

Juliaと歩く 自動微分

Julia Tokyo #11

@abap34
2023/02/03



各種リンク

.....

ソースコード・スライドのソース:

<https://github.com/abap34/juliatokyo11>

このスライド(アニメーション付きのhtml版):

<https://www.abap34.com/slides/juliatokyo11/slide.html>

自己紹介

東京工業大学
情報理工学院 情報工学系 B2

趣味

-  機械学習
-  個人開発
-  野球をする/みる



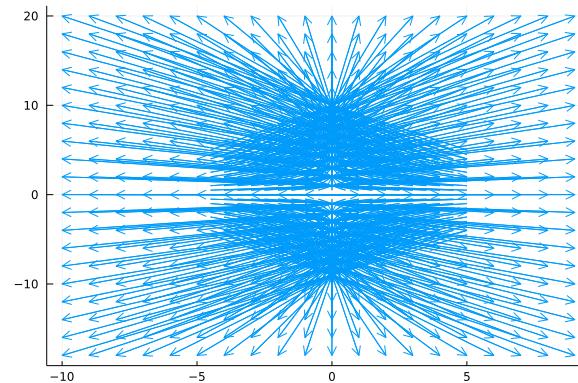
 [@abap34](https://github.com/abap34)  [@abap34](https://twitter.com/abap34)
 <https://abap34.com>

✓ Juliaのこんなところが気に入っています！

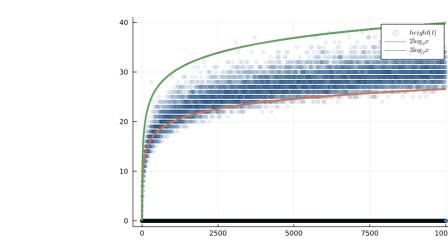
1. 綺麗な可視化・ベンチマークライブラリ
 - i. Plotまわり, @code_... マクロ, BenchmarkTools.jl たち
2. パッケージ管理ツール
 - i. 言語同梱、仮想環境すぐ作れる、パッケージ化簡単
3. すぐ書ける すぐ動く
 - i. Jupyter サポート, 強力な REPL
4. 速い！！
 - i. 速い正義
 - ii. 裏が速いライブラリの「芸人」にならなくても、素直に書いてそのまま速い

Julia を使って解かれた・書かれたレポートたち

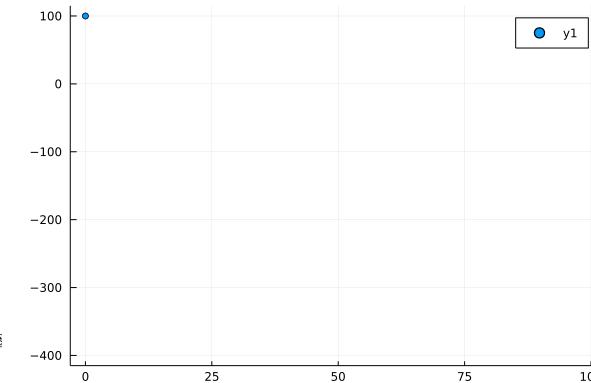
Introduction



② これで各用例に対する実装が完了した。



整列済みの長さ N の配列に対して生成される Treap の高さを調べたグラフである。比較する。



⇒ ✓ 16倍～高速化！！

実装 / オセロ自体の実装

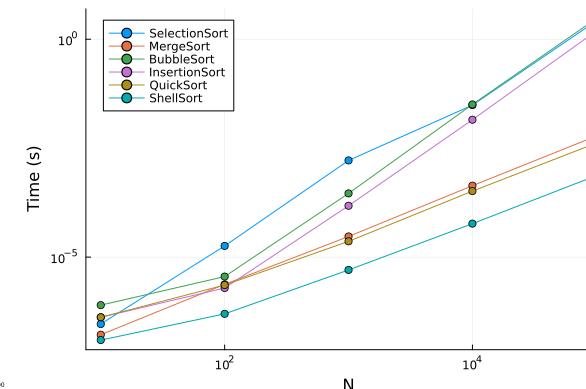
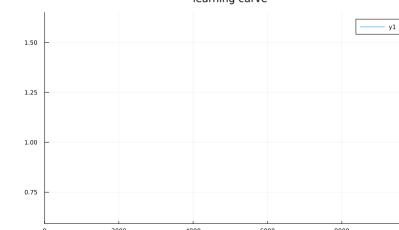
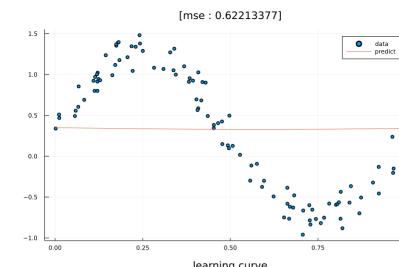
吐かれるネイティブコードを見ると...

