Mixed-Mode Automatic Differentiation in julia

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

Hi, I’m Jarrett

- ❏ Started writing Julia code in 2013, working on AD in Julia since 2015
- ❏ Authored Julia's performance regressions testing facilities (*BenchmarkTools*)
- ❏ Downstream packages: *JuMP*, *Celeste*, *Optim*, *DifferentialEquations*, *RigidBodyDynamics*, *ValidatedNumerics*, etc…
- ❏ Previously worked in the Julia Group @ CSAIL under Alan Edelman
- ❏ Recently started a new position under Juan Pablo Vielma @ MIT ORC
- ❏ Currently focused on AD, maybe transitioning to JuMP development in the fall

Last Year’s Talk: ForwardDiff.jl

- ❏ Implements multidimensional dual numbers
- ❏ Fully stack-allocated and aggressively inlined, plays well with SIMD
- ❏ Provides a differentiation API instead of exposing dual numbers directly
- ❏ Tagging system prevents perturbation confusion and drives nested differentiation

This Year’s Talk: More AD Stuff

❏ Description of reverse-mode AD

❏ Some reflections on ReverseDiff.jl

❏ A new thing I’m building that will hopefully solve most of my problems

Reverse-Mode AD

Compared to Forward-Mode AD

- Propagating input perturbation forward ⟶ propagate 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 by running code + intercepting operations (ReverseDiff, Autograd, PyTorch)
- Which mode should I use?
  - `output_dimension > input_dimension || input_dimension << code_size` ⟶ Use forward mode
  - `output_dimension < input_dimension && input_dimension >> code_size` ⟶ Use reverse mode
  - `output_dimension ≈ input_dimension` ⟶ That’s tough

