A powerful, high level language with high performance.

Pythonic, mathematical syntax that looks like notation.

Performance consistently within 2x of tuned C code.

Most of Julia is written in Julia!

```julia
function mandel(z)
    c = z
    maxiter = 80
    for n = 1:maxiter
        if abs(z) > 2
            return n-1
        end
        z = z^2 + c
    end
    return maxiter
end
```
Scientific Computing

JUMP

DSGE.jl

DifferentialEquations.jl

Cori

Federal Reserve Bank of New York

Incorporated May 18, 1914
Machine Learning

Turing is a universal probabilistic programming language with an intuitive modelling interface, composable probabilistic inference, and computational scalability.
Machine Learning

High-level and flexible (Python)

High overhead, focus on tensor operations and manual vectorisation

Relatively simple programs (network architectures)

Mutation support considered advanced/unusual.

Scientific Computing

Low-level and manual (Fortran)

Low overhead, focus on scalar operations

Regularly run over millions of lines of code.

Research on auto-vectorisation, shared memory parallelism, checkpointing etc.
Languages that are both *high level* and *high performance*.

the key:
Tapenade

Ruthless pragmatism and scalability. Output can be highly optimised using existing optimising compilers.

λ the Ultimate Backpropagator

Elegant recursive formalism, including nested AD (closure), convenience (callee-derives) and bags of expressive power.
pkg> add Zygote

Resolving package versions...
Updating `~/.julia/environments/v1.2/Project.toml`
[e88e6eb3] + Zygote v0.3.2
Updating `~/.julia/environments/v1.2/Manifest.toml`
[1a297f60] + FillArrays v0.6.3
[7869d1d1] + IRTools v0.2.2
[e88e6eb3] + Zygote v0.3.2

julia> using Zygote

julia> function pow(x, n)
    r = 1
    while n > 0
        n -= 1
        r *= x
    end
    return r
end

pow (generic function with 1 method)

julia> pow(5, 3)
125

julia> gradient(x -> pow(x, 3), 5)
(75,)

function pow(x, n)
    r = 1
    while n > 0
        n -= 1
        r *= x
    end
    return r
end

pow(5, 3) == 125
gradient(pow, 5, 3) == (75, 0)
\[
\begin{align*}
    y &= f(x_1, x_2, ...) \\
    y, B &= J(f, x_1, x_2, ...) \\
    \bar{x}_1, \bar{x}_2, ... &= B(\bar{y})
\end{align*}
\]
```
function foo(x)
    a = bar(x)
    b = baz(a)
    return b
end

function J(::typeof(foo), x)
    a, da = J(bar, x)
    b, db = J(baz, a)
    return b, function(b~)
        ā = db(b~)
        x~ = da(ā)
        return x~
    end
end
```
```
fs = Dict("sin" => sin, "cos" => cos, "tan" => tan);

f(x) = fs[readline()])(x)
```

```
julia> f(1)
sin
0.8414709848078965

julia> gradient(f, 1)
sin
(0.5403023058681398,)
```

Documentation: https://docs.julialang.org

Type "?" for help, "]??" for Pkg help.

Version 1.2.0-rc1.2 (2019-05-31) release-1.2/3fcb168ceb (fork: 74 commits, 81 days)
\[ J(::\text{typeof}(\sin), \ x) = \sin(x), \ \dot{y} \rightarrow \dot{y} \cdot \cos(x) \]

\[ \text{@adjoint } \sin(x) = \sin(x), \ \dot{y} \rightarrow \dot{y} \cdot \cos(x) \]

Core compiler pass is \(~200\) lines of code

All semantics added via custom adjoints – mutation, data structures, checkpointing, etc.
nestlevel() = 0

@adjoint nestlevel() = nestlevel()+1, _ -> nothing

julia> function f(x)
    println(nestlevel(), " levels of nesting")
    return x
end

julia> f(1);
0 levels of nesting

julia> grad(f, 1);
1 levels of nesting

julia> grad(x -> x*grad(f, x), 1);
2 levels of nesting
@adjoint hook(f, x) = x, Δ -> (f(Δ),)
hook(-, x)  # reverse the gradient of x

@adjoint checkpoint(f, x...) =
  f(x...), Δ -> J(f, x...)[2](Δ)

@adjoint function forwarddiff(f, x)
  y, J = forward_jacobian(f, x)
  y, Δ -> (J’Δ,)
end
julia> hook(f, x) = x
hook (generic function with 1 method)

julia> @adjoint hook(f, x) = x, Δ → (nothing, f(Δ),)

julia> gradient(2, 3) do a, b
    a*b
end
(3, 2)

julia> gradient(2, 3) do a, b
    hook(−, a) * b
end
(−3, 2)

julia> gradient(2, 3) do a, b
    hook(ā ->@show(ā), a) * b
end
ā = 3
(3, 2)
Differentiation á la Carte

