Libraries and Tools:

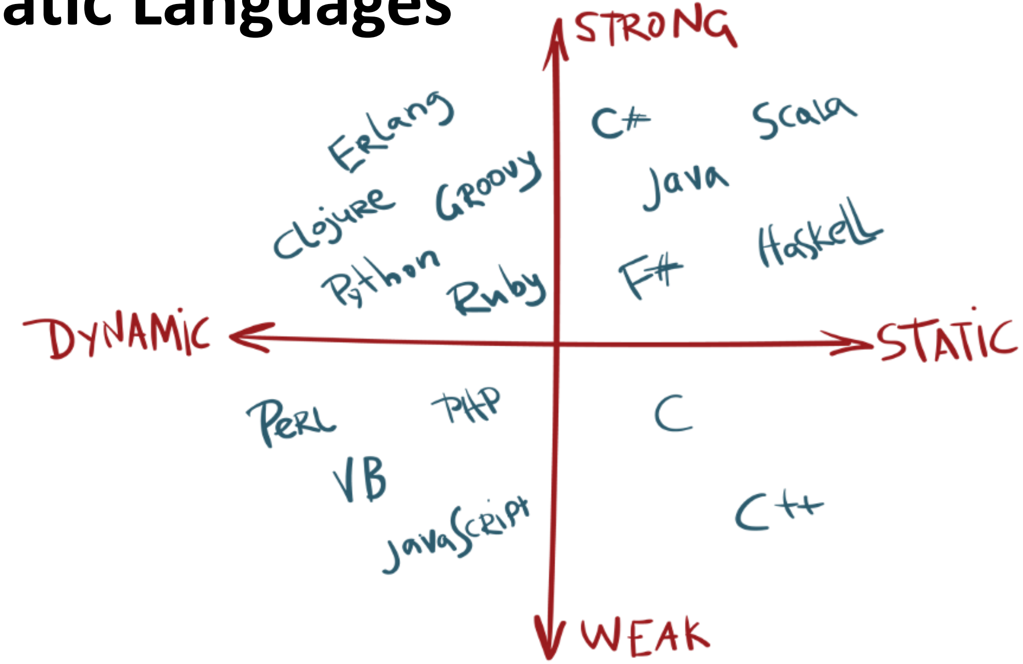
Which is Better for Numerical Linear Algebra?

Group 3
Roshan Karande, Huining Chen
June 9, 2022

Sec 1: Introduction

The Evolution of Programming Languages

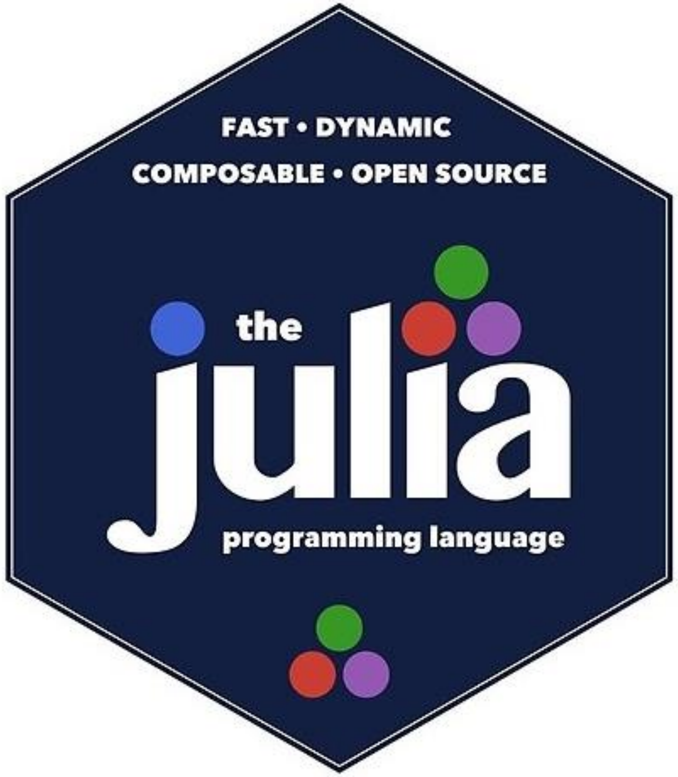
- **Quest**: High-Level Generic Formulas → Low-Level Efficient Code
- **Dynamic vs Static Languages**



Plot 1 - Comparison of Dynamic and Static Languages

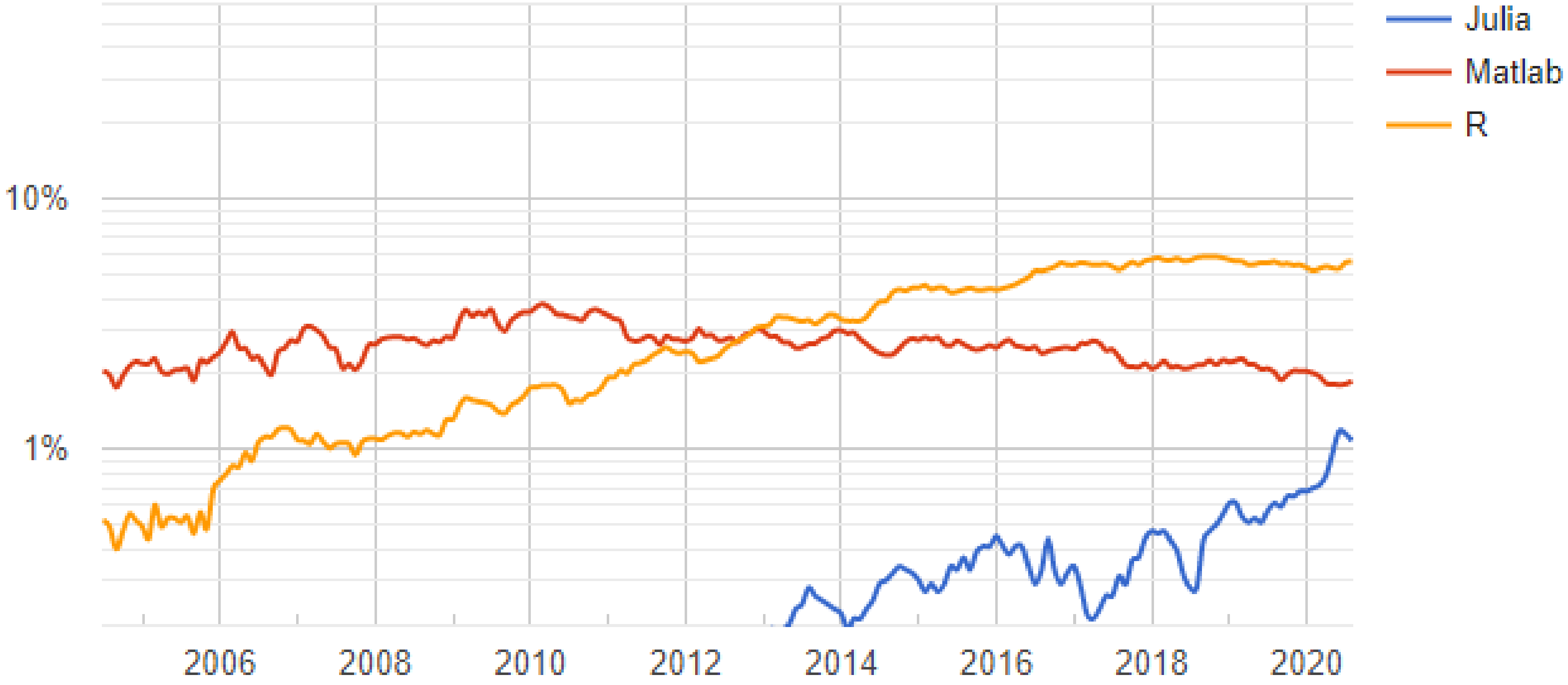
A New Language Design Philosophy

- Goal: **High Performance**
- Writes its own libraries
- Makes possible all basic functionality



Importance

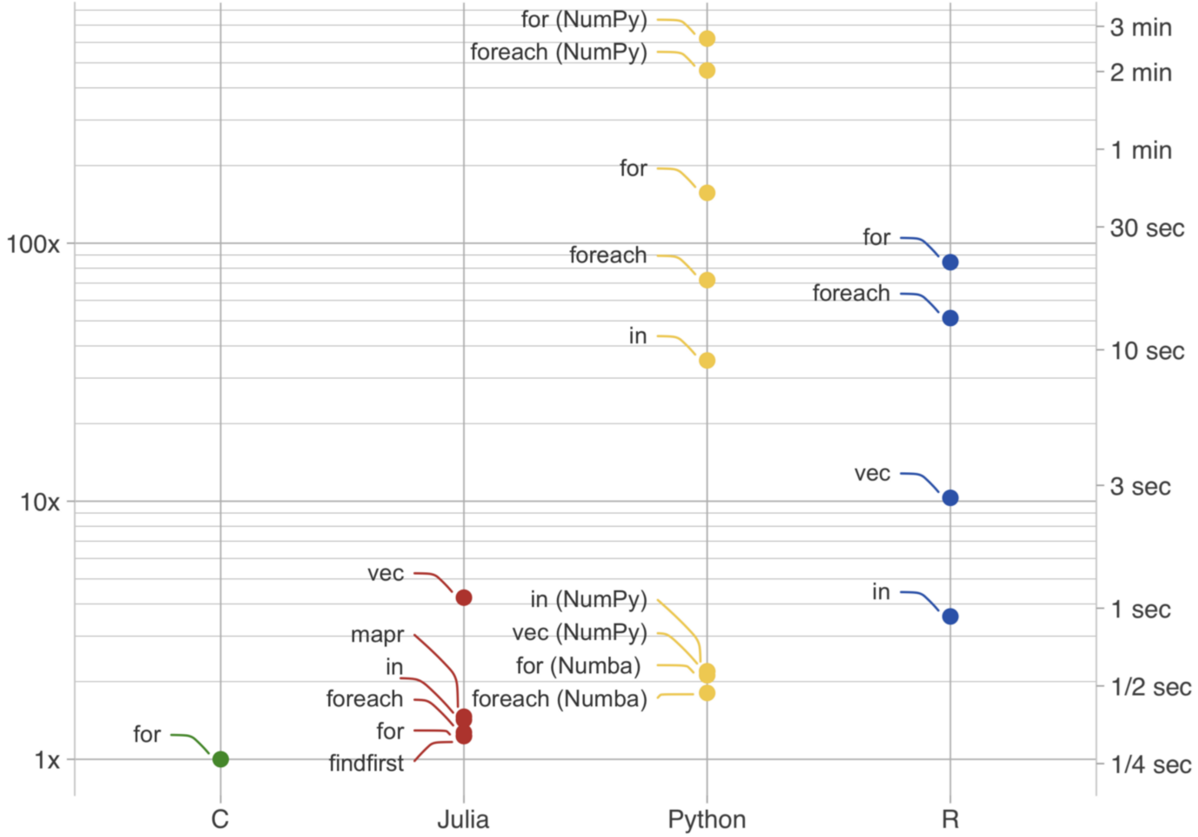
- **Efficient scientific computing without sacrificing productivity:**



Plot 2 - Comparison of programming language usage in the USA over time for Matlab, Julia, and R.

