present

Table of Contents

The four methods to differentiate a function

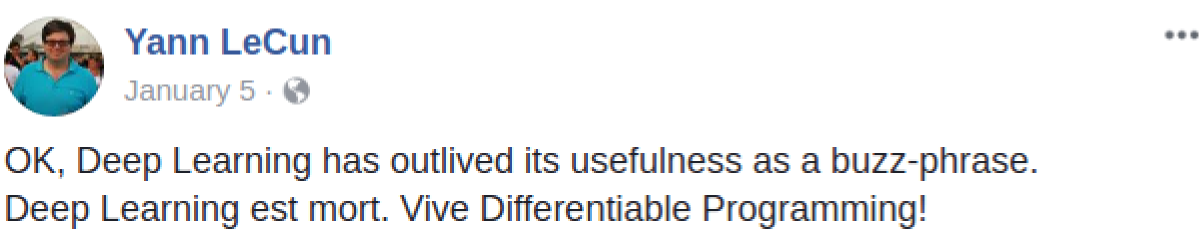
“谁要你教，不是草头底下一个来回的回字么？”

孔乙己显出极高兴的样子，将两个指头的长指甲敲着柜台，点头说，“对呀对呀！……回字有四样写法，你知道么？”我愈不耐烦了，努着嘴走远。

孔乙己刚用指甲蘸了酒，想在柜上写字，见我毫不热心，便又叹一口气，显出极惋惜的样子。

The history of autodiff

- 1964 ~ Robert Edwin Wengert, A simple automatic derivative evaluation program. ◀ first forward mode AD
- 1970 ~ Seppo Linnainmaa, Taylor expansion of the accumulated rounding error. ◀ first backward mode AD
- 1986 ~ Rumelhart, D. E., Hinton, G. E., and Williams, R. J., Learning representations by back-propagating errors.◀ bring AD to machine learning people.
- 1992 ~ Andreas Griewank, Achieving logarithmic growth of temporal and spatial complexity in reverse automatic differentiation. ◀ also known as optimal checkpointing.
- 2000s ~ The boom of tensor based AD frameworks for machine learning.
- 2018 ~ Re-inventing AD as differential programming (wiki.)



- 2020 ~ Moses, William and Churavy, Valentin, Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients ◀ AD on LLVM.

Differentiating the Bessel function

$$J_\nu(z) = \sum_{n=0}^{\infty} \frac{(z/2)^\nu}{\Gamma(k+1)\Gamma(k+\nu+1)}(-z^2/4)^n$$

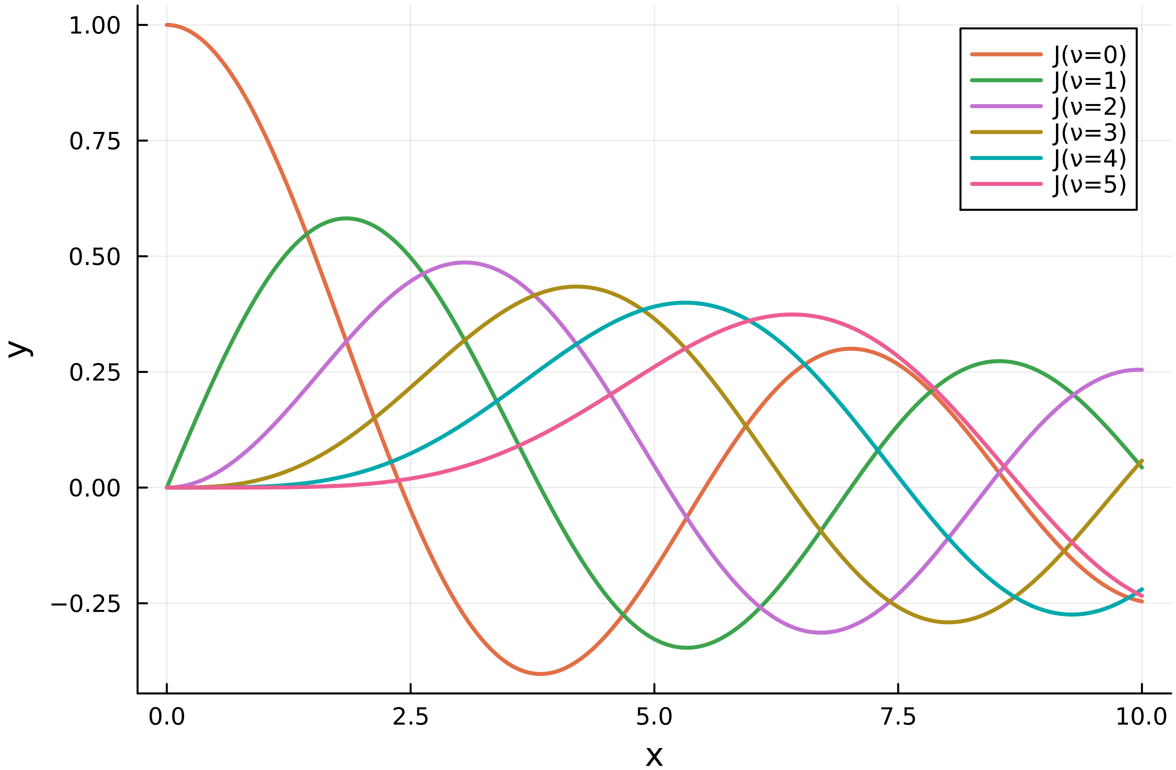
Poorman's Besssel function

```
poor_besselj (generic function with 1 method)
```

```julia
function poor_besselj(ν, z::T; atol=eps(T)) where T
    k = 0
    s = (z/2)^ν / factorial(ν)
    out = s
    while abs(s) > atol
        k += 1
        s *= (-1) / k / (k+ν) * (z/2)^2
        out += s
    end
    out
end
```

