# ForwardDiff.jl

Fast Derivatives Made Easy

# Overview

I. Numerical Derivatives 101

II. Going to Higher Dimensions

III. ForwardDiff In Action

IV. Benchmarks vs. `autograd`

V. Pitfalls

VI. Future Work

VII. Resources

VIII. Acknowledgements

# Numerical Derivatives 101

Let's say I give you an implementation of a function $f : \mathbb{R} \to \mathbb{R}$ whose code you **didn't write** and **can't read**. How do you calculate $f'(x)$?

# Finite Difference Method

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h} = \boxed{\frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2)}$$

**Advantages**

- works “out of the box”

**Disadvantages**

- $\mathcal{O}(h^2)$ truncation error
- $h$ too small $\rightarrow$ subtractive cancellation error
- requires 2 evaluations of $f$

# Complex Step Method

$$f(x+ih) = f(x)+f'(x)hi+\frac{f''(x)}{2!}h^2i^2\ldots = f(x) + f'(x)hi - \frac{f''(x)}{2!}h^2\ldots$$

$$f'(x) = \frac{\mathrm{Im}[f(x+hi)]}{h} + \mathcal{O}(h^2)$$

**Advantages**

- no subtractive cancellation error
- Doesn't need 2 calls to $f$
- A lot of languages offer efficient complex number implementations
- truncation error can be close to machine epsilon in practice

**Disadvantages**

- complex number operations can be much slower than real number operations
- unsafe for programs which already incorporate complex inputs
- requires operator overloading and/or source code transformation

