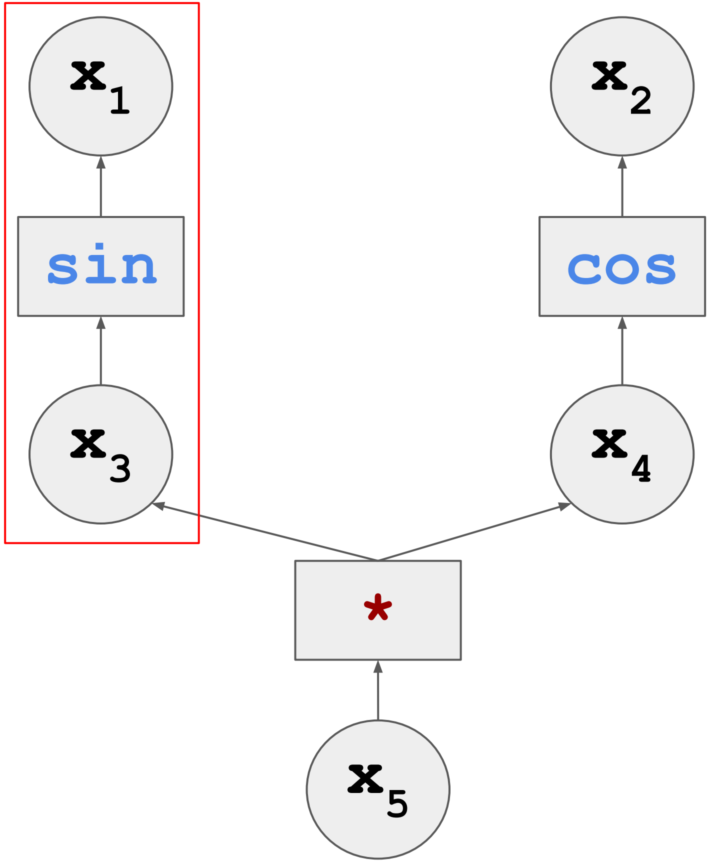


$$x_5 = x_3 * x_4$$



$$x_4 = \cos(x_2)$$

$$x_5 = x_3 * x_4$$



$x_3 = \sin(x_1)$

$x_4 = \cos(x_2)$

$x_5 = x_3 * x_4$

Multivariable Chain Rule

$y_i = \partial x_5/\partial x_i$

$= \text{sum}(y_j * \partial x_j/\partial x_i \text{ for } j \text{ in parents}(i))$

Derivative Outputs

$\partial f(x_1,x_2)/\partial x_1 = \partial x_5/\partial x_1$
$= y_1$

$\partial f(x_1,x_2)/\partial x_2 = \partial x_5/\partial x_2$
$= y_2$

Numerical Results

$x_1 = 1.0$	$y_1 = 0.0$
$x_2 = 1.0$	$y_2 = 0.0$
$x_3 = 0.0$	$y_3 = 0.0$
$x_4 = 0.0$	$y_4 = 0.0$
$x_5 = 0.0$	$y_5 = 0.0$

$x_3 = \sin(x_1)$

$x_4 = \cos(x_2)$

$x_5 = x_3 * x_4$

Multivariable Chain Rule

$y_i = \partial x_5 / \partial x_i$

$= \text{sum}(y_j * \partial x_j / \partial x_i \text{ for } j \text{ in } \text{parents}(i))$

Derivative Outputs

$\partial f(x_1, x_2)/\partial x_1 = \partial x_5/\partial x_1$
$= y_1$

$\partial f(x_1, x_2)/\partial x_2 = \partial x_5/\partial x_2$
$= y_2$

Numerical Results

$x_1 = 1.0$	$y_1 = 0.0$
$x_2 = 1.0$	$y_2 = 0.0$
$x_3 = 0.8$	$y_3 = 0.0$
$x_4 = 0.0$	$y_4 = 0.0$
$x_5 = 0.0$	$y_5 = 0.0$

$x_3 = \sin(x_1)$

$x_4 = \cos(x_2)$

$x_5 = x_3 * x_4$

Multivariable Chain Rule

$y_i = \partial x_5 / \partial x_i$

$= \text{sum}(y_j * \partial x_j / \partial x_i \text{ for } j \text{ in parents}(i))$

Derivative Outputs

$\partial f(x_1, x_2)/\partial x_1 = \partial x_5/\partial x_1$
$= y_1$

$\partial f(x_1, x_2)/\partial x_2 = \partial x_5/\partial x_2$
$= y_2$

Numerical Results

$x_1 = 1.0$	$y_1 = 0.0$
$x_2 = 1.0$	$y_2 = 0.0$
$x_3 = 0.8$	$y_3 = 0.0$
$x_4 = 0.5$	$y_4 = 0.0$
$x_5 = 0.0$	$y_5 = 0.0$