In each step, the state transfer can be described as $(k_i, s_i, out_i) \rightarrow (k_{i+1}, s_{k+1}, out_{i+1})$.

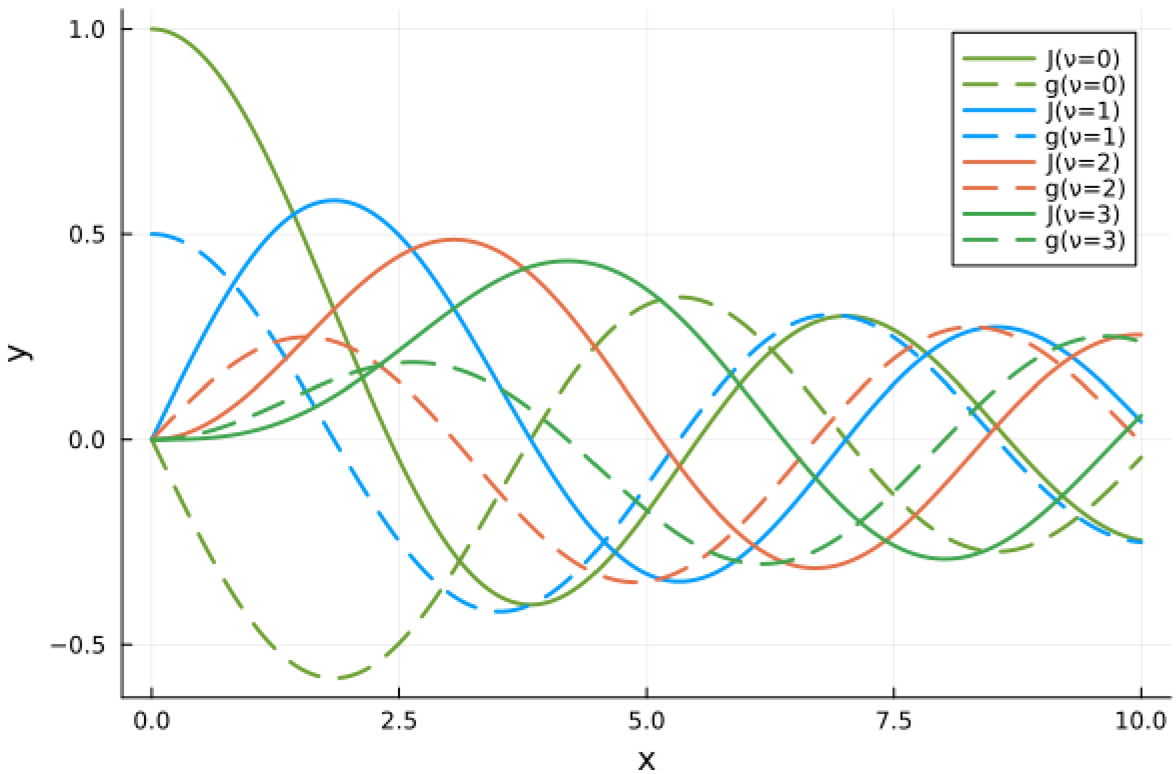
```julia
using Plots
```




```julia
let
    x = 0.0:0.01:10
    plt = plot([], []; label="", xlabel="x", ylabel="y")
    for i=0:5
        yi = poor_besselj.(i, x)
        plot!(plt, x, yi; label="J(ν=$i)", lw=2)
    end
    plt
end
```