# Dual Number Method

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

$$= f(x) + f'(x)y\epsilon$$

$$\boxed{f'(x) = \mathrm{Eps}[f(x+\epsilon)]}$$

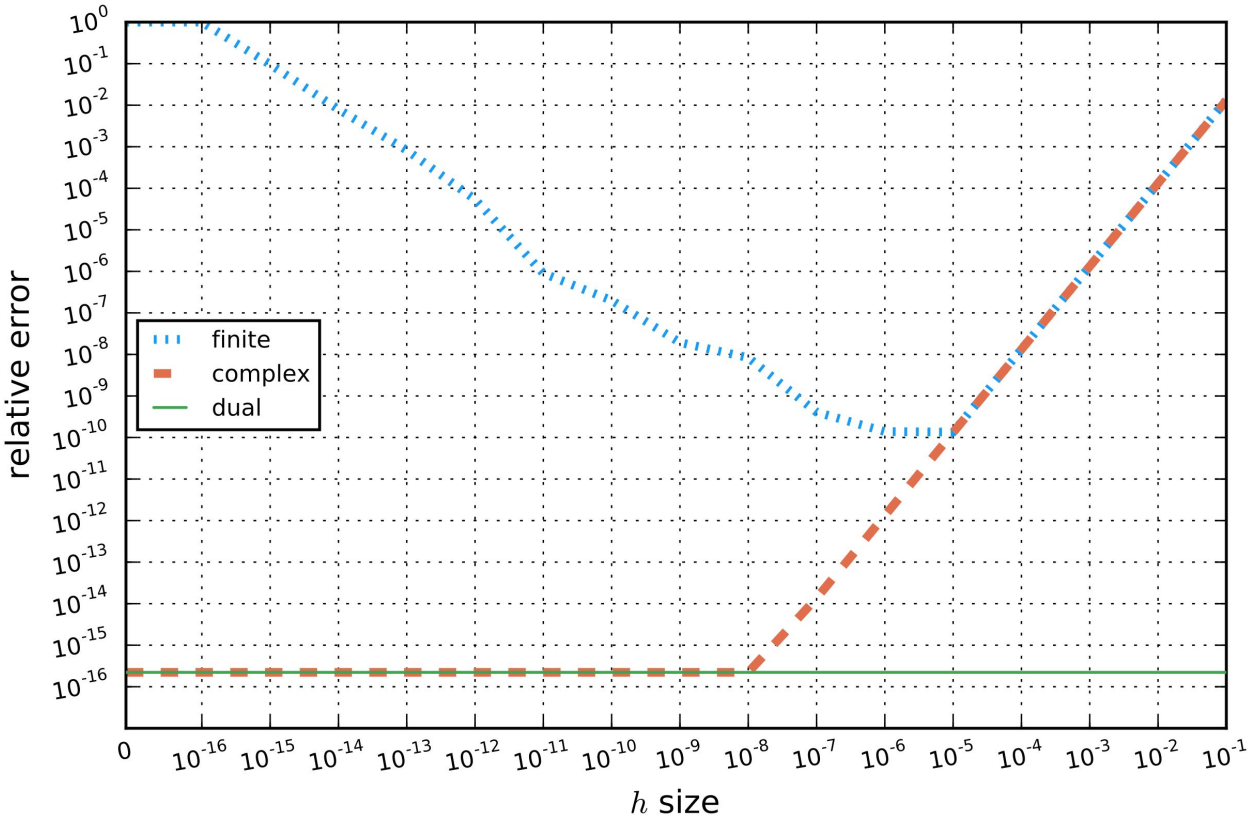
**Advantages**

- no approximation error

**Disadvantages**

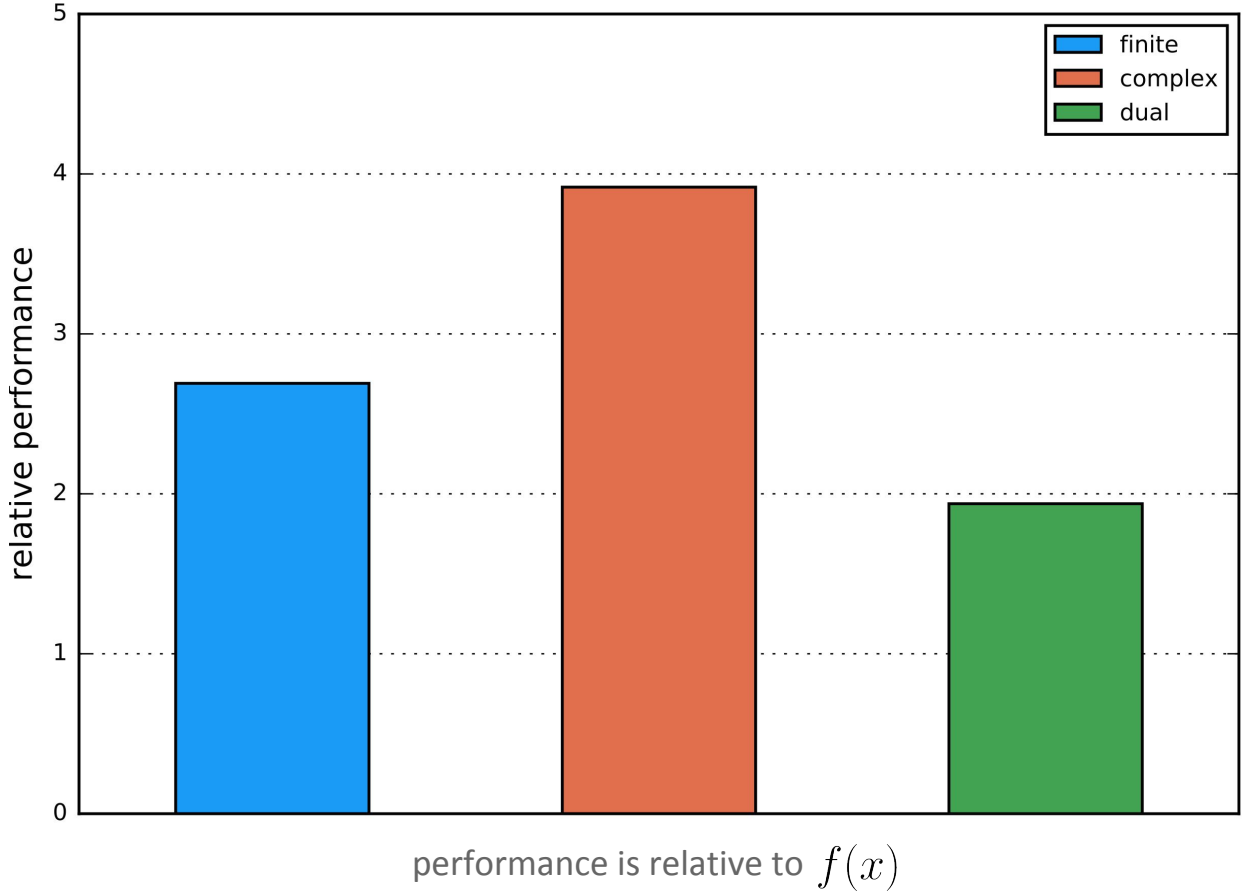
- target program must accept generic number types
- requires operator overloading and/or source code transformation