ビットボード 二次元配列を使った実装

ビットボードに対する操作は基本的なビット演算で済む
⇒ 少ない命令数で動く
例) 石の数を数える: popcnt命令のみでOK！

実装 / オセロ自体の実装

そのほかの操作もビット演算のみで行える
例1) 合法手の列挙: 6回右シフトを繰り返すことで行える



今日のお話

one of 興味があるもの

機械学習(特に深層学習)

の基盤

The screenshot shows the GitHub page for the JITrench repository. At the top, there are links for 'README' and 'MIT license'. Below that is the repository name 'JITrench'. A central image features a cartoon submarine-like character with three colored circles (green, red, purple) on its deck, swimming in a dark blue trench. Below the image is a subtitle: 'Let's dive into the deep trenches of the loss function with JITrench.jl.' Underneath are several status badges: 'docs stable', 'docs dev', 'build passing'. A section titled 'Install' contains a command-line snippet: `jadd https://github.com/abap34/JITrench.jl`. Another section titled 'Automatic gradient calculation' shows a Julia REPL session:

```
julia> using JITrench
julia> f(x) = sin(x) + 1
f (generic function with 1 method)

julia> JITrench.@diff! f(x)
f' (generic function with 1 method)

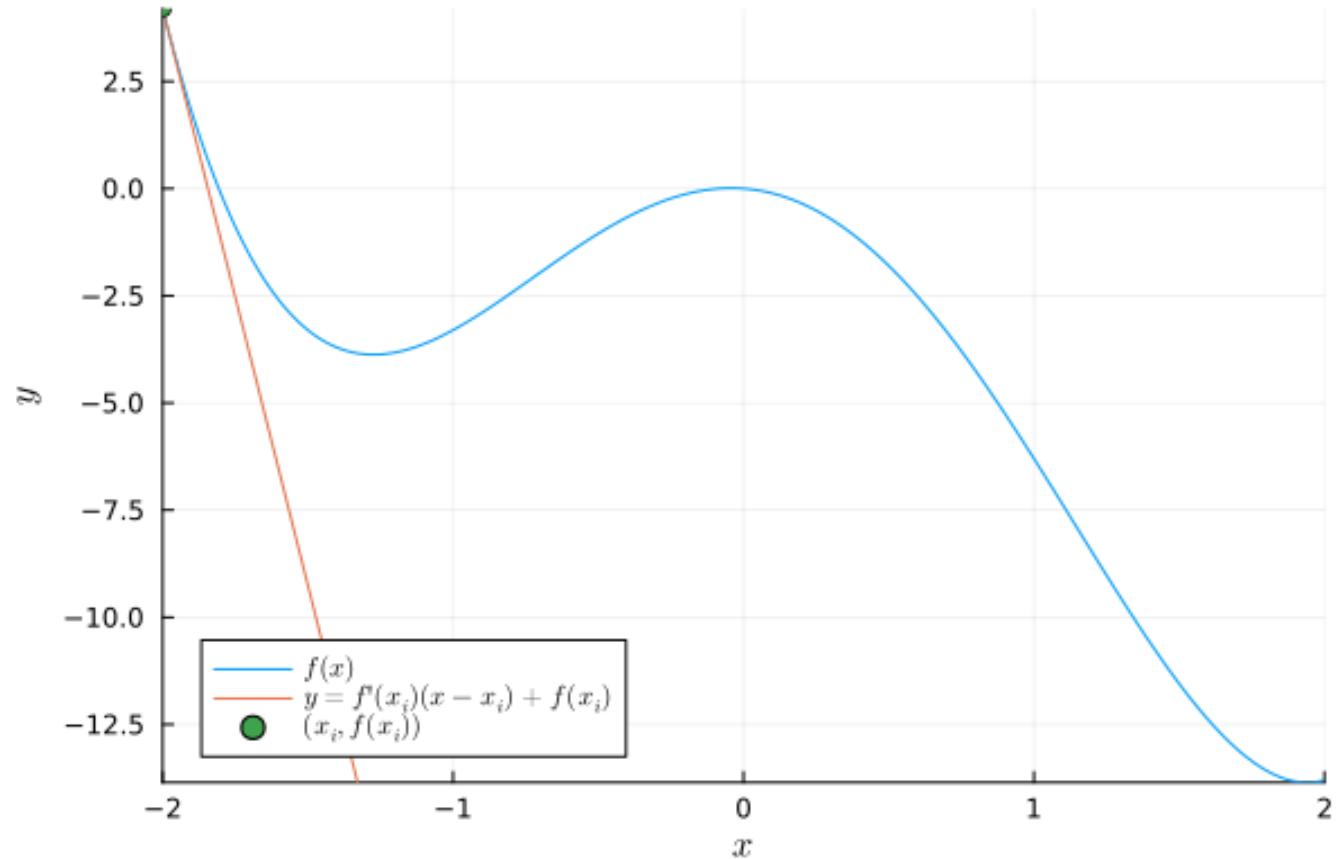
julia> f'(π)
-1.0
```