Sec 2: Problem Formulation

Julia vs Python

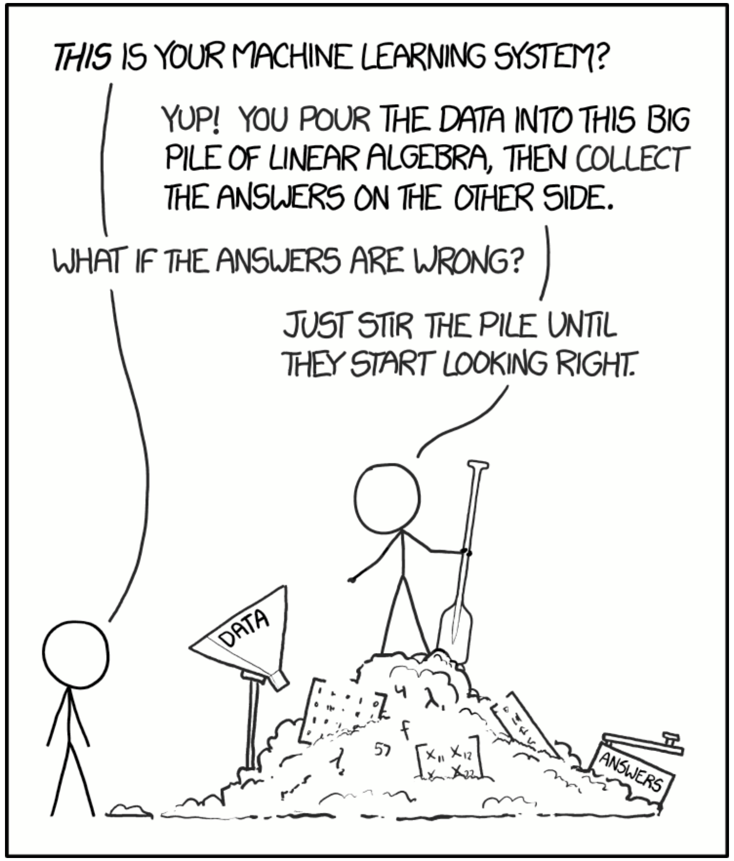


Plot 3 - CPU time relative to C

- **Design Principles**
- **Libraries:**
  - ML -
    - FluxML vs Pytorch
  - Optimization -
    - JUMP | Convex vs Pyomo | CVXPY
  - Auto Diff -
    - Zygote | ForwardDiff | ReverseDiff vs torch.autograd

Relation to Numerical Linear Algebra

- These libraries use numerical linear algebra for a variety of operations -


THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

Approach of Numerical Linear Algebra (NLA)

- These libraries use numerical linear algebra for a variety of operations -
- More specifically -
    - **ML**: matrix-matrix computations and matrix-vector computations
    - **Optimization**: gradients and linear algebra
    - **Automatic differentiation**: dual arithmetic

Sec 3: State of the Art (SOTA)

SOTA Designs in Julia

Python

- no type annotations
- single dispatch
- little code specialization
- extensions via inheritance

Julia

- Optional type annotations
- multiple dispatch
- metaprogramming
- efficient type inference
- aggressive code specialization
- JIT compilation

Experimental setup

- Comparison in state of the art auto diff packages.
- Basic ML problem and its extensions on different platforms and paradigms
- Optimization problem

➢ **Evaluation**

  - Performance
  - Productivity

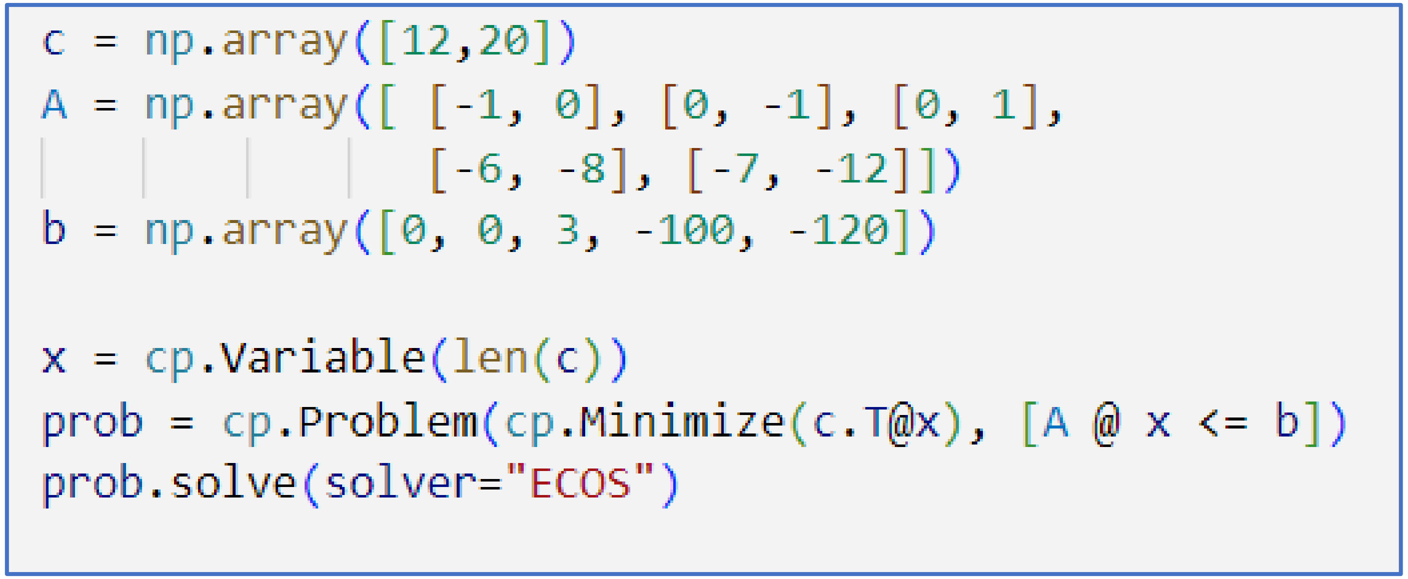
Optimization

- In this section we will be comparing:
  - **Julia**
    - JuMP
    - Convex.jl
  - **Python**
    - Pyomo
    - CVXPY

Optimization: Linear Program

Python - Pyomo

$$\begin{aligned} \min_{x,y} \quad & 12x + 20y \\ s.t. \quad & x \geq 0 \\ & 0 \leq y \leq 3 \\ & 6x + 8y \geq 100 \\ & 7x + 12y \geq 120 \end{aligned}$$

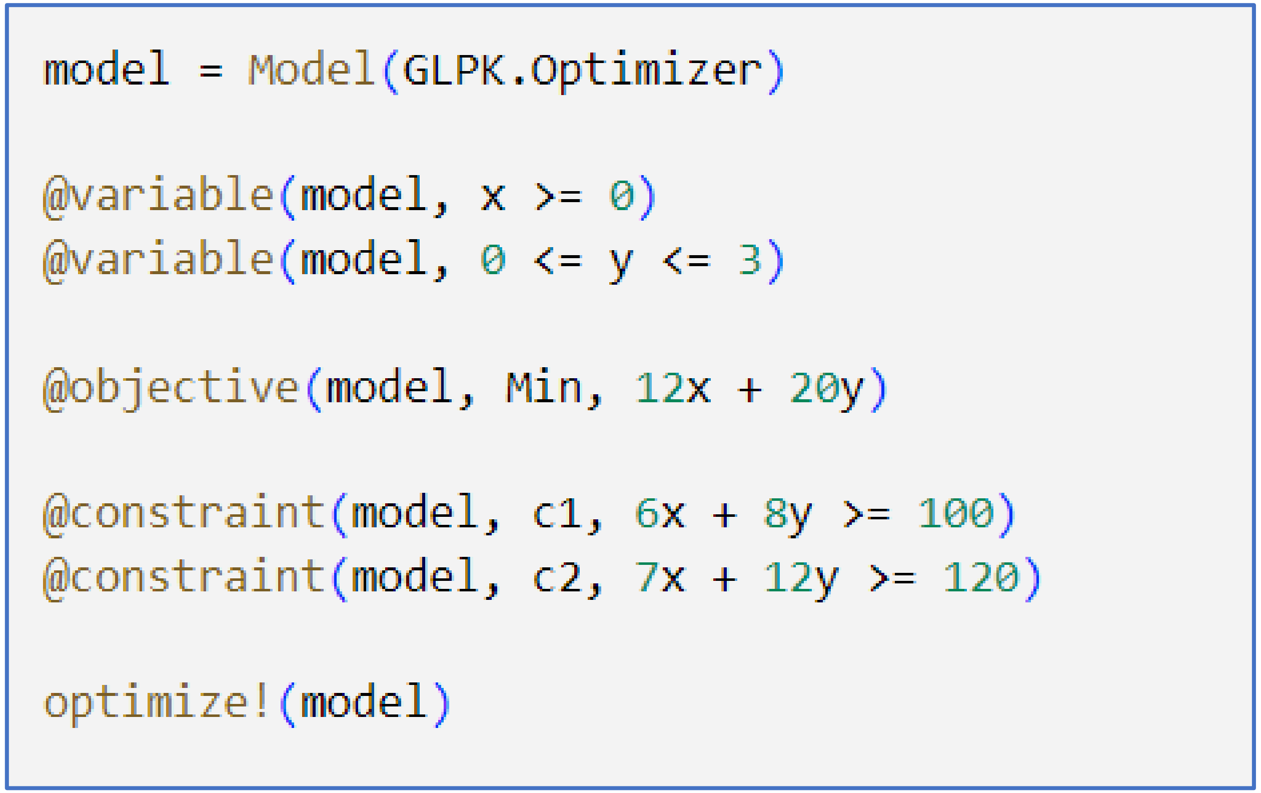


```
c = np.array([12,20])
A = np.array([ [-1, 0], [0, -1], [0, 1],
               [-6, -8], [-7, -12]])
b = np.array([0, 0, 3, -100, -120])

x = cp.Variable(len(c))
prob = cp.Problem(cp.Minimize(c.T@x), [A @ x <= b])
prob.solve(solver="ECOS")
```

Optimization: Linear Program

$$
\begin{array}{ll}
\min\limits_{x,y} & 12x + 20y \\
s.t. & x \geq 0 \\
 & 0 \leq y \leq 3 \\
 & 6x + 8y \geq 100 \\
 & 7x + 12y \geq 120
\end{array}
$$