$x_3 = \sin(x_1)$

$x_4 = \cos(x_2)$

$x_5 = x_3 * x_4$

Multivariable Chain Rule

$y_i = \partial x_5/\partial x_i$

$= \text{sum}(y_j * \partial x_j/\partial x_i \text{ for } j \text{ in parents}(i))$

Derivative Outputs

$\partial f(x_1,x_2)/\partial x_1 = \partial x_5/\partial x_1$
$= y_1$

$\partial f(x_1,x_2)/\partial x_2 = \partial x_5/\partial x_2$
$= y_2$

Numerical Results

$x_1 = 1.0$	$y_1 = 0.0$
$x_2 = 1.0$	$y_2 = 0.0$
$x_3 = 0.8$	$y_3 = 0.0$
$x_4 = 0.5$	$y_4 = 0.0$
$x_5 = 0.4$	$y_5 = 0.0$

$x_3 = \sin(x_1)$

$x_4 = \cos(x_2)$

$x_5 = x_3 * x_4$

Multivariable Chain Rule

$y_i = \partial x_5/\partial x_i$

$\quad = \mathrm{sum}(y_j * \partial x_j/\partial x_i \text{ for } j \text{ in } \mathrm{parents}(i))$

Derivative Outputs

$\partial f(x_1,x_2)/\partial x_1 = \partial x_5/\partial x_1$
$\quad = y_1$

$\partial f(x_1,x_2)/\partial x_2 = \partial x_5/\partial x_2$
$\quad = y_2$

Numerical Results

$x_1 = 1.0$	$y_1 = 0.0$
$x_2 = 1.0$	$y_2 = 0.0$
$x_3 = 0.8$	$y_3 = 0.0$
$x_4 = 0.5$	$y_4 = 0.0$
$x_5 = 0.4$	→ $y_5 = 1.0$

$x_3 = \sin(x_1)$

$x_4 = \cos(x_2)$

$x_5 = x_3 * x_4$

Multivariable Chain Rule

$y_i = \partial x_5/\partial x_i$

$= \text{sum}(y_j * \partial x_j/\partial x_i \text{ for } j \text{ in } \text{parents}(i))$

Derivative Outputs

$\partial f(x_1,x_2)/\partial x_1 = \partial x_5/\partial x_1$
$= y_1$

$\partial f(x_1,x_2)/\partial x_2 = \partial x_5/\partial x_2$
$= y_2$

Numerical Results

$x_1 = 1.0$		$y_1 = 0.0$	
$x_2 = 1.0$		$y_2 = 0.0$	
$x_3 = 0.8$	→	$y_3 = 0.5$	
$x_4 = 0.5$	→	$y_4 = 0.8$	
$x_5 = 0.4$		$y_5 = 1.0$	

$x_3 = \sin(x_1)$

$x_4 = \cos(x_2)$

$x_5 = x_3 * x_4$

$y_3 \mathrel{+}= y_5 * x_4$

$y_4 \mathrel{+}= y_5 * x_3$

Multivariable Chain Rule

$y_i = \partial x_5/\partial x_i$

$= \text{sum}(y_j * \partial x_j/\partial x_i \text{ for } j \text{ in parents}(i))$

Derivative Outputs

$\partial f(x_1,x_2)/\partial x_1 = \partial x_5/\partial x_1$
$= y_1$

$\partial f(x_1,x_2)/\partial x_2 = \partial x_5/\partial x_2$
$= y_2 = -0.7$

Numerical Results

$x_1 = 1.0$	$y_1 = 0.0$
$x_2 = 1.0$	$y_2 = -0.7$
$x_3 = 0.8$	$y_3 = 0.5$
$x_4 = 0.5$	$y_4 = 0.8$
$x_5 = 0.4$	$y_5 = 1.0$

$x_3 = \sin(x_1)$

$x_4 = \cos(x_2)$

$y_2 \mathrel{+}= y_4 * -(\sin(x_2))$

$x_5 = x_3 * x_4$

$y_3 \mathrel{+}= y_5 * x_4$

$y_4 \mathrel{+}= y_5 * x_3$

Multivariable Chain Rule

$y_i = \partial x_5/\partial x_i$

$= \mathrm{sum}(y_j * \partial x_j/\partial x_i \text{ for } j \text{ in parents}(i))$

Derivative Outputs

$\partial f(x_1,x_2)/\partial x_1 = \partial x_5/\partial x_1$
$= y_1 = 0.2$

$\partial f(x_1,x_2)/\partial x_2 = \partial x_5/\partial x_2$
$= y_2 = -0.7$

Numerical Results

$x_1 = 1.0$	$y_1 = 0.2$
$x_2 = 1.0$	$y_2 = -0.7$
$x_3 = 0.8$	$y_3 = 0.5$
$x_4 = 0.5$	$y_4 = 0.8$
$x_5 = 0.4$	$y_5 = 1.0$