Code Representation

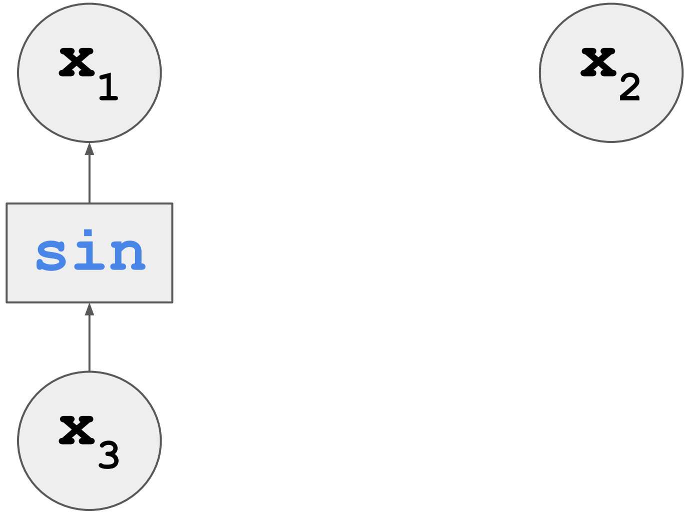
```
function f(x₁, x₂)
    # ?
end
```

Graph Representation

Code Representation

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

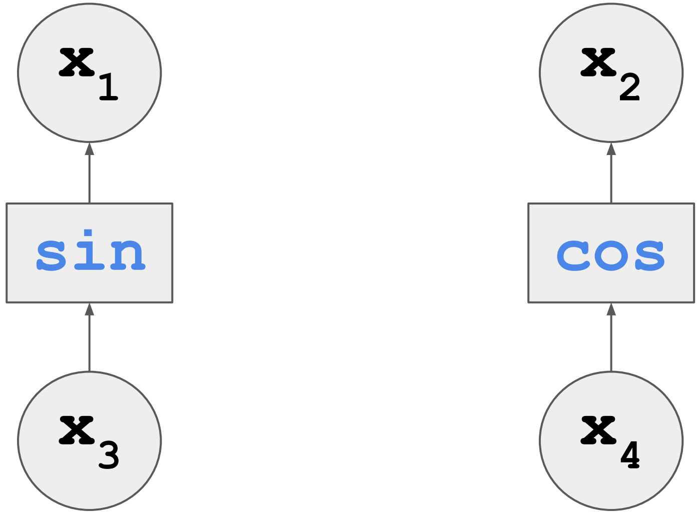
Graph Representation



Code Representation

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

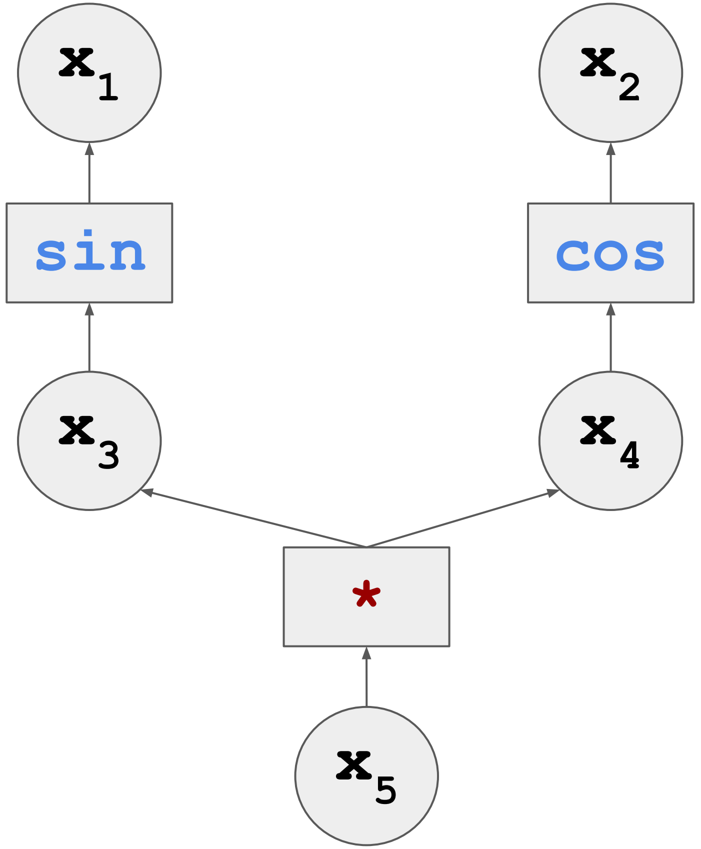