# Error Comparison



# Performance Comparison

$$f(x) = \frac{e^x}{\sqrt{\sin^3(x) + \cos^3(x)}}$$



# Going to Higher Dimensions

First-order derivatives of scalar functions are **boring**. What about functions like $g : \mathbb{R}^n \to \mathbb{R}$ or $\mathbf{g} : \mathbb{R}^n \to \mathbb{R}^m$? What about **gradients** and **Jacobians**?

# Gradients

$$\nabla g(\mathbf{x}) = \begin{bmatrix} \frac{\partial g(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_i} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

# Gradients

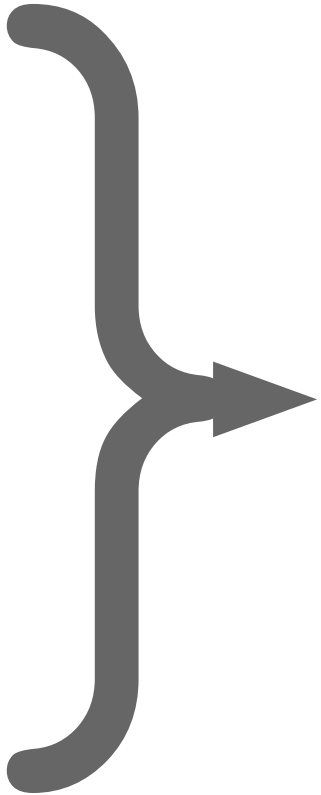
$$\nabla g(\mathbf{x}) = \begin{bmatrix} \frac{\partial g(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_i} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

$$f(x + \epsilon) = f(x) + f'(x)\epsilon$$

# Gradients

$$\nabla g(\mathbf{x}) = \begin{bmatrix} \frac{\partial g(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_i} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

$$f(x+\epsilon) = f(x) + f'(x)\epsilon$$



# Gradients

$$\nabla g(\mathbf{x}) = \begin{bmatrix} \frac{\partial g(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_i} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

$$f(x+\epsilon) = f(x) + f'(x)\epsilon$$

$$g\left(\begin{bmatrix} x_1 \\ \vdots \\ x_i + \epsilon \\ \vdots \\ x_n \end{bmatrix}\right) = g(\mathbf{x}) + \frac{\partial g(\mathbf{x})}{\partial x_i}\epsilon$$

# Gradients

$$\nabla g(\mathbf{x}) = \begin{bmatrix} \frac{\partial g(\mathbf{x})}{\partial x_1} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_i} \\ \vdots \\ \frac{\partial g(\mathbf{x})}{\partial x_n} \end{bmatrix}$$

$$f(x+\epsilon) = f(x) + f'(x)\epsilon$$

But do we really need $n$ calls to $g$?