$x_3 = \sin(x_1)$

$y_1 \mathrel{+}= y_3 * \cos(x_1)$

$x_4 = \cos(x_2)$

$y_2 \mathrel{+}= y_4 * -(\sin(x_2))$

$x_5 = x_3 * x_4$

$y_3 \mathrel{+}= y_5 * x_4$

$y_4 \mathrel{+}= y_5 * x_3$

ReverseDiff.jl

- ❑ Uses operator overloading to dynamically intercept and record native Julia code to an instruction tape.
- ❑ Re-recording allows for complex control flow (loops, recursion, etc.).
- ❑ Multiple dispatch + JIT + run-time type information enables compiled, specialized primitive execution methods
- ❑ Supports array primitives, linear algebraic derivative definitions, and most `AbstractArray` types
- ❑ Supports static forward/reverse passes over the tape with precomputed dispatch and preallocated instruction caches
- ❑ Leverages mixed-mode AD; intermediary scalar subgraphs can be automatically rewritten into new primitives differentiated via ForwardDiff. This includes application of scalar kernels via elementwise higher-order functions (e.g. map/broadcast).

A Few Realizations

"He must be a thorough fool who can learn nothing from his own folly."

- A.W. Hare

Julia Is Pretty Good At This Stuff

- ❑ Multiple dispatch + JIT compilation enables seamless and precise operator overloading with essentially no performance penalties. Auto-differentiable Julia code doesn't need to "know" about ForwardDiff/ReverseDiff, as long as the code is numerically type-generic.

- ❑ Writing data-flow semantics in Julia over a Julia-represented DAG means grants efficient nested data-flow semantics for “free”.

- ❑ Since primitive definition and execution is performed via normal Julia dispatch, no magic is required to extend or create primitives.

- ❑ We can get heterogeneous device support for "free" (e.g. via GPUArrays), and hardware-specialized primitives can easily be added via dispatch

ReverseDiff For JuMP?

- Cons vs. ReverseDiffSparse:
  - ReverseDiffSparse, as the name implies, does indeed exploit Hessian sparsity
  - ReverseDiffSparse has better variable storage locality for scalar operations

- Pros vs. ReverseDiffSparse:
  - ReverseDiffSparse doesn’t support array primitives
  - ReverseDiffSparse doesn’t directly support native Julia code
  - ReverseDiffSparse isn’t numerically type-generic
  - ReverseDiffSparse can’t easily handle nested differentiation

- Takeaway: ReverseDiff is more versatile and extensible, but ReverseDiffSparse has some important performance optimizations for tackling large-scale problems

ReverseDiff For Deep Learning?

- ReverseDiff’s API doesn’t expose variable construction directly
  - ...though internal utilities are similar to PyTorch’s/TensorFlow’s exposed APIs
- ReverseDiff’s dynamic recording mechanism is implemented as if its performance costs can be amortized w.r.t. whole computations
  - ...which is true for static graphs in traditional optimization
  - ...but is untrue for dynamic graphs in deep learning
- Different Graph Regimes
  - Optimization: Many nodes, computationally cheap scalar operations
  - Deep Learning: Fewer nodes, computationally expensive array operations
  - This is why ML people are cool with fully dynamic taping methods - traversal overhead is negligble

ReverseDiff For...Not AD?

- A native-Julia-to-DAG package would be generally useful outside of AD
  - Dynamic code analysis/optimization
  - Parallel operation scheduling
  - Automatic pre-allocation/memory management
  - Interval constraint programming
  - Serialization of Julia code to other DAG frameworks

- It would require generalizing ReverseDiff’s taping/execution mechanisms to support arbitrary metadata propagation

- It could also enable better AD anyway (e.g. edge-pushing algorithm for sparse Hessians, which requires propagating dependency information)

Enter *Cassette.jl*

What is Cassette?

- A native-Julia-to-DAG data-flow package for forward/backward-propagating values and arbitrary metadata through pure-Julia computation graphs.
- Inspired by both deep learning and traditional optimization worlds - different representations are supported for static and dynamic graphs
- Exposes taping/execution mechanisms to downstream library authors as a hijackable processing pipeline. Cassette primitives should be easy to define and extend.
- The next version of ReverseDiff is Cassette’s prototypical application

Acknowledgements

- *Juan Pablo Vielma* @ MIT Operations Research Center
- *Cosmin Petra* @ Lawrence Livermore National Lab
- The Julia Group @ MIT CSAIL: *Alan Edelman*, *Andreas Noack*, *Peter Ahrens*
- *Robin Deits* @ MIT CSAIL
- JuMP Meetup Organizers: *Miles Lubin*, *Chris Coey*, *Jennifer Challis*