Graph Representation



Code Representation

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

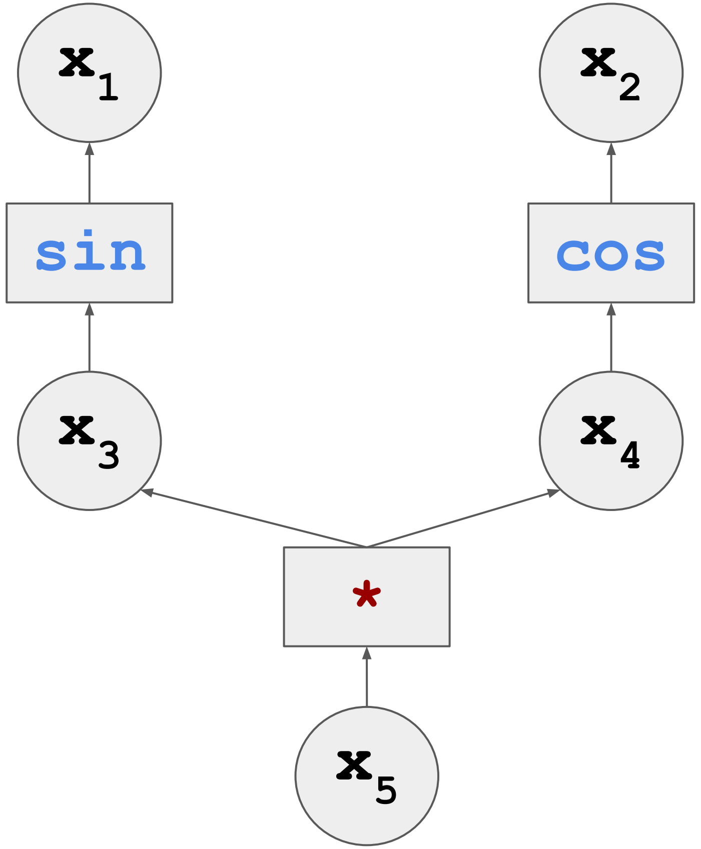
Graph Representation

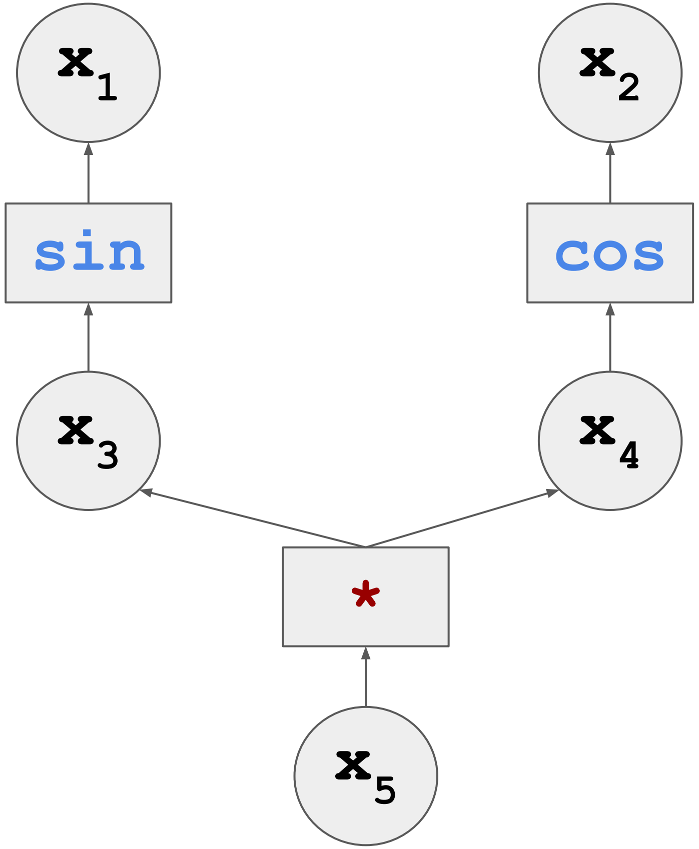


Code Representation

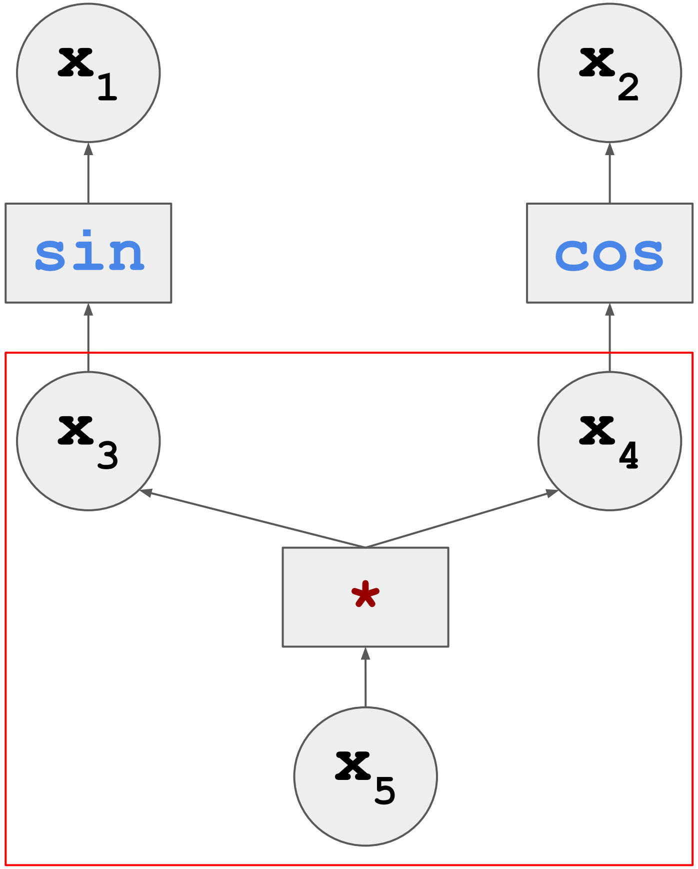
```
function f(x₁, x₂)
    x₃ = sin(x₁)
    x₄ = cos(x₂)
    x₅ = x₃ * x₄
    return x₅
end
```