<https://github.com/abap34/JITrench.jl>

今日のお話

one of 深層学習の基盤

自動微分



こんなことがありますなんか？

1. 深層学習フレームワークを使っているけど、あまり中身がわかっていない。
2. 微分を求めたいことがあって既存のライブラリを使っているけど、どの場面でどれを使うのが適切かわからない
3. 自分の計算ルーチンに微分の計算を組み込みたいけどやり方がわからない
4. え、自動微分ってただ計算するだけじゃないの？何がおもしろいの？

今日話すこと

.....

実は...

- 自動微分は奥が深くて面白い！
- 状況に応じて適切なアルゴリズムを選ぶことで、幸せになれる
- Julia を使うことで簡単に、そして拡張性の高い自動微分エコシステムに乗っかることができる！

[1] 微分と連続最適化

1.1 微分のおさらい

1.2 勾配降下法

1.3 勾配降下法と機械学習

[2] 自動で微分

2.1 自動微分の枠組み

2.2 数式微分 —式の表現と微分

2.3 自動微分 —式からアルゴリズムへ

[3] Juliaに微分させる

3.1 FiniteDiff.jl/FiniteDifferences.jl

3.1 ForwardDiff.jl

3.2 Zygote.jl

[6] 付録

1. 微分を求めることでなにが嬉しくなるのか, なぜ今自動微分が必要なのか理解する



2. いろいろな微分をする手法のメリット・デメリットを理解する



3. Julia でそれぞれを利用 / 拡張する方法を理解する

1. 微分を求めることでなにが嬉しくなるのか, なぜ今微分が必要なのか理解する



3. いろいろな微分をする手法のメリット・デメリットを理解する



4. Julia でそれぞれを利用 / 拡張する方法を理解する

[1] 微分と連続最適化

1.1 微分のおさらい

1.2 勾配降下法

1.3 勾配降下法と機械学習

微分の定義 from 高校数学

[定義1. 微分係数]

関数 f の x における微分係数は

$$\lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

- 偏微分の定義 from 大学数学

x_i について偏微分 $\leftrightarrow x_i$ 以外の変数を固定して微分

[定義2. 偏微分係数]

n 変数関数 f の (x_1, \dots, x_n) の x_i に関する偏微分係数

$$\frac{\partial f}{\partial x_i}(x_1, \dots, x_n) := \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_{i-1}, x_i + h, x_{i+1}, \dots, x_n) - f(x_1, \dots, x_n)}{h}$$