Julia- JuMP

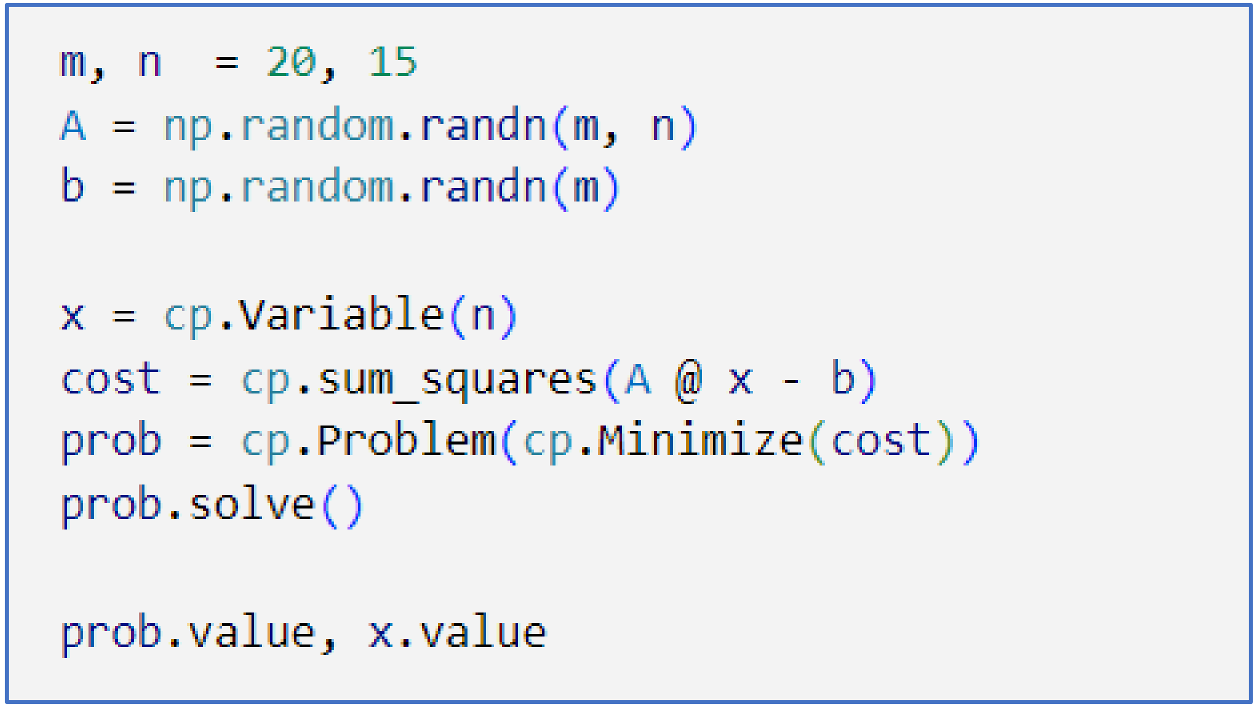


```
model = Model(GLPK.Optimizer)

@variable(model, x >= 0)
@variable(model, 0 <= y <= 3)

@objective(model, Min, 12x + 20y)

@constraint(model, c1, 6x + 8y >= 100)
@constraint(model, c2, 7x + 12y >= 120)

optimize!(model)
```

Optimization: Quadratic Program

Python - CVXPY

$$
\begin{aligned}
\min_{x} \quad & \|Ax - b\|^2 \\
\text{s.t.} \quad & x \geq 0
\end{aligned}
$$

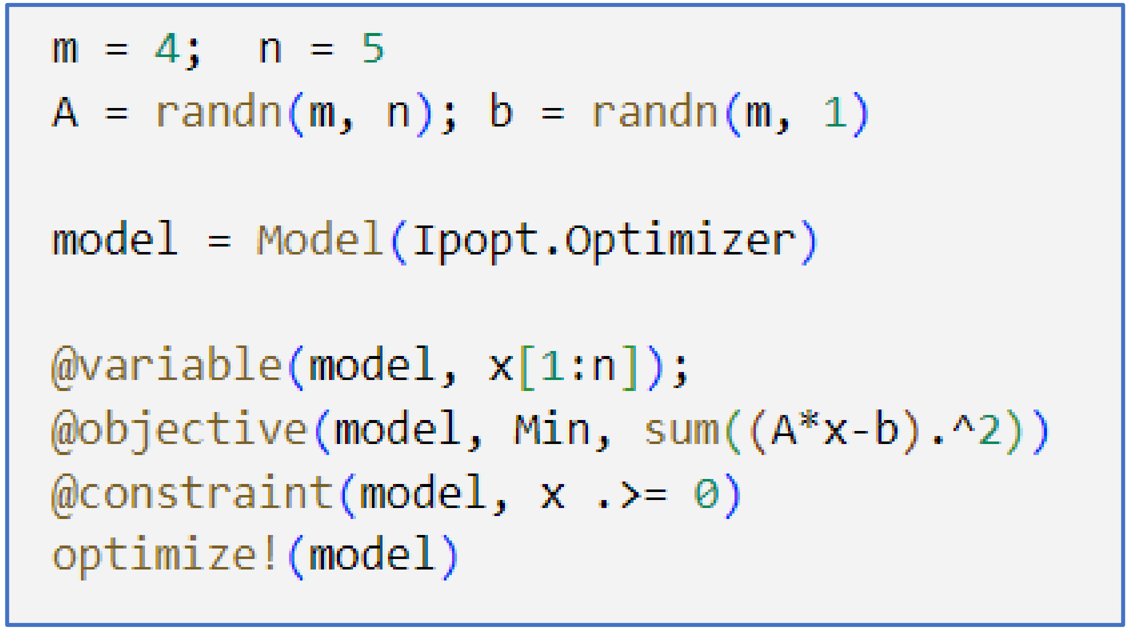


```
m, n  = 20, 15
A = np.random.randn(m, n)
b = np.random.randn(m)

x = cp.Variable(n)
cost = cp.sum_squares(A @ x - b)
prob = cp.Problem(cp.Minimize(cost))
prob.solve()

prob.value, x.value
```

Optimization: Quadratic Program

Julia - JuMP

$$\begin{aligned} \min_{x} \quad & \|Ax - b\|^2 \\ \text{s.t.} \quad & x \geq 0 \end{aligned}$$

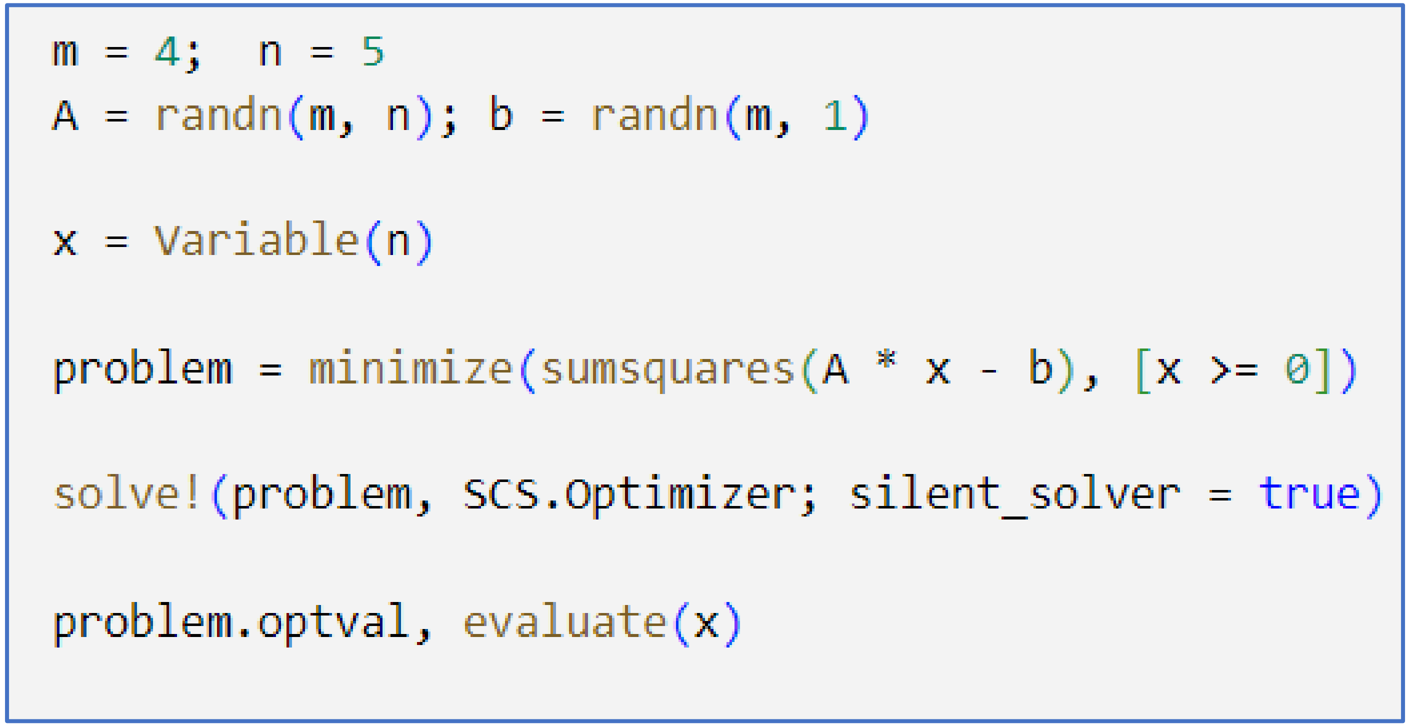


```
m = 4;  n = 5
A = randn(m, n); b = randn(m, 1)

model = Model(Ipopt.Optimizer)

@variable(model, x[1:n]);
@objective(model, Min, sum((A*x-b).^2))
@constraint(model, x .>= 0)
optimize!(model)
```

Optimization: Quadratic Program

Julia - Convex

$$\begin{aligned} \min_{x} \quad & \|Ax - b\|^2 \\ \text{s.t.} \quad & x \geq 0 \end{aligned}$$



```
m = 4;  n = 5
A = randn(m, n); b = randn(m, 1)

x = Variable(n)

problem = minimize(sumsquares(A * x - b), [x >= 0])

solve!(problem, SCS.Optimizer; silent_solver = true)

problem.optval, evaluate(x)
```

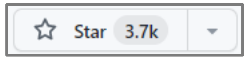
Comparison

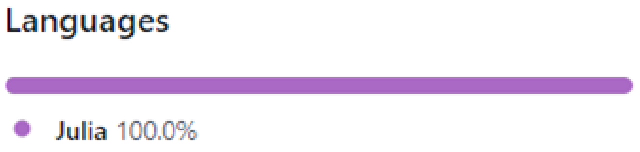
JuMP	**Convex**
👍 Allows nonlinear programming through an interface 👍 More flexibility 👍 Many supported solvers	👍 Converts to a standard conic form 👍 Requires convexity and DCP compliance 👍 Guarantees global optimality of the solution 👍 Supported solvers - Gurobi, Mosek, GLPK, ECOS, SCS
Pyomo	**CVXPY**
👍 Supported solvers - AMPL, PICO, CBC, CPLEX, IPOPT, and GLPK	👍 For convex optimization 👎 Slower and more complex

→ **JuMP is the best!**

Machine Learning

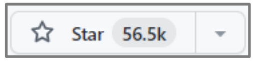
Flux

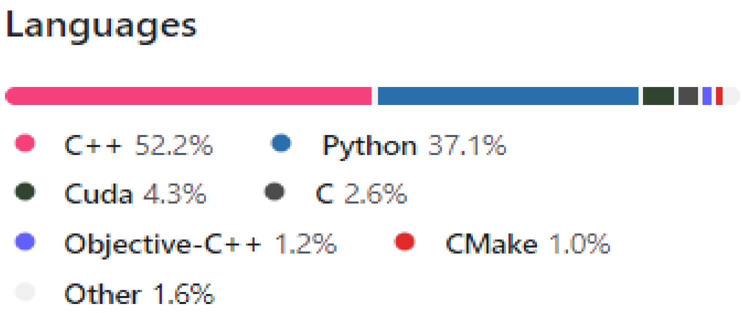



Languages

● Julia 100.0%

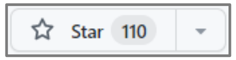
Pytorch

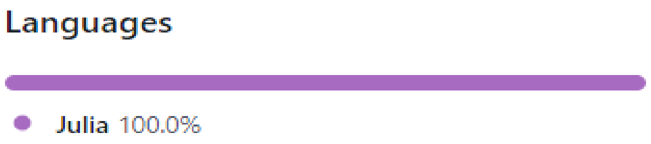



Languages

● C++ 52.2%
● Python 37.1%
● Cuda 4.3%
● C 2.6%
● Objective-C++ 1.2%
● CMake 1.0%
● Other 1.6%