Manual

```julia
@bind select_gradient_method Select(["Manual", "Forward", "Backward", "FiniteDiff"])
```




```
let
    x = 0.0:0.01:10
    plt = plot([], []; label="", xlabel="x", ylabel="y")
    for i=0:3
        yi = poor_besselj.(i, x)
        if select_gradient_method == "Forward"
            gi = [autodiff(Forward, poor_besselj, i, Enzyme.Duplicated(xi, 1.0))[1] for xi in x]
        elseif select_gradient_method == "Manual"
            gi = ((i == 0 ? -poor_besselj.(i+1, x) : poor_besselj.(i-1, x)) - poor_besselj.(i+1, x)) ./ 2
        elseif select_gradient_method == "Backward"
            gi = [autodiff(Reverse, poor_besselj, i, Enzyme.Active(xi))[1] for xi in x]
        elseif select_gradient_method == "FiniteDiff"
            gi = [autodiff(Reverse, poor_besselj, i, Enzyme.Active(xi))[1] for xi in x]
        end
        plot!(plt, x, yi; label="J(ν=$i)", lw=2, color=i)
        plot!(plt, x, gi; label="g(ν=$i)", lw=2, color=i, ls=:dash)
    end
    plt
end
```

Finite difference

First order forward Difference

$$\frac{\partial f}{\partial x} \approx \frac{f(x+\Delta) - f(x)}{\Delta}$$

First order backward Difference

$$\frac{\partial f}{\partial x} \approx \frac{f(x) - f(x-\Delta)}{\Delta}$$

First order central Difference

$$\frac{\partial f}{\partial x} \approx \frac{f(x+\Delta) - f(x-\Delta)}{2\Delta}$$

Table of finite difference coefficient: wiki page.

Example: central finite difference to the 4th order

1. Check the table

-2	-1	0	1	2
1/12	-2/3	0	2/3	-1/12

2. Apply the fomula

$$\frac{\partial f}{\partial x} \approx \frac{f(x-2\Delta) - 8f(x-\Delta) + 8f(x+\Delta) - f(x+2\Delta)}{12\Delta}$$

$$\begin{pmatrix} f(x-2\Delta) \\ f(x-\Delta) \\ f(x) \\ f(x+\Delta) \\ f(x+2\Delta) \end{pmatrix} \approx \begin{pmatrix} 1 & (-2)^1 & (-2)^2 & (-2)^3 & (-2)^4 \\ 1 & (-1)^1 & (-1)^2 & (-1)^3 & (-1)^4 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & (1)^1 & (1)^2 & (1)^3 & (1)^4 \\ 1 & (2)^1 & (2)^2 & (2)^3 & (2)^4 \end{pmatrix} \begin{pmatrix} f(x) \\ f'(x)\Delta \\ f''(x)\Delta^2/2 \\ f'''(x)\Delta^3/6 \\ f''''(x)\Delta^4/24 \end{pmatrix}$$