Graph Representation

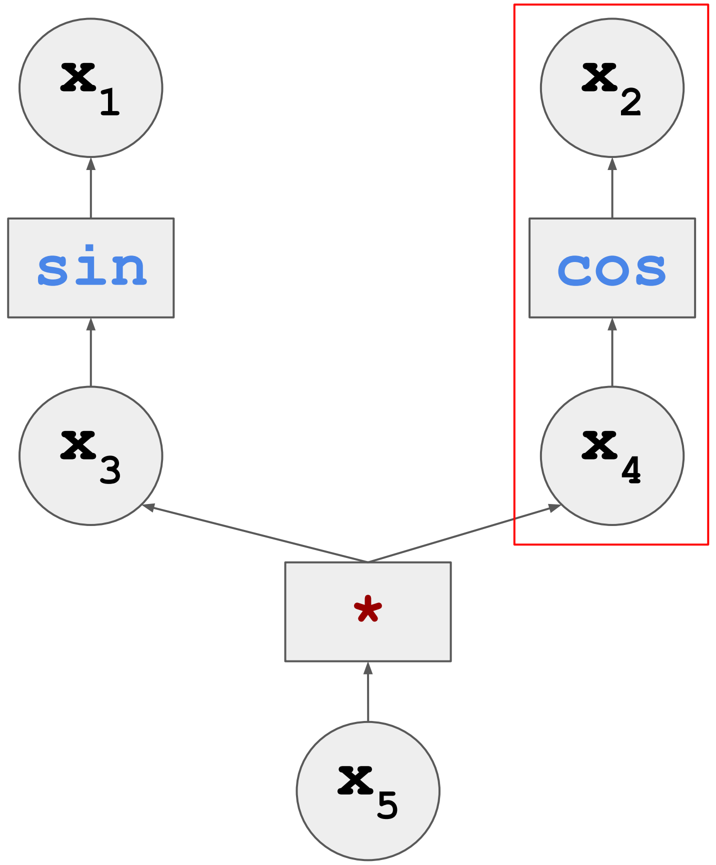




x1
x2
sin
cos
x3
x4
*
x5

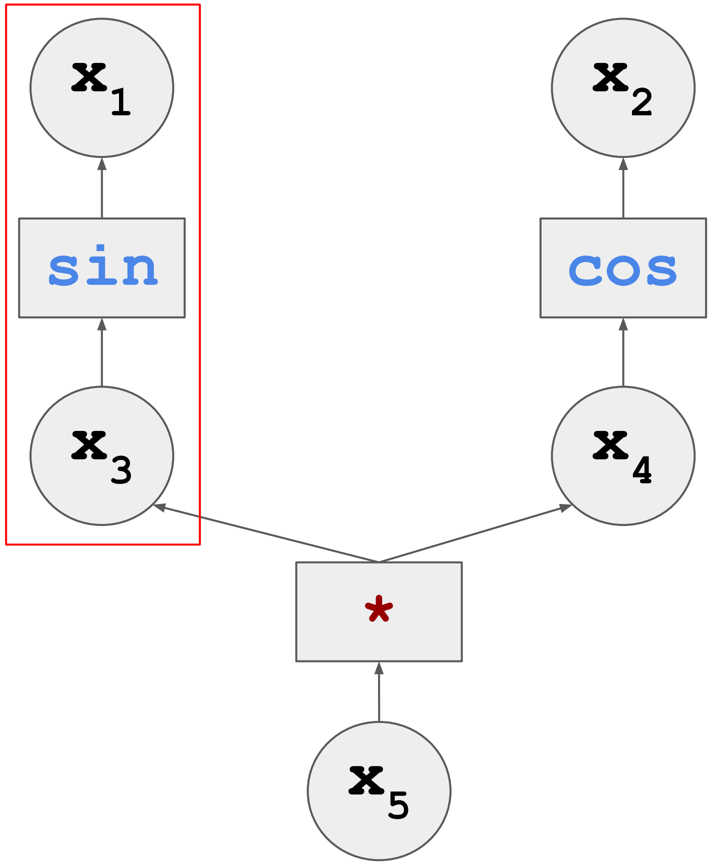


$x_5 = x_3 * x_4$



$x_4 = \cos(x_2)$

$x_5 = x_3 * x_4$



```
x₃ = sin(x₁)
```

```
x₄ = cos(x₂)
```

```
x₅ = x₃ * x₄
```

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

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

$x_3 = \sin(x_1)$

$x_4 = \cos(x_2)$

$x_5 = x_3 * x_4$

Multivariable Chain Rule

$y_i = \partial x_5/\partial x_i$

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

Derivative Outputs

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

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

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	= 0.0	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

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

$x_3 = \sin(x_1)$

$\boxed{x_4 = \cos(x_2)}$

$x_5 = x_3 * x_4$

$\downarrow$

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

$y_i = \partial x_5/\partial x_i$

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

Derivative Outputs

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

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

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

$y_i = \partial x_5 / \partial x_i$

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

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

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

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

Numerical Results

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

$x_3 = \sin(x_1)$

$x_4 = \cos(x_2)$

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

$x_5 = x_3 * x_4$

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

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

$\uparrow$

Multivariable Chain Rule

$$y_i = \partial x_5/\partial x_i$$

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

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
- ❑ Multiple dispatch + JIT + run-time type information enables compiled, specialized primitives
- ❑ Supports array primitives, linear algebraic derivative definitions, and most **`AbstractArray`** types
- ❑ Supports dynamic forward pass (re-recording allows for complex control flow - loops, recursion, etc.)
- ❑ Supports static forward/reverse passes over tape (precomputed dispatch + preallocated instruction caches)