$$g\left(\begin{bmatrix} x_1 \\ \vdots \\ x_i + \epsilon \\ \vdots \\ x_n \end{bmatrix}\right) = g(\mathbf{x}) + \frac{\partial g(\mathbf{x})}{\partial x_i}\epsilon$$

# Extending Dual Numbers

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

# Extending Dual Numbers

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

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

# Extending Dual Numbers

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

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

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

# Extending Dual Numbers

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

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

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

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

# Jacobians

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

# Jacobians

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

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

# Jacobians

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

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

# ForwardDiff In Action

# The `Dual` Type

```
# special fixed-sized vector for storing partial derivatives
immutable Partials{N,T}
    values::NTuple{N,T}
end

# type definition of a dual number
immutable Dual{N,T<:Real} <: Real
    value::T
    partials::Partials{N,T}
end
```

# The `Dual` Type

```
# special fixed-sized vector for storing partial derivatives
immutable Partials{N,T}
    values::NTuple{N,T}
end

# type definition of a dual number
immutable Dual{N,T<:Real} <: Real
    value::T
    partials::Partials{N,T}
end

# defined to obey `Real` ordering
Base.isless(a::Dual, b::Dual) = a.value < b.value
```

# The `Dual` Type

```
# special fixed-sized vector for storing partial derivatives
immutable Partials{N,T}
    values::NTuple{N,T}
end

# type definition of a dual number
immutable Dual{N,T<:Real} <: Real
    value::T
    partials::Partials{N,T}
end

# defined to obey `Real` ordering
Base.isless(a::Dual, b::Dual) = a.value < b.value

# overload math functions to propagate derivatives to `Partials`
Base.sin(d::Dual) = Dual(sin(d.value), cos(d.value) * d.partials)
Base.:*(a::Dual, b::Dual) = Dual(a.value * b.value,
                                 b.value * a.partials + a.value * b.partials)
```

# The `Dual` Type

```
# special fixed-sized vector for storing partial derivatives
immutable Partials{N,T}
    values::NTuple{N,T}
end

# type definition of a dual number
immutable Dual{N,T<:Real} <: Real
    value::T
    partials::Partials{N,T}
end

# defined to obey `Real` ordering
Base.isless(a::Dual, b::Dual) = a.value < b.value

# overload math functions to propagate derivatives to `Partials`
Base.sin(d::Dual) = Dual(sin(d.value), cos(d.value) * d.partials)
Base.:*(a::Dual, b::Dual) = Dual(a.value * b.value,
                                 b.value * a.partials + a.value * b.partials)

# ...and all of the other methods one needs to define for new number types
```

# Jacobian of **cumprod**

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

```
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
```

# Jacobian of **cumprod**

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

# Jacobian of **`cumprod`**

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

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

# Jacobian of **cumprod**

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



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



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

# Jacobian of **`cumprod`**

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

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

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

```
julia> using ForwardDiff: Dual

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

# Jacobian of **cumprod**

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

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

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

```
julia> using ForwardDiff: Dual

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{ForwardDiff.Dual{3,Int64},1}:
 Dual(1,1,0,0)
 Dual(2,2,1,0)
 Dual(6,6,3,2)
```

# Jacobian of **cumprod**

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

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

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

```
julia> using ForwardDiff: Dual

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{ForwardDiff.Dual{3,Int64},1}:
 Dual(1,1,0,0)
 Dual(2,2,1,0)
 Dual(6,6,3,2)