Let the finite difference coefficients be $\vec{\alpha}^T = (\alpha_{-2}, \alpha_{-1}, \alpha_0, \alpha_1, \alpha_2)$, we want $\alpha^T \vec{f} = f'(x)\Delta + O(\Delta^5)$, where $\vec{f} = A\vec{g}$ is the vector on the left side. $\vec{\alpha}$ can be solved by $A^T\backslash(0,1,0,0,0)^T$

```
[0.0833333, -0.666667, -2.37905e-16, 0.666667, -0.0833333]
```

```julia
let
    b = [0.0, 1, 0, 0, 0]
    A = [i^j for i=-2:2, j=0:4]
    A' \ b
end
```

```
5×5 Matrix{Int64}:
 1  -2  4  -8  16
 1  -1  1  -1   1
 1   0  0   0   0
 1   1  1   1   1
 1   2  4   8  16
```

```julia
[i^j for i=-2:2, j=0:4]
```

```julia
using FiniteDifferences
```

```
0.11985236384013791
```

```julia
central_fdm(5, 1)(x->poor_besselj(2, x), 0.5)
```

```julia
using BenchmarkTools
```

```
BenchmarkTools.Trial: 10000 samples with 8 evaluations.
 Range (min … max):  3.604 μs …  1.081 ms  ┊ GC (min … max): 0.00% … 98.88%
 Time  (median):     3.827 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   4.187 μs ± 14.999 μs  ┊ GC (mean ± σ):  5.03% ±  1.40%
```

```
  3.6 μs       Histogram: log(frequency) by time     6.42 μs <

 Memory estimate: 2.59 KiB, allocs estimate: 36.
```

```julia
@benchmark central_fdm(5, 1)(y->poor_besselj(2, y), x) setup=(x=0.5)
```

Forward mode automatic differentiation

Forward mode AD attaches a infitesimal number ϵ to a variable, when applying a function f, it does the following transformation

$$f(x+g\epsilon) = f(x) + f'(x)g\epsilon + \mathcal{O}(\epsilon^2)$$

The higher order infinitesimal is ignored.

In the program, we can define a *dual number* with two fields, just like a complex number

```
f((x, g)) = (f(x), f'(x)*g)
```

```julia
using ForwardDiff
```

```
res = Dual{Nothing}(0.7071067811865475,1.4142135623730951)
```

```julia
res = sin(ForwardDiff.Dual(π/4, 2.0))
```

```
true
```

```julia
res === ForwardDiff.Dual(sin(π/4), cos(π/4)*2.0)
```

We can apply this transformation consecutively, it reflects the chain rule.

$$\frac{\partial \vec{y}_{i+1}}{\partial x} = \underbrace{\boxed{\frac{\partial \vec{y}_{i+1}}{\partial \vec{y}_i}}}_{\text{local Jacobian}} \frac{\partial \vec{y}_i}{\partial x}$$

Example: Computing two gradients $\frac{\partial z \sin x}{\partial x}$ and $\frac{\partial \sin^2 x}{\partial x}$ at one sweep

```julia
using Enzyme
```

```
0.11985236384014333
```

```julia
autodiff(Forward, poor_besselj, 2, Duplicated(0.5, 1.0))[1]
```

```
BenchmarkTools.Trial: 10000 samples with 993 evaluations.
 Range (min … max):  35.001 ns … 55.673 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     35.212 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   35.565 ns ±  1.585 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
 35 ns         Histogram: log(frequency) by time        43 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```julia
@benchmark autodiff(Forward, poor_besselj, 2, Duplicated(x, 1.0))[1] setup=(x=0.5)
```

What if we want to compute gradients for multiple inputs?

The computing time grows **linearly** as the number of variables that we want to differentiate. But does not grow significantly with the number of outputs.

Reverse mode automatic differentiation

On the other side, the back-propagation can differentiate **many inputs** with respect to a **single output** efficiently

$$\frac{\partial \mathcal{L}}{\partial \vec{y}_i} = \frac{\partial \mathcal{L}}{\partial \vec{y}_{i+1}} \underbrace{\boxed{\frac{\partial \vec{y}_{i+1}}{\partial \vec{y}_i}}}_{\text{local jacobian?}}$$

```
0.11985236384014332
```

```julia
autodiff(Reverse, poor_besselj, 2, Enzyme.Active(0.5))[1]
```

```
BenchmarkTools.Trial: 10000 samples with 918 evaluations.
 Range (min … max):  115.186 ns … 221.536 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     119.291 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   120.377 ns ±   4.129 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
 115 ns          Histogram: frequency by time          132 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```julia
@benchmark autodiff(Reverse, poor_besselj, 2, Enzyme.Active(x))[1] setup=(x=0.5)
```

How to visit local Jacobians in the reversed order?