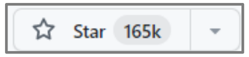
SimpleChains

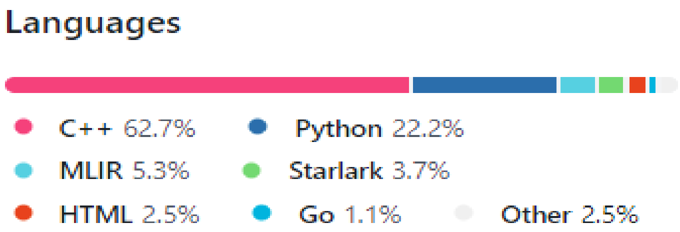



Languages

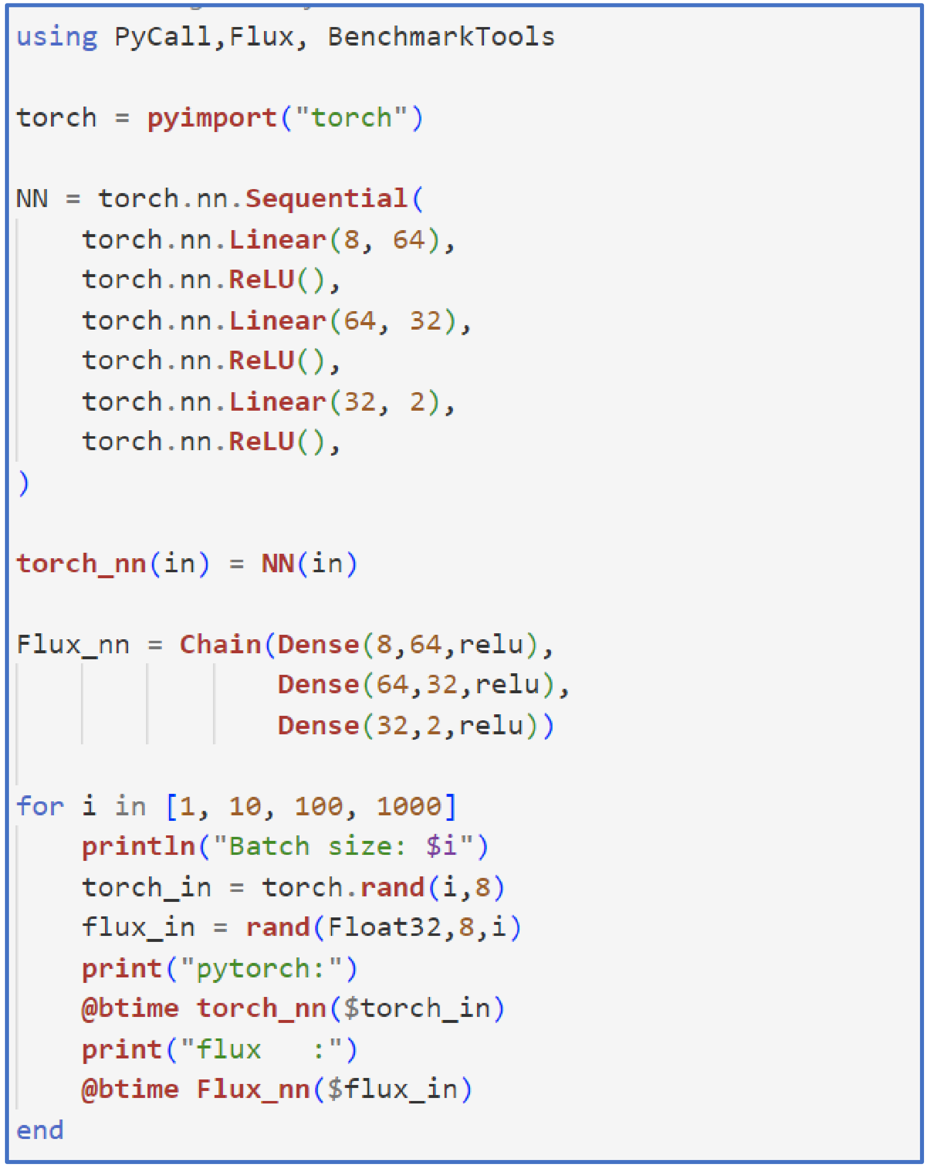
● Julia 100.0%

Tensorflow




Languages

● C++ 62.7%
● Python 22.2%
● MLIR 5.3%
● Starlark 3.7%
● HTML 2.5%
● Go 1.1%
● Other 2.5%



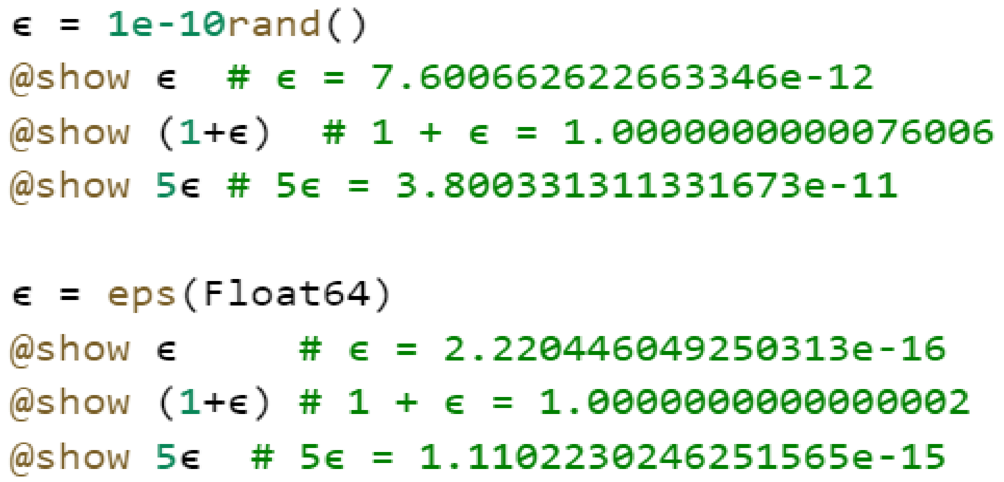
```
using PyCall,Flux, BenchmarkTools

torch = pyimport("torch")

NN = torch.nn.Sequential(
    torch.nn.Linear(8, 64),
    torch.nn.ReLU(),
    torch.nn.Linear(64, 32),
    torch.nn.ReLU(),
    torch.nn.Linear(32, 2),
    torch.nn.ReLU(),
)

torch_nn(in) = NN(in)

Flux_nn = Chain(Dense(8,64,relu),
                Dense(64,32,relu),
                Dense(32,2,relu))

for i in [1, 10, 100, 1000]
    println("Batch size: $i")
    torch_in = torch.rand(i,8)
    flux_in = rand(Float32,8,i)
    print("pytorch:")
    @btime torch_nn($torch_in)
    print("flux   :")
    @btime Flux_nn($flux_in)
end
```

Batch Size	Pytorch (µs)	Flux (µs)
1	109.100	2.433
10	110.300	4.486
100	198.000	16.200
1000	446.200	367.400

- model definition is very similar
- Can use Python from Julia
- Speed is comparable.

Automatic Differentiation

Floating point arithmetic



```
ϵ = 1e-10rand()
@show ϵ  # ϵ = 7.600662622663346e-12
@show (1+ϵ)  # 1 + ϵ = 1.0000000000076006
@show 5ϵ # 5ϵ = 3.800331311331673e-11

ϵ = eps(Float64)
@show ϵ     # ϵ = 2.220446049250313e-16
@show (1+ϵ) # 1 + ϵ = 1.0000000000000002
@show 5ϵ  # 5ϵ = 1.1102230246251565e-15
```

Finite Differences

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

Finite differencing uses addition of small perturbations ☹

Complex Differentiation

$$f(x+ih) = f(x) + f'(x)ih + o(h)$$

$$if'(x) = \frac{f(x+ih) - f(x)}{h} + o(h)$$

$$f'(x) = \frac{Im\big(f(x+ih)\big)}{h} + o(h)$$

- we are tracking perturbations in a different dimension.
- accuracy of real and imaginary parts would be much higher as there is no numerical cancellation for small values of h

Derivatives as measures of sensitivities

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

we will ignore higher order terms, formally we set $\epsilon^2 = 0$

$$f(x+\epsilon) \rightsquigarrow f(x) + \epsilon f'(x)$$
$$g(x+\epsilon) \rightsquigarrow g(x) + \epsilon g'(x)$$

Dual Numbers

- extend the idea of complex step differentiation beyond complex analytic functions.
- A dual number is a multidimensional number where the sensitivity of the function is propagated

along the dual portion.

$$f(x+\epsilon) \rightsquigarrow f(x) + \epsilon f'(x)$$
$$g(x+\epsilon) \rightsquigarrow g(x) + \epsilon g'(x)$$

$\boldsymbol{\epsilon}$ is dimensional signifier

Operations

$$f(x+\epsilon)+g(x+\epsilon)=[f(x)+g(x)]+\epsilon[f'(x)+g'(x)]$$

$$f(x+\epsilon)-g(x+\epsilon)=[f(x)-g(x)]+\epsilon[f'(x)-g'(x)]$$

$$f(x+\epsilon)\cdot g(x+\epsilon)=[f(x)\cdot g(x)]+\epsilon[f(x)\cdot g'(x)+g(x)\cdot f'(x)]$$

$$f(x+\epsilon)/g(x+\epsilon)=[f(x)\cdot g(x)]+\epsilon\left[\frac{g(x)\cdot f'(x)-f(x)\cdot g'(x)}{g(x)^2}\right]$$

Chain Rule

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

Higher Dimensions

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

$$\lim_{\epsilon \to 0} \frac{f(\mathbf{x} + \epsilon \mathbf{v}) - f(\mathbf{x})}{\epsilon} = [\nabla f(\mathbf{x})] \cdot \mathbf{v}$$

$x \in \mathbb{R}^n$ and $\nabla f(x)$ is the **gradient** of f at x

- different components of $\nabla f(x)$ - $\mathbf{v}$ as $e_i \quad i = 1..n$
- computing total derivative, would need multiple passes each with a different value of v.
- Can we compute the total derivative in one pass?
  - ✓ Yes!
  - ✓ We can track every partial in a different dimension.
  - ✓ We can think of these as ϵ_i's as perturbations in different directions, satisfying $\epsilon_i \epsilon_j = 0$

Higher Dimensions

$$d = d_0 + v_1\epsilon_1 + v_2\epsilon_2 + \cdots + v_m\epsilon_m$$

d_0 is the *primal* vector $[x_1, x_2 \cdots x_n]$
v_i are the vectors for the *dual* directions. If total derivative needs to be computed $m = n$

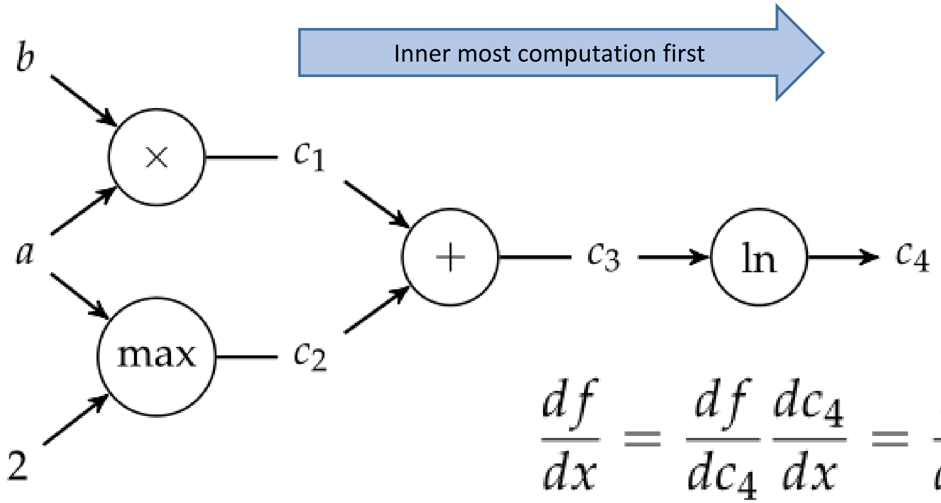
Single application of f to a multidimensional dual number calculates:

$$f(d) = f(d_0) + \nabla f(d_0)^\top v_1\epsilon_1 + \nabla f(d_0)^\top v_2\epsilon_2 + \cdots + \nabla f(d_0)^\top v_m\epsilon_m$$