- Mixed-mode AD (forward, reverse, Taylor series, ...)
- Forward-over-reverse (Hessians)
- Cross-language AD
- Support for Complex and other number types
- Easy custom gradients
- Checkpointing
- Gradient hooks
- Custom types (colours!)
- Hardware backends: CPU, CUDA, TPU, ...
- Deeply nested AD (WIP)
Data Structures & Mutation
```julia
using Colors

julia> a, b = RGB(1, 0, 0), RGB(0, 1, 0)
   (RGB{N0f8}(1.0,0.0,0.0), RGB{N0f8}(0.0,1.0,0.0))

julia> a.r^2
1.0N0f8

julia> gradient(c -> c.r^2, a)
   ((r = 2.0f0, g = nothing, b = nothing),)

julia> colordiff(a, b)
   86.60823557376344

julia> gradient(a -> colordiff(a, b), a)
   ((r = 0.4590887719632896, g = -9.598786801605689, b = 14.181383399012862),)
```
Deep learning in 5 lines.

dense(W, b, σ = identity) =
  x → σ(W * x + b)

chain(f ...) = foldl(., reverse(f))

mlp = chain(
  dense(randn(5, 10), randn(5), tanh),
  dense(randn(2, 5), randn(2)))

x = rand(10)

mlp(x) → Float64[2]
  0.646...
  2.51...

\[ \dot{\mathbf{m}} = \text{gradient}(\mathbf{mlp}) \text{ do } \mathbf{m} \\
\quad \text{sum}(\mathbf{m}(\mathbf{x})) \]

end (f = (W = [-0.9909137325976834 0.11388709497399903 ... -0.7210152885786678 0.9901037595800784])

\[ \eta \cdot \dot{\mathbf{m}} \quad \# \text{ Gradient descent} \]
julia> vars = Dict(:r => 0, :n => 0)
Dict{Symbol,Int64} with 2 entries:
  :n => 0
  :r => 0

julia> function pow(x, n)
   vars[:r] = 1
   vars[:n] = n
   while vars[:n] > 0
      vars[:n] -= 1
      vars[:r] *= x
   end
end
pow (generic function with 1 method)

julia> pow(5, 3); vars[:r]
125

julia> gradient(x -> (pow(x, 3); vars[:r]), 5)
(75,)

julia> vars[:r]
125
Some Bonus Features
@grad function (a::Real * b::Real)
    c = a*b
    function back(Δ)
        0//0
    end
    c, back
end  { > _forward

function pow(x, n)
    r = one(x)
    while n > 0
        r *= x
        n -= 1
    end
    return r
end  { > pow

gradient(pow, 2, 3)  ArgumentError: invalid rational: zero(Int64)//zero(Int64)
in top-level scope at base/none
in gradient at Zygote/src/compiler/interface.jl:34
in at Zygote/src/compiler/interface.jl:28
in at Zygote/src/compiler/interface2.jl
in pow at test.jl:17
in at Zygote/src/lib/lib.jl:33
in at test.jl:9
in // at base/rational.jl:13
using Zygote

function f(x)
    for i = 1:5
        x = sin(cos(x))
    end
    return x
end

function loop(x, n)
    r = x/x
    for i = 1:n
        r *= f(x)
    end
    return sin(cos(r))
end

gradient(loop, 2, 3)

Zygote.@profile loop(2, 3)

function logsumexp(x::Array{Float64,1})
```python
import torch.nn.functional as F

def foo(W, b, x):
    return F.sigmoid(W @ x + b)

W = randn(2, 5)
b = randn(2)
x = rand(5)

foo(W, b, x)  # Float64[2]
    0.207...
    0.499...

dW, db = gradient(W, b) do W, b
    sum(((foo(W, b, x) - [0, 1]).^2)
end  # Array[Float64,2]; Float64[2])
```
@adjoint function pycall(f, x...; kw...)
    x = map(py, x)
    y = pycall(f, x...; kw...)
    y.detach().numpy(), function (ŷ)
        y.backward(gradient = py(ŷ))
        (nothing, map(x -> x.grad.numpy(), x)...)  
    end
end
end
Future Challenges

● Mutation of values is hard
● Need adjoints to cover the entire standard library
● Compiler improvements
  ○ More functional-style optimisations
  ○ Better heuristics for AD-generated code
● Fast code vs. dynamic semantics
● Differentiating Julia’s concurrency and parallelism constructs
● Reducing overheads: currently ~50ns per operation
  ○ Great compared to ML frameworks but far from optimal
Zygote

Unifying Machine Learning and Scientific Computing

mike@juliacomputing.com