julia> ForwardDiff.jacobian(cumprod, [1,2,3])
3x3 Array{Int64,2}:
 1  0  0
 2  1  0
 6  3  2
```

# Benchmarking vs. `autograd`

The Python package **autograd** is a popular **reverse-mode** automatic differentiation tool. Reverse-mode AD is algorithmically more efficient than Forward-mode AD for taking gradients of functions $g : \mathbb{R}^n \to \mathbb{R}$.

# Benchmarking v.s. **autograd**

$$\text{Rosenbrock}(\vec{x}) = \sum_{i=1}^{k-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2$$

| Function | Input Size | autograd Time (s) | ForwardDiff Time (s) | Ratio |
|---|---|---|---|---|
| Rosenbrock | 10 | 0.000866 | 0.000001 | 866.0 |
| Rosenbrock | 100 | 0.004395 | 0.000028 | 156.96 |
| Rosenbrock | 1000 | 0.040702 | 0.002605 | 15.62 |
| Rosenbrock | 10000 | 0.411095 | 0.257495 | 1.60 |
| Rosenbrock | 100000 | 4.173851 | 26.596339 | 0.16 |

# Benchmarking v.s. **`autograd`**

$$\text{Ackley}(\vec{x}) = -a \exp\left(-b\sqrt{\frac{1}{k}\sum_{i=1}^{k} x_i^2}\right) - \exp\left(\frac{1}{k}\sum_{i=1}^{k}\cos(cx_i)\right) + a + \exp(1)$$

| Function | Input Size | autograd Time (s) | ForwardDiff Time (s) | Ratio |
|---|---|---|---|---|
| Ackley | 10 | 0.001204 | 0.000001 | 1204.00 |
| Ackley | 100 | 0.008472 | 0.000048 | 176.50 |
| Ackley | 1000 | 0.081499 | 0.004925 | 16.55 |
| Ackley | 10000 | 0.835441 | 0.516848 | 1.65 |
| Ackley | 100000 | 8.361769 | 52.337054 | 0.15 |

# Pitfalls

# Target Function Restrictions

# Target Function Restrictions

1. **The function must only be composed of generic Julia code** (e.g. no LAPACK)

# Target Function Restrictions

1. **The function must only be composed of generic Julia code** (e.g. no LAPACK)

2. **The function must accept** `Real` **or** `Array{T<:Real}` **inputs**

# Target Function Restrictions

1. **The function must only be composed of generic Julia code** (e.g. no LAPACK)

2. **The function must accept** `Real` **or** `Array{T<:Real}` **inputs**

3. **The function should be type-stable** (okay, not strictly, but it really should be)

# 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
```

```
const D = ForwardDiff.derivative

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

# what ForwardDiff will compute
d2 = D(x -> x * D(y -> x + y, 1), 1)
d2 = D(x -> x * epsilon(x + (1 + ϵ)), 1)
d2 = epsilon((1 + ϵ) * epsilon((1 + ϵ) + (1 + ϵ)))
d2 = epsilon((1 + ϵ) * epsilon(2 + 2ϵ))
d2 = epsilon((1 + ϵ) * 2)
d2 = epsilon(2 + 2ϵ)
d2 = 2 # != d1
```

# Future Work

1. **Fix Perturbation Confusion. Tagging System? Perturbation Interception?**

2. **SIMD for the `Partials` type**

3. **Overload matrix functions, provide API for derivative injection**

4. **Reverse-mode AD?**

# Resources

- Presentation repository: https://github.com/jrevels/ForwardDiffPresentation
- ForwardDiff repository: https://github.com/JuliaDiff/ForwardDiff.jl
- AD research web portal: http://www.autodiff.org/
- Complex Step Method: http://www.math.u-psud.fr/~maury/paps/NUM_CompDiff.pdf
- Finite Difference Method: http://ocw.usu.edu/Civil_and_Environmental_Engineering/Numerical_Methods_in_Civil_Engineering/ODEsMatlab.pdf
- Perturbation Confusion: http://www.bcl.hamilton.ie/~barak/papers/ifl2005.pdf

# Acknowledgements


- **The Julia Group:** Alan Edelman, Jiahao Chen, Andreas Noack, Oscar Blumberg, and others
- **Steven G. Johnson and The Nanosoldier Grant**
- **Original Project Mentors:** Miles Lubin, Theodore Papamarkou
- **Contributors to ForwardDiff:** Kristoffer Carlsson, Tim Holy, and others
- **JuliaCon Sponsors and Organizers**