Forward mode AD

- Forward accumulation will automatically differentiate a function using a single forward pass through the function’s computational graph.
- Iteratively expanding the chain rule of the inner operation

The computational graph : ln(ab + max(a, 2))



$$\frac{df}{dx} = \frac{df}{dc_4}\frac{dc_4}{dx} = \frac{df}{dc_4}\left(\frac{dc_4}{dc_3}\frac{dc_3}{dx}\right) = \frac{df}{dc_4}\left(\frac{dc_4}{dc_3}\left(\frac{dc_3}{dc_2}\frac{dc_2}{dx} + \frac{dc_3}{dc_1}\frac{dc_1}{dx}\right)\right)$$

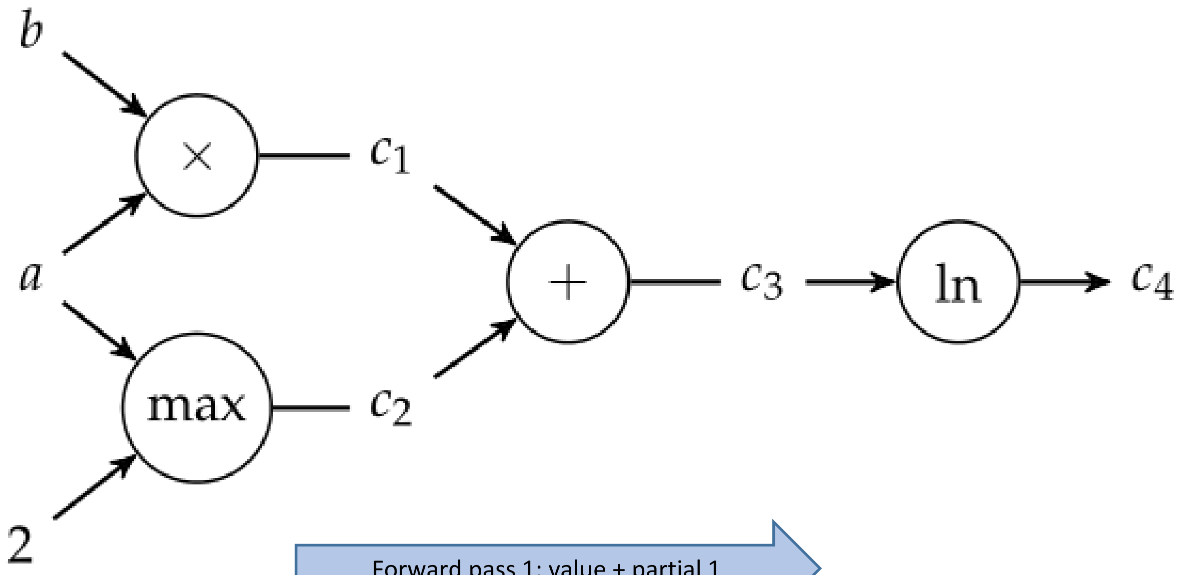
Forward mode AD

$b = 2$

$\dot{b} = 0$

$a = 3$

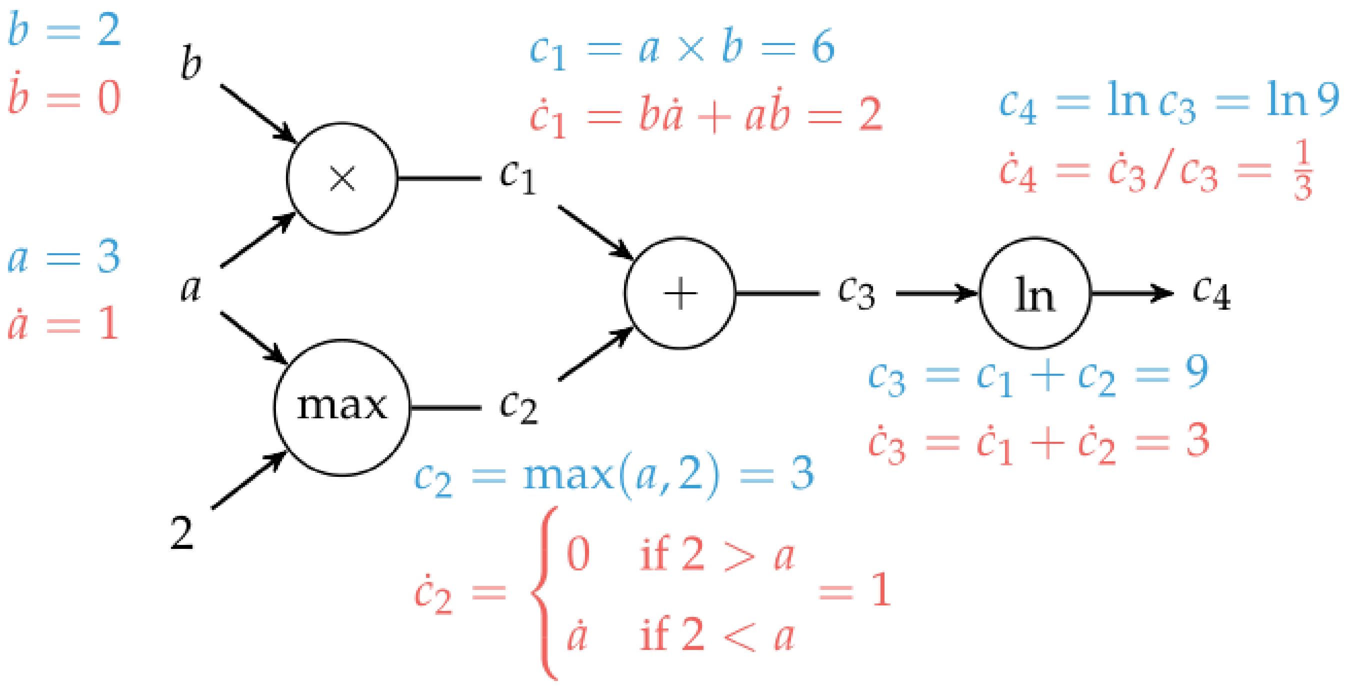
$\dot{a} = 1$



Forward pass 1: value + partial 1

Forward pass 2: value + partial 2

Forward Mode



$b = 2$

$\dot{b} = 0$

$a = 3$

$\dot{a} = 1$

$c_1 = a \times b = 6$

$\dot{c}_1 = b\dot{a} + a\dot{b} = 2$

$c_2 = \max(a, 2) = 3$

$$\dot{c}_2 = \begin{cases} 0 & \text{if } 2 > a \\ \dot{a} & \text{if } 2 < a \end{cases} = 1$$

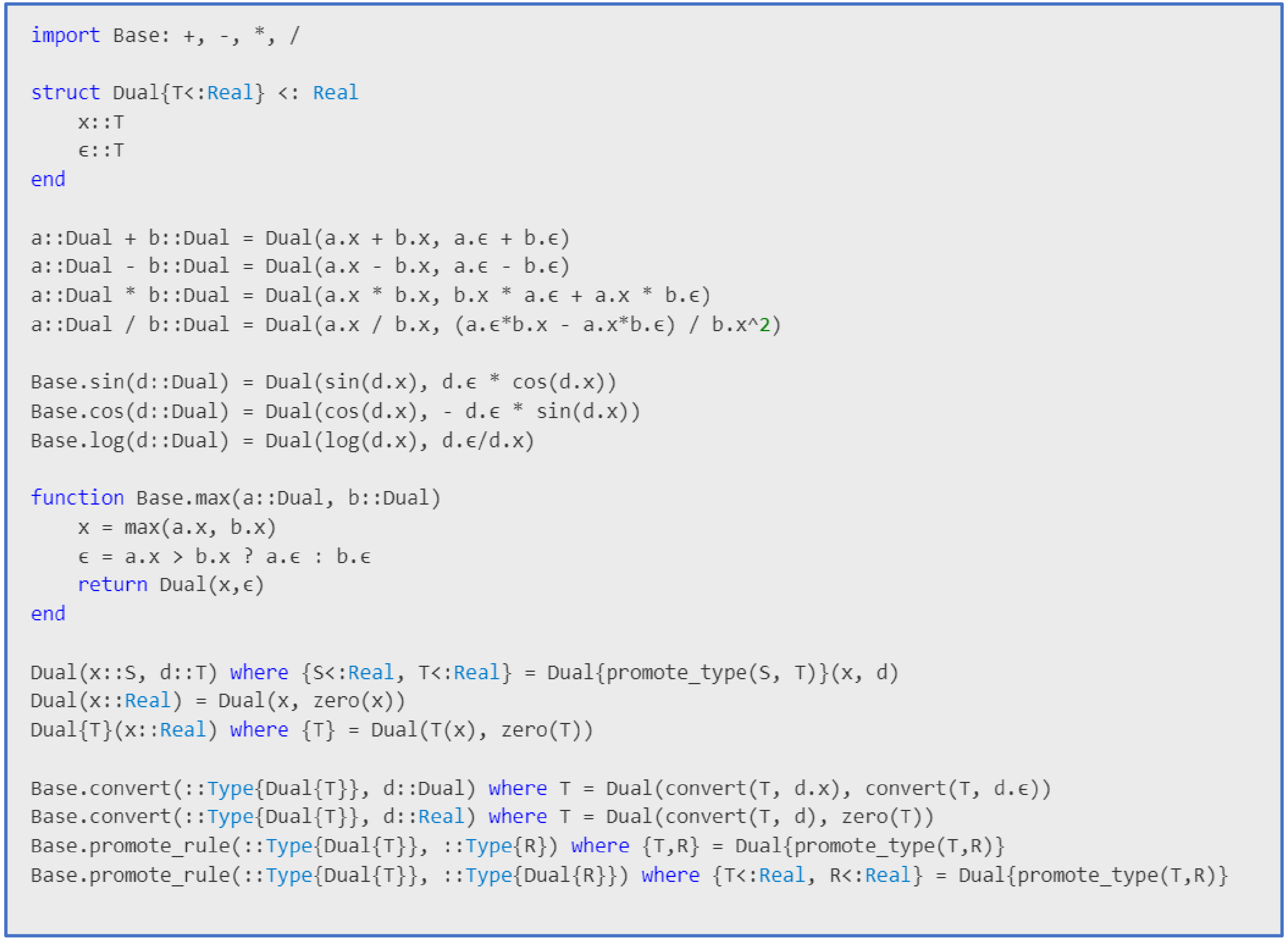
$c_3 = c_1 + c_2 = 9$

$\dot{c}_3 = \dot{c}_1 + \dot{c}_2 = 3$

$c_4 = \ln c_3 = \ln 9$

$\dot{c}_4 = \dot{c}_3 / c_3 = \frac{1}{3}$

Implementation



Dual number

```
import Base: +, -, *, /

struct Dual{T<:Real} <: Real
    x::T
    ϵ::T
end
```