Caching intermediate results in a stack!

Rule based autodiff

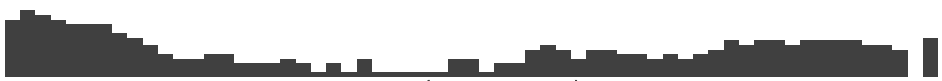
The backward rule of the Bessel function is

$$
\begin{aligned}
J'_\nu(z) &= \frac{J_{\nu-1}(z) - J_{\nu+1}(z)}{2}\\
J'_0(z) &= -J_1(z)
\end{aligned}
$$

```
0.11985236384014333
```

```julia
0.5 * (poor_besselj(1, 0.5) - poor_besselj(3, 0.5))
```

```
BenchmarkTools.Trial: 10000 samples with 993 evaluations.
 Range (min … max):  33.998 ns … 91.596 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     34.315 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   34.770 ns ±  1.747 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```



```
  34 ns         Histogram: log(frequency) by time      42.6 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```julia
@benchmark 0.5 * (poor_besselj(1, x) - poor_besselj(3, x)) setup=(x=0.5)
```

Deriving the backward rule of matrix multiplication

Please check blog

Rule based or not?

	rule based	differential programming
meaning	defining backward rules manully for functions on tensors	defining backward rules on a limited set of basic scalar operations, and generate gradient code using source code transformation
pros and cons	1. Good tensor performance 2. Mature machine learning ecosystem 3. Need to define backward rules manually	1. Reasonalbe scalar performance 2. hard to utilize BLAS
packages	Jax PyTorch	Tapenade Adept Enzyme

Obtaining Hessian

Hessian is the Jacobian of the gradient. We can use **forward over backward**.

Optimal checkpointing, towards solving the memory wall problem

Game: Pass the ball

In each step, if you have the ball, you pick one of the following actions

1. raise your hand, and pass the ball to the next,
2. pass the ball to the next without raising your hand,
3. only if you are the last one in the queue, you can left the queue and pass the ball to those raising hands.

Otherwise, you may

1. put down your hand, or
2. do nothing.

Goal: We require the number of raised hands being at most m at the same time, please empty the queue while minimizing the number of ball passings.

The connection to checkpointing

- A person: a computing state s_k,
- The queue: a linear program $s_1, s_2, \ldots, s_n$,
- Passing ball: program running forward $s_k \rightarrow s_{k+1}$,
- Left queue: the gradient g_k being computed,
- Rasing hand: create a checkpoint in the main memory,
- put down the hand: deallocate a checkpoint.

Homeworks

1. Given the binomial function $\eta(\tau, \delta) = \frac{(\tau+\delta)!}{\tau!\delta!}$, show that the following statement is true.

$$\eta(\tau, \delta) = \sum_{k=0}^{\delta} \eta(\tau - 1, k)$$

2. Given the following program to compute the l_2-norm of a vector $x \in R^n$.

```
function poorman_norm(x::Vector{<:Real})
    nm2 = zero(real(eltype(x)))
    for i=1:length(x)
        nm2 += abs2(x[i])
    end
    ret = sqrt(nm2)
    return ret
end
```

In the program, the `abs2` and `sqrt` functions can be treated as primitive functions, which means they should not be further decomposed as more elementary functions.

Tasks

1. Rewrite the program (on paper or with code) to implement the forward mode autodiff, where you can use the notation $\dot{y}_i \equiv \frac{\partial y}{\partial x_i}$ to denote a derivative.
2. Rewrite the program (on paper or with code) to implement the reverse mode autodiff, where you can use the notation $\overline{y} \equiv \frac{\partial \mathcal{L}}{\partial y}$ to denote an adjoint, $y \rightarrow T$ to denote pushing a variable to the global stack, and $y \leftarrow T$ to denote poping a variable from the global stack. In your submission, both the forward pass and backward pass should be included.
3. Estimate how many intermediate states is cached in your reverse mode autodiff program?

Reference

- Griewank A, Walther A. Evaluating derivatives: principles and techniques of algorithmic differentiation[M]. Society for industrial and applied mathematics, 2008.