- ❑ Mixed-mode AD! Scalar subgraphs automatically differentiated via ForwardDiff. Includes scalar kernels of elementwise 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

- ❏ Seamless/precise operator overloading with no performance penalty
- ❏ Target code can be mostly “AD-unaware”; just needs to be numerically type-generic.
- ❏ Primitives defined via normal Julia code - no magic for creation/extension
- ❏ Writing data-flow semantics in Julia over a Julia-represented DAG means grants efficient nested data-flow semantics for “free”.
- ❏ Heterogeneous device support for "free" (e.g. GPUArrays)

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 support dynamic graphs
  - 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 writes to a static graph representation
  - ...great for recording traditional optimization graphs
  - ...not so great for recording 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 negligible

ReverseDiff For...Not AD?

- A native-Julia trace-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

- It could also enable better AD anyway (e.g. edge-pushing algorithm for sparse Hessians)

Enter *Cassette.jl*

“Multiple dispatch is dead, long live multiple dispatch!”
- Anonymous

What is Cassette?

- ❏ A native Julia execution tracer + data flow package for propagating values and arbitrary metadata through pure-Julia computation graphs.
- ❏ Inspired by both traditional optimization and deep learning worlds - different representations are supported for static and dynamic graphs
- ❏ Exposes trace interception mechanisms to downstream library authors as a hijackable processing pipeline.
- ❏ The next version of ReverseDiff is Cassette's prototypical application
- ❏ ***Doesn't rely on argument type propagation to intercept function calls!***

Multiple Dispatch Is Dead…

```julia
# we'll define primitives for this on the next slide
struct Interceptor{T,N} <: AbstractArray{T,N}
    data::AbstractArray{T,N}
end
```

Multiple Dispatch Is Dead…

```
# we'll define primitives for this on the next slide
struct Interceptor{T,N} <: AbstractArray{T,N}
     data::AbstractArray{T,N}
end

# primitives defined on `Interceptor` will just call this
struct Intercepted{F} <: Function
     func::F
end

unwrap(x) = x
unwrap(i::Interceptor) = i.data
unwrap(i::Intercepted) = i.func

(i::Intercepted{F})(args...) = (println(“called $F”);  unwrap(i)(unwrap.(args)...))
```