Operations

```
a::Dual + b::Dual = Dual(a.x + b.x, a.ϵ + b.ϵ)
a::Dual - b::Dual = Dual(a.x - b.x, a.ϵ - b.ϵ)
a::Dual * b::Dual = Dual(a.x * b.x, b.x * a.ϵ + a.x * b.ϵ)
a::Dual / b::Dual = Dual(a.x / b.x, (a.ϵ*b.x - a.x*b.ϵ) / b.x^2)
```

Higher primitives

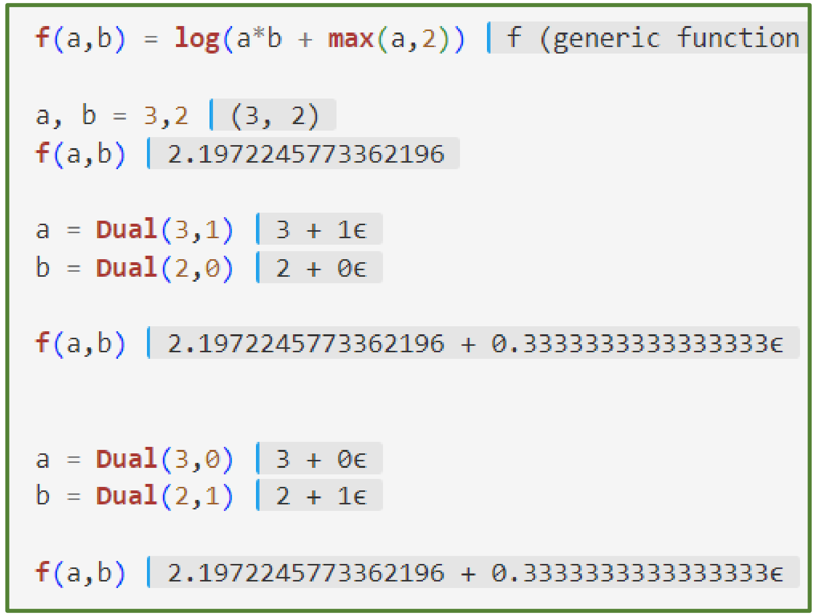
```
Base.sin(d::Dual) = Dual(sin(d.x), d.ϵ * cos(d.x))
Base.cos(d::Dual) = Dual(cos(d.x), - d.ϵ * sin(d.x))
Base.log(d::Dual) = Dual(log(d.x), d.ϵ/d.x)

function Base.max(a::Dual, b::Dual)
    x = max(a.x, b.x)
    ϵ = a.x > b.x ? a.ϵ : b.ϵ
    return Dual(x,ϵ)
end
```

Syntactic Sugar

```
Dual(x::S, d::T) where {S<:Real, T<:Real} = Dual{promote_type(S, T)}(x, d)
Dual(x::Real) = Dual(x, zero(x))
Dual{T}(x::Real) where {T} = Dual(T(x), zero(T))

Base.convert(::Type{Dual{T}}, d::Dual) where T = Dual(convert(T, d.x), convert(T, d.ϵ))
Base.convert(::Type{Dual{T}}, d::Real) where T = Dual(convert(T, d), zero(T))
Base.promote_rule(::Type{Dual{T}}, ::Type{R}) where {T,R} = Dual{promote_type(T,R)}
Base.promote_rule(::Type{Dual{T}}, ::Type{Dual{R}}) where {T<:Real, R<:Real} = Dual{promote_type(T,R)}
```

Results

Example



```
f(a,b) = log(a*b + max(a,2)) | f (generic function

a, b = 3,2 | (3, 2)
f(a,b) | 2.1972245773362196

a = Dual(3,1) | 3 + 1ϵ
b = Dual(2,0) | 2 + 0ϵ

f(a,b) | 2.1972245773362196 + 0.3333333333333333ϵ


a = Dual(3,0) | 3 + 0ϵ
b = Dual(2,1) | 2 + 1ϵ

f(a,b) | 2.1972245773362196 + 0.3333333333333333ϵ
```

$b = 2$

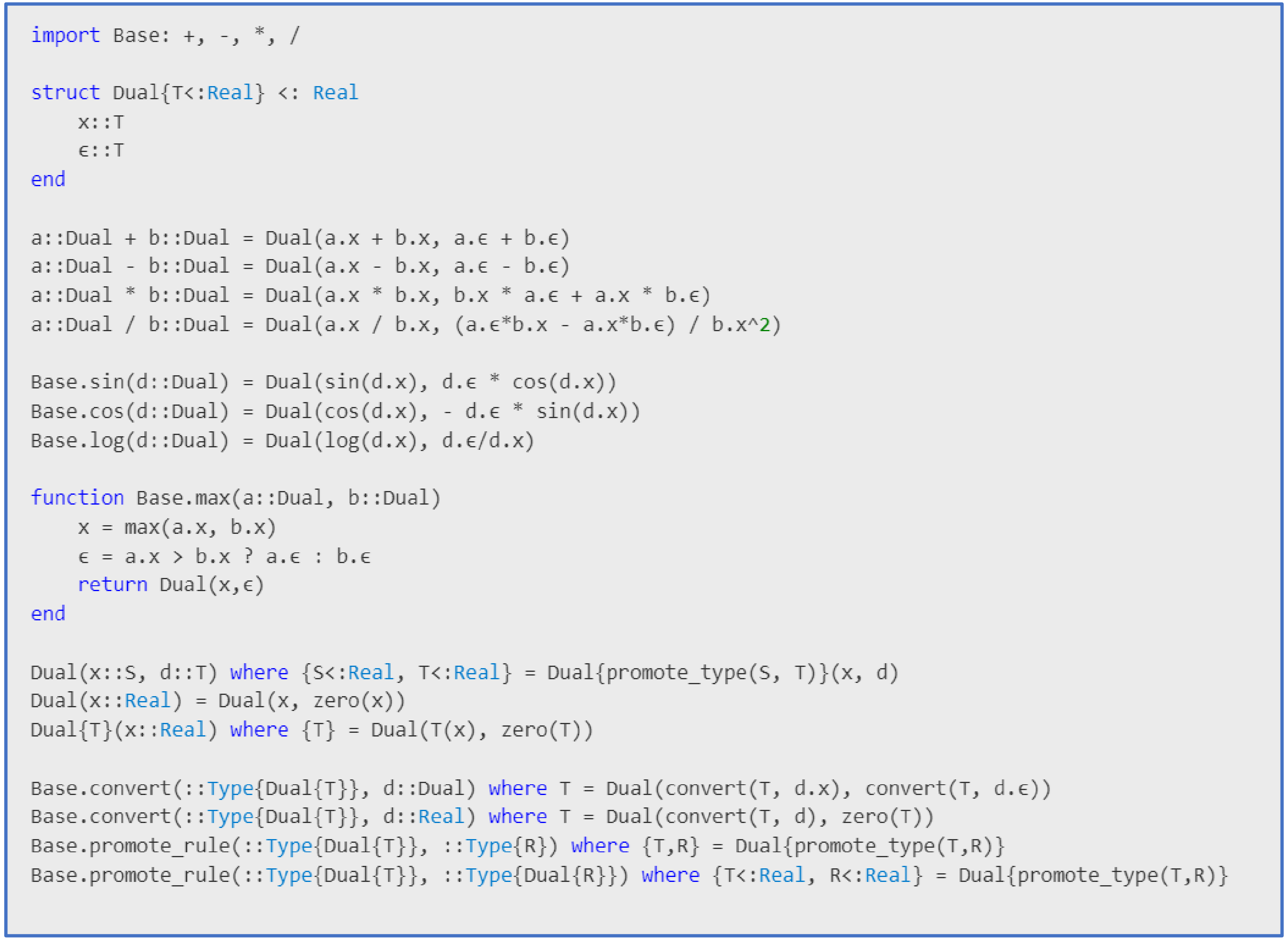
$\dot{b} = 0$

$c_4 = \ln c_3 = \ln 9$

$\dot{c}_4 = \dot{c}_3 / c_3 = \frac{1}{3}$

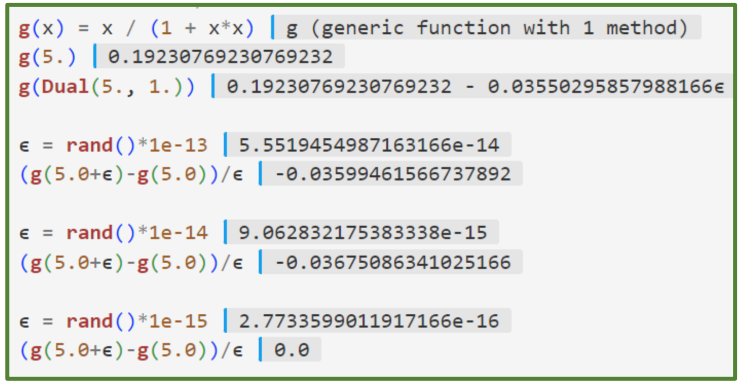
$a = 3$

$\dot{a} = 1$



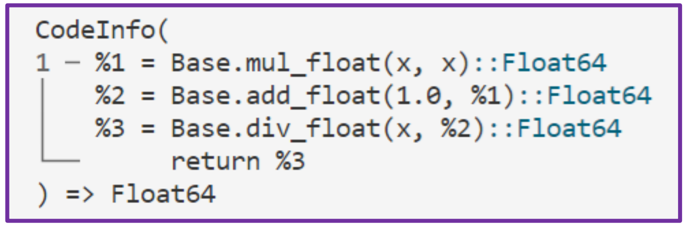
```
import Base: +, -, *, /

struct Dual{T<:Real} <: Real
    x::T
    ϵ::T
end

a::Dual + b::Dual = Dual(a.x + b.x, a.ϵ + b.ϵ)
a::Dual - b::Dual = Dual(a.x - b.x, a.ϵ - b.ϵ)
a::Dual * b::Dual = Dual(a.x * b.x, b.x * a.ϵ + a.x * b.ϵ)
a::Dual / b::Dual = Dual(a.x / b.x, (a.ϵ*b.x - a.x*b.ϵ) / b.x^2)

Base.sin(d::Dual) = Dual(sin(d.x), d.ϵ * cos(d.x))
Base.cos(d::Dual) = Dual(cos(d.x), - d.ϵ * sin(d.x))
Base.log(d::Dual) = Dual(log(d.x), d.ϵ/d.x)

function Base.max(a::Dual, b::Dual)
    x = max(a.x, b.x)
    ϵ = a.x > b.x ? a.ϵ : b.ϵ
    return Dual(x,ϵ)
end

Dual(x::S, d::T) where {S<:Real, T<:Real} = Dual{promote_type(S, T)}(x, d)
Dual(x::Real) = Dual(x, zero(x))
Dual{T}(x::Real) where {T} = Dual(T(x), zero(T))

Base.convert(::Type{Dual{T}}, d::Dual) where T = Dual(convert(T, d.x), convert(T, d.ϵ))
Base.convert(::Type{Dual{T}}, d::Real) where T = Dual(convert(T, d), zero(T))
Base.promote_rule(::Type{Dual{T}}, ::Type{R}) where {T,R} = Dual{promote_type(T,R)}
Base.promote_rule(::Type{Dual{T}}, ::Type{Dual{R}}) where {T<:Real, R<:Real} = Dual{promote_type(T,R)}
```

To summarize



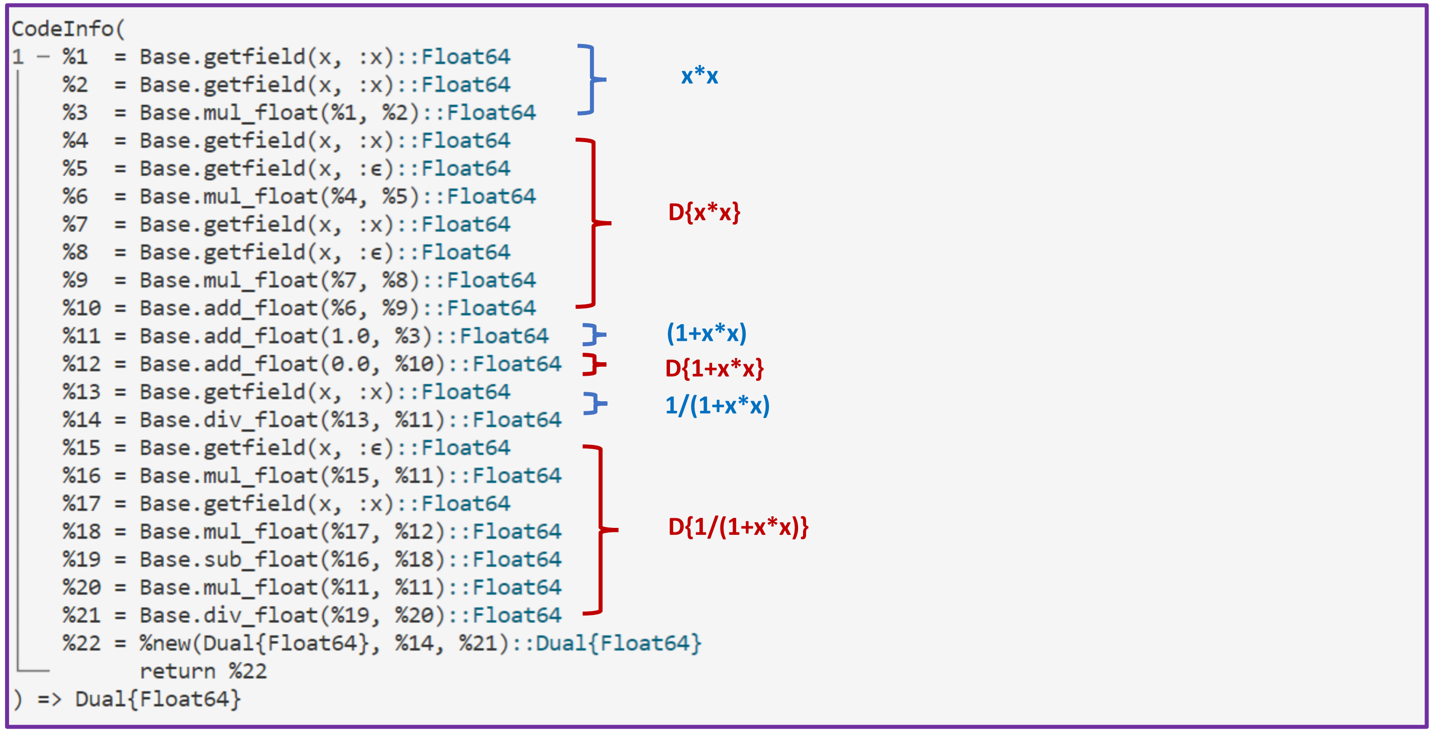
```
g(x) = x / (1 + x*x) | g (generic function with 1 method)
g(5.) | 0.19230769230769232
g(Dual(5., 1.)) | 0.19230769230769232 - 0.03550295857988166ϵ

ϵ = rand()*1e-13 | 5.5519454987163166e-14
(g(5.0+ϵ)-g(5.0))/ϵ | -0.03599461566737892

ϵ = rand()*1e-14 | 9.062832175383338e-15
(g(5.0+ϵ)-g(5.0))/ϵ | -0.03675086341025166

ϵ = rand()*1e-15 | 2.7733599011917166e-16
(g(5.0+ϵ)-g(5.0))/ϵ | 0.0
```

@code_lowered g(5.0)



```
CodeInfo(
1 ─ %1 = Base.mul_float(x, x)::Float64
│   %2 = Base.add_float(1.0, %1)::Float64
│   %3 = Base.div_float(x, %2)::Float64
└──      return %3
) => Float64
```

@code_lowered g(Dual(5., 1.))

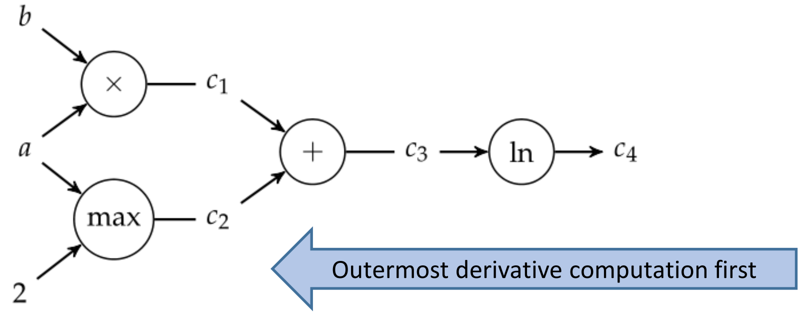


```
CodeInfo(
1 ─ %1  = Base.getfield(x, :x)::Float64
│   %2  = Base.getfield(x, :x)::Float64
│   %3  = Base.mul_float(%1, %2)::Float64
│   %4  = Base.getfield(x, :x)::Float64
│   %5  = Base.getfield(x, :ϵ)::Float64
│   %6  = Base.mul_float(%4, %5)::Float64
│   %7  = Base.getfield(x, :x)::Float64
│   %8  = Base.getfield(x, :ϵ)::Float64
│   %9  = Base.mul_float(%7, %8)::Float64
│   %10 = Base.add_float(%6, %9)::Float64
│   %11 = Base.add_float(1.0, %3)::Float64
│   %12 = Base.add_float(0.0, %10)::Float64
│   %13 = Base.getfield(x, :x)::Float64
│   %14 = Base.div_float(%13, %11)::Float64
│   %15 = Base.getfield(x, :ϵ)::Float64
│   %16 = Base.mul_float(%15, %11)::Float64
│   %17 = Base.getfield(x, :x)::Float64
│   %18 = Base.mul_float(%17, %12)::Float64
│   %19 = Base.sub_float(%16, %18)::Float64
│   %20 = Base.mul_float(%11, %11)::Float64
│   %21 = Base.div_float(%19, %20)::Float64
│   %22 = %new(Dual{Float64}, %14, %21)::Dual{Float64}
└──       return %22
) => Dual{Float64}
```

Reverse Mode

- Forward accumulation requires n passes in order to compute an n-dimensional gradient.

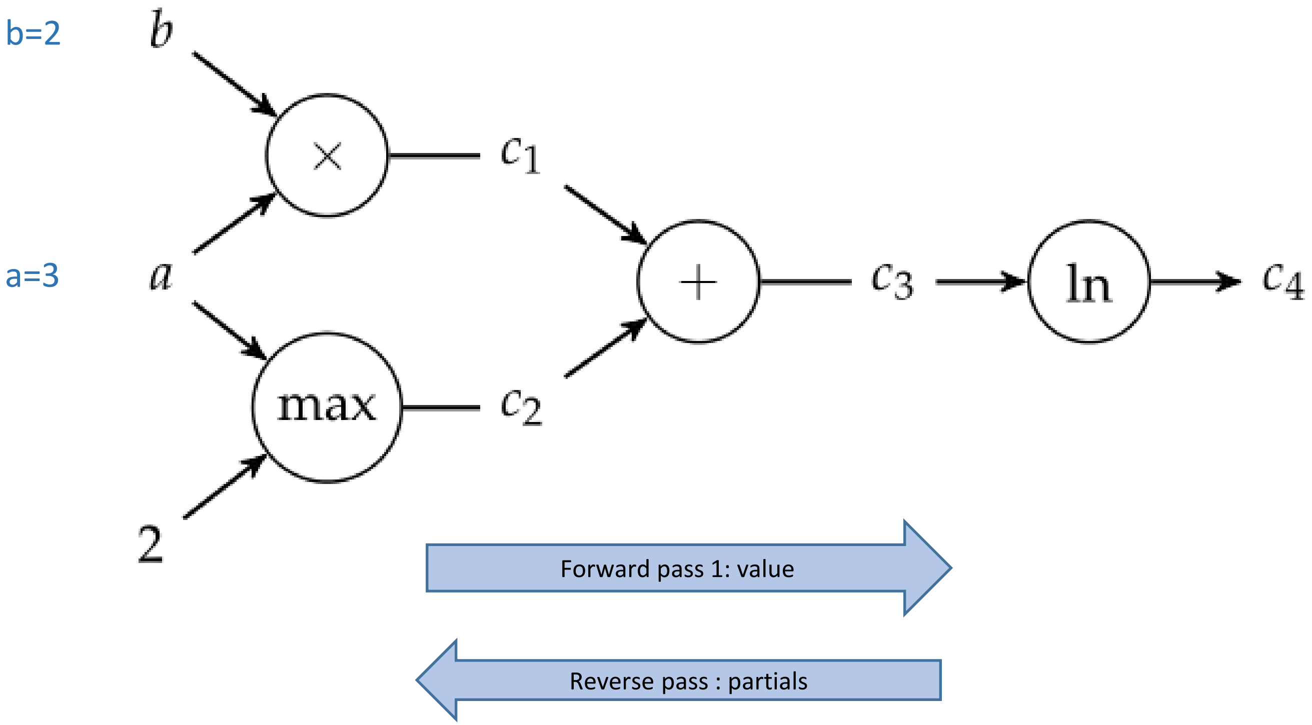
- Reverse accumulation requires only a single run to compute a complete gradient but requires two passes through the graph:
  - forward pass - computes intermediate values
  - backward pass - computes the gradient



Outermost derivative computation first

$$\frac{df}{dx} = \frac{df}{dc_4}\frac{dc_4}{dx} = \left(\frac{df}{dc_3}\frac{dc_3}{dc_4}\right)\frac{dc_4}{dx} = \left(\left(\frac{df}{dc_2}\frac{dc_2}{dc_3} + \frac{df}{dc_1}\frac{dc_1}{dc_3}\right)\frac{dc_3}{dc_4}\right)\frac{dc_4}{dx}$$