Multiple Dispatch Is Dead…

```
const AMBIGUOUS_TYPES = [subtypes(AbstractArray)...]

#### 1-arg primitive --> 1 method ################################
Base.f(x::Interceptor) = Intercepted(f)(x)
```

Multiple Dispatch Is Dead...

```
const AMBIGUOUS_TYPES = [subtypes(AbstractArray)...]

#### 1-arg primitive --> 1 method ###############################
Base.f(x::Interceptor) = Intercepted(f)(x)

#### 2-arg primitive --> ~50 methods! ###########################
Base.f(x::Interceptor, y::Interceptor) = Intercepted(f)(x, y)
for T in AMBIGUOUS_TYPES
     Base.f(x::Interceptor, y::T) = Intercepted(f)(x, y)
     Base.f(x::T, y::Interceptor) = Intercepted(f)(x, y)
end
```

Multiple Dispatch Is Dead...

```
const AMBIGUOUS_TYPES = [subtypes(AbstractArray)...]

#### 1-arg primitive --> 1 method ##############################
Base.f(x::Interceptor) = Intercepted(f)(x)

#### 2-arg primitive --> ~50 methods ###########################
Base.f(x::Interceptor, y::Interceptor) = Intercepted(f)(x, y)
for T in AMBIGUOUS_TYPES
     Base.f(x::Interceptor, y::T) = Intercepted(f)(x, y)
     Base.f(x::T, y::Interceptor) = Intercepted(f)(x, y)
end

#### 3-arg primitive --> ~2000 methods!!! ######################
Base.f(x::Interceptor, y::Interceptor, z::Interceptor) = Intercepted(f)(x, y, z)
for T in AMBIGUOUS_TYPES
     Base.f(x::Interceptor, y::Interceptor, z::T) = Intercepted(f)(x, y, z)
     Base.f(x::Interceptor, y::T, z::Interceptor) = Intercepted(f)(x, y, z)
     Base.f(x::T, y::Interceptor, z::Interceptor) = Intercepted(f)(x, y, z)
     for S in AMBIGUOUS_TYPES
          Base.f(x::Interceptor, y::T, z::S) = Intercepted(f)(x, y, z)
          Base.f(x::T, y::Interceptor, z::S) = Intercepted(f)(x, y, z)
          Base.f(x::T, y::S, z::Interceptor) = Intercepted(f)(x, y)
     end
end
```

Multiple Dispatch Is Dead...

```
const AMBIGUOUS_TYPES = [subtypes(AbstractArray)...]

#### 1-arg primitive --> 1 method ###############################
Base.f(x::Interceptor) = Intercepted(f)(x)

#### 2-arg primitive --> ~50 methods ############################
Base.f(x::Interceptor, y::Interceptor) = Intercepted(f)(x, y)
for T in AMBIGUOUS[illegible]
     Base.f(x::I[illegible]ercep[illegible], y[illegible]) = [illegible]erc[illegible]ed(f[illegible] y)
     Base.f(x:[illegible] y::Interc[illegible]r) = [illegible]er[illegible]pted(f) [illegible] y)
end

#### 3-arg prim[illegible]ve --> [illegible]200[illegible]metho[illegible]###[illegible]
Base.f(x::Interce[illegible]:In[illegible]tor, [illegible]:In[illegible]tor) [illegible]pted([illegible] z)
for T in AMBIGUOUS_TYPES
     Base.f(x::Interceptor, y::Interceptor, z::T) = Intercepted(f)(x, y, z)
     Base.f(x::Interceptor, y::T, z::Interceptor) = Intercepted(f)(x, y, z)
     Base.f(x::T, y::Interceptor, z::Interceptor) = Intercepted(f)(x, y, z)
     for S in AMBIGUOUS_TYPES
          Base.f(x::Interceptor, y::T, z::S) = Intercepted(f)(x, y, z)
          Base.f(x::T, y::Interceptor, z::S) = Intercepted(f)(x, y, z)
          Base.f(x::T, y::S, z::Interceptor) = Intercepted(f)(x, y)
     end
end
```