Reverse mode AD



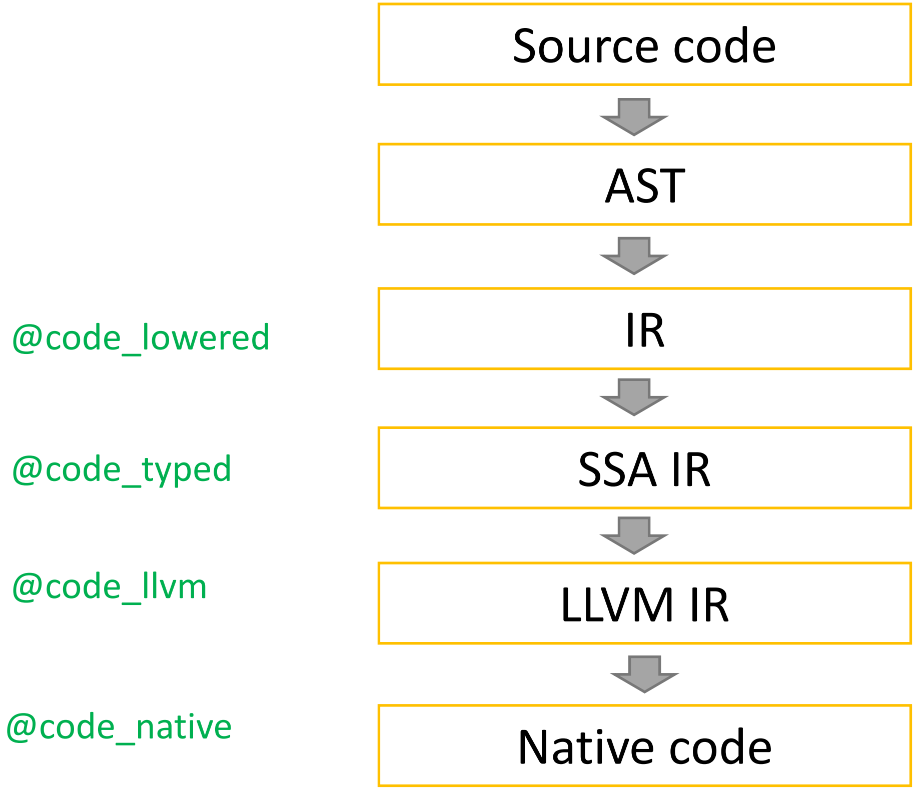
b=2

a=3

Forward pass 1: value

Reverse pass : partials

Compilation pipeline



Source code

AST

@code_lowered IR

@code_typed SSA IR

@code_llvm LLVM IR

@code_native Native code

SOTA Python

Tensorflow	Pytorch - autograd	Jax
• Graph building system – users define variables in specific language separate from host language • Static sublanguage is represented into IR known as XLA that performed optimization • AD would be performed on static graph.	• Tape based • Generates the code to autodiff for every forward pass by storing the operation that it sees in forward pass, and then differentiates that set of operations in reverse • Building of tape is done by operator overloading	• Hybrid – tries mixing advantages of both sides • Jax uses non-standard interpretation to build copy of the full code in its own IR • AD is performed on IR, finally lowering it to TensorFlow's XLA for optimizations
• Can see the entire computational graph - hence can do many optimizations. • Efficient because of XLA (Accelerated Linear Algebra)	• AD does not see the dynamic control flow. • AD does not have to handle dynamic control flow. • Implementation is easier.	• Efficiency of Tensorflow but in a form that looks more natural. • Can be seen as a more natural graph builder for Tensorflow
• Not flexible and inconvenient for users.	• AD is *per value,* so cannot do a lot of optimizations. • Generally slow especially for small kernels.	• Not fully dynamic – use Jax primitives e.g. Jax.while • Functions must be pure • Jax requires functional style with pure functions rather than OOP of python.

Next

- AD approaches in TensorFlow/Pytorch/Jax aim to eliminate dynamic constructs before the AD
  - smaller surface of language support required

- Is it possible to keep the full dynamism of the host language in the AD system?
  - It is possible, but it is hard.
  - AD having to deal with full dynamic nature of entire programming language is difficult
  - Python is too dynamic
  - So people working on these solutions flocked to languages with clear syntax that is easy for compilers to optimize, i.e. Julia and Swift.
  - This is what a lot of the Julia AD tools have focused on with source code transformations.
  - However, since source code is "for humans", it can be a rather difficult level to algorithmically work on.
    - Instead these tools work on lowered IR, where these lowered representations remove a lot complicated syntax to give a much smaller support surface

SOTA Julia

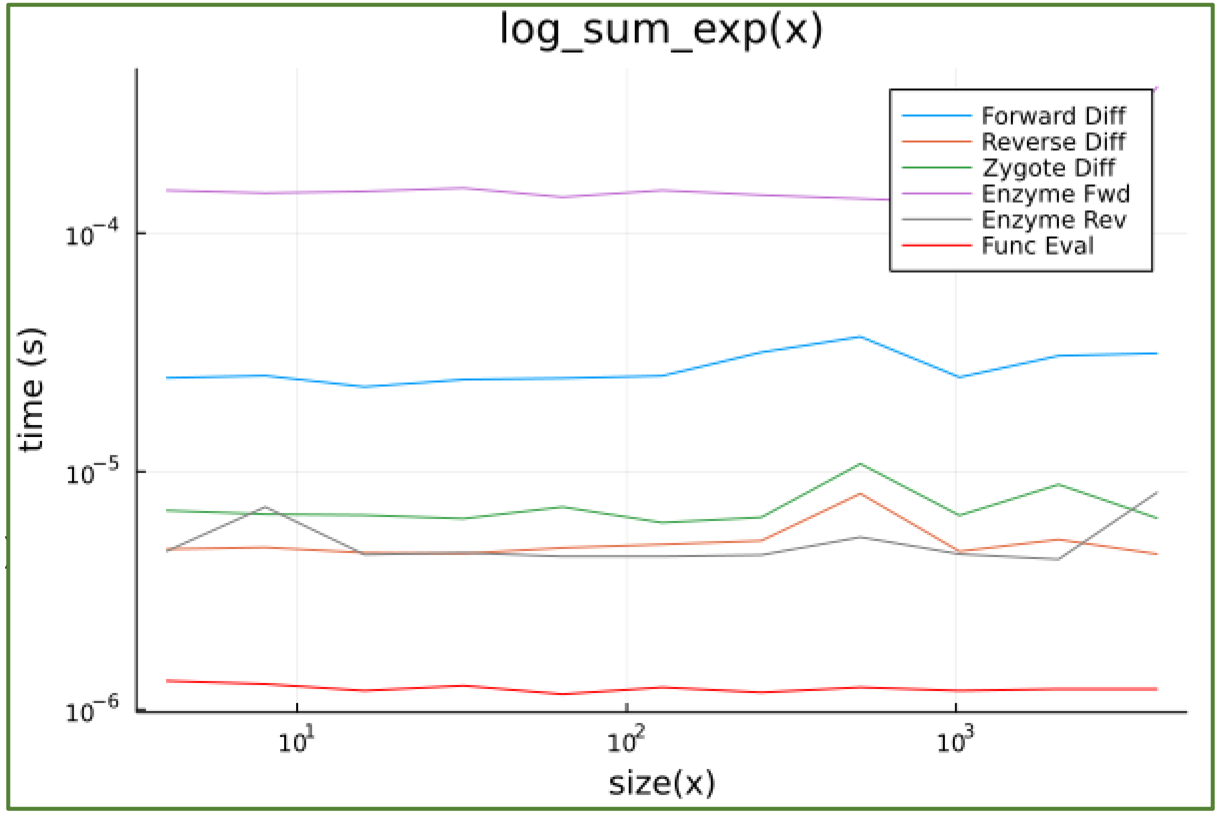
Zygote	Enzyme	Diffractor (in Dev)
• Source-to-source transformation. • Current AD system for Flux. • Operates on SSA IR	• Source-to-source transformation. • Operates on LLVM’s IR	• Source-to-source transformation. • Next-gen AD system • In development • Operates on Julia’s typed IR
• Very flexible	• Its ability to perform AD on optimized code allows Enzyme be very efficient.	• Ultra high performance for both scalar and array code • Efficient higher order derivatives • Reasonable compile times • High flexibility (like Zygote) • Support for forward/reverse/mixed modes
• Acts on high level IR – before compiler optimizations • Requires you do AD on unoptimized code only delete most of the work later	• Can act after compiler optimizations. However, does not have higher level information which could be useful for AD transformations.	• Additional tooling needs to be built – Escape Analysis

Benchmarks

- f(x) = log(sum(exp.(x)))

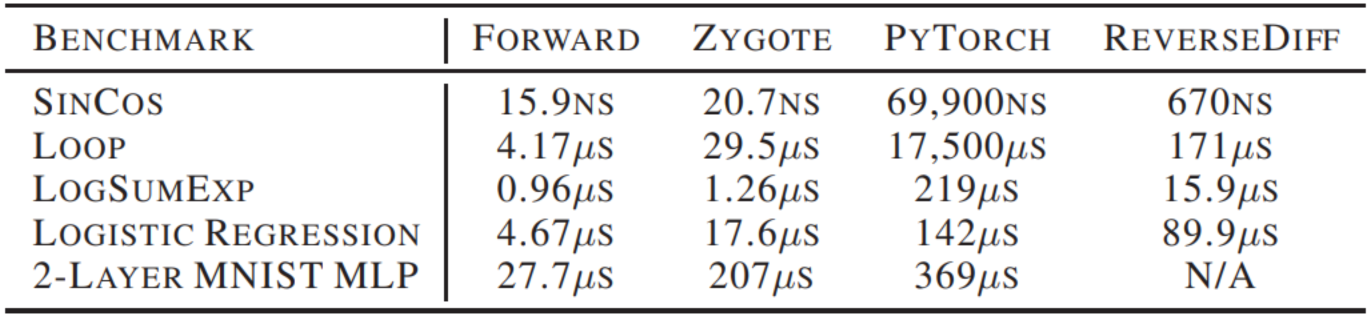
  x is a vector

- Observations
  - Function evaluation takes least time
  - Forward mode is slower than reverse mode
  - Enzyme reverse is much efficient
  - Zygote (Backend of flux performs sufficiently well)
  - Pytorch did as well as Zygote.
  - Jax did as well as enzyme.



Benchmarks

Some other existing benchmarks.



Benchmark	Forward	Zygote	PyTorch	ReverseDiff
SinCos	15.9ns	20.7ns	69,900ns	670ns
Loop	4.17μs	29.5μs	17,500μs	171μs
LogSumExp	0.96μs	1.26μs	219μs	15.9μs
Logistic Regression	4.67μs	17.6μs	142μs	89.9μs
2-Layer MNIST MLP	27.7μs	207μs	369μs	N/A

Sec 4: Concluding remarks

Conclusions

- Julia
  - Julia is a new language with many promising libraries in domains of Automatic Differentiation, Machine Learning, Optimization.
  - Some of these libraries are relatively new.

- Python
  - Very popular in the domain of machine learning. ML libraries are very mature.
  - Relies on other language for speed

References

Bezanson, J., Edelman, A., Karpinski, S. and Shah, V.B., 2017. Julia: A fresh approach to numerical computing. *SIAM review*, *59*(1), pp.65-98.

Seeger, M., Hetzel, A., Dai, Z., Meissner, E. and Lawrence, N.D., 2017. Auto-differentiating linear algebra. *arXiv preprint arXiv:1710.08717*.

Schäfer, F., Tarek, M., White, L. and Rackauckas, C., 2021. Abstract Differentiation. jl: Backend-Agnostic Differentiable Programming in Julia. *arXiv preprint arXiv:2109.12449*.

Innes, M., Edelman, A., Fischer, K., Rackauckas, C., Saba, E., Shah, V.B. and Tebbutt, W., 2019. A differentiable programming system to bridge machine learning and scientific computing. *arXiv preprint arXiv:1907.07587*.

Innes, M., Karpinski, S., Shah, V., Barber, D., Saito Stenetorp, P.L.E.P.S., Besard, T., Bradbury, J., Churavy, V., Danisch, S., Edelman, A. and Malmaud, J., 2018, February. On machine learning and programming languages. Association for Computing Machinery (ACM).

Darve, E. and Wootters, M., 2021. *Numerical Linear Algebra with Julia* (Vol. 172). SIAM.

https://userpages.umbc.edu/~dfrey1/ench445/is-it-worth-the-effort-to-learn-Julia.html

https://juliatrend.github.io/

https://medium.com/android-news/magic-lies-here-statically-typed-vs-dynamically-typed-languages-d151c7f95e2b

https://towardsdatascience.com/r-vs-python-vs-julia-90456a2bcbab