GROSS

...Long Live Multiple Dispatch!

```
#### AST pruning pseudocode ##############################

function code_info_with_intercepted_calls(Tuple{F,A,B,C...})
     # 1. Get `CodeInfo` for signature `Tuple{F,A,B,C}`
     # 2. Walk through SSA-form AST and wrap all calls with `Intercepted`
     # 3. Return the modified `CodeInfo`
end

#### `Trace` function wrapper ############################

struct Trace{F,world} <: Function
     func::F
     Trace(func::F) where {F} = new{F,get_world_counter()}(func)
end

@generated function (t::Trace{F,world})(args...) where {F,world}
     return code_info_with_intercepted_calls(F, args...)
end
```

...Long Live Multiple Dispatch!

```julia
function f(x)
    a = one(eltype(x))
    b = 100 * a
    result = zero(eltype(x))
    for i in 1:length(x)
       result += b * (a - x[i])
    end
    return result
end
```

...Long Live Multiple Dispatch!

```
function f(x)
    a = one(eltype(x))
    b = 100 * a
    result = zero(eltype(x))
    for i in 1:length(x)
       result += b * (a - x[i])
    end
    return result
end

# ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

function (::Trace{typeof(f)})(x)
    a = Intercepted(one)(Intercepted(eltype)(x))
    b = Intercepted(*)(100, a)
    result = Intercepted(zero)(Intercepted(eltype)(x))
    for i in Intercepted(UnitRange)(1, Intercepted(length)(x))
       result = Intercepted(+)(result, Intercepted(*)(b, Intercepted(-)(a,
                Intercepted(getindex)(x, i))))
    end
    return result
end
```

...Long Live Multiple Dispatch!

```
function f(x)
    a = one(eltype(x))
    b = 100 * a
    result = zero(eltype(x))
    for i in 1:length(x)
       result += b * (a - x[i])
    end
    return result
end

# ↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓↓

function (::Trace{typeof(f)})(x)
    a = Intercepted(one)(Intercepted(eltype)(x))
    b = Intercepted(*)(100, a)
    result = Intercepted(zero)(Intercepted(eltype)(x))
    for i in Intercepted(UnitRange)(1, Intercepted(length)(x))
       result = Intercepted(+)(result, Intercepted(*)(b, Intercepted(-)(a,
                Intercepted(getindex)(x, i))))
    end
    return result
end

(i::Intercepted)(ws::MyWrapper...) = MyWrapper.(unwrap(i)(unwrap.(ws)...))
```

…Long Live Multiple Dispatch!

- ❑ No need for target functions to be type generic
- ❑ No need to define an ungodly number of methods per primitive
- ❑ No need to define new number/array/etc. types just to propagate metadata or hijack execution flow
- ❑ Hijack behavior can be overloaded via normal Julia dispatch of downstream function wrappers
- ❑ In the future, we can also wrap SSA-form control flow instructions

The Future

- ❏ Finish + document + test + release Cassette (targeting Julia v0.7)
- ❏ Replace ForwardDiff/ReverseDiff with new Cassette-based packages
- ❏ Replace ReverseDiffSparse → new backend for JuMP
  - ❏ Locality + sparse Hessian optimizations for Cassette graphs
- ❏ Evangelize Cassette for other regimes

Acknowledgements

- ❏ *Juan Pablo Vielma, Miles Lubin* @ MIT Operations Research Center
- ❏ *Cosmin Petra* @ Lawrence Livermore National Lab
- ❏ The Julia Group @ MIT CSAIL: *Alan Edelman*, *Andreas Noack*, *Peter Ahrens*
- ❏ *Jameson Nash, Mike Innes* @ Julia Computing (...and everybody else there as well!)
- ❏ *Simon Danisch* @ GPUArrays, Inc
- ❏ *Robin Deits* @ MIT CSAIL