各 x_i について偏微分係数を計算して並べたベクトルを **勾配ベクトル** と呼ぶ

$$\nabla f(x_1, \dots, x_n) := \left(\frac{\partial f}{\partial x_1}(x_1, \dots, x_n), \dots, \frac{\partial f}{\partial x_n}(x_1, \dots, x_n) \right)$$

例) $f(x, y) = x^2 + xy$ の $(1, 2)$ における勾配ベクトルは

$$\begin{cases} \frac{\partial f}{\partial x} = 2x + y \\ \frac{\partial f}{\partial y} = x \end{cases} \Rightarrow \underline{\nabla f(1, 2) = (4, 1)}$$

 勾配ベクトルの重要なポイント

– $-\nabla f(x)$ は x における f の値がもっとも小さくなる方向
を指す

[定理1. 勾配ベクトルの性質]

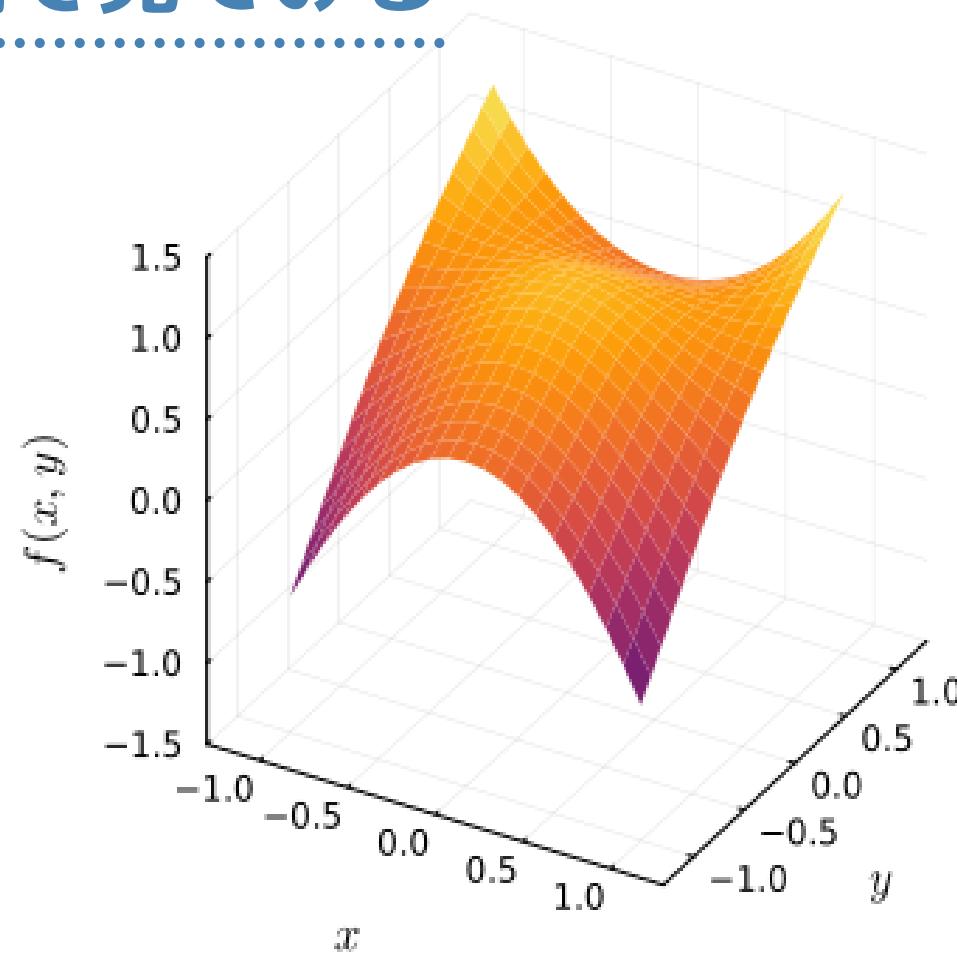
$-\nabla f(\mathbf{x})$ は

$$g(\mathbf{v}) = \lim_{h \rightarrow 0} \frac{f(\mathbf{x}) - f(\mathbf{x} + h\mathbf{v})}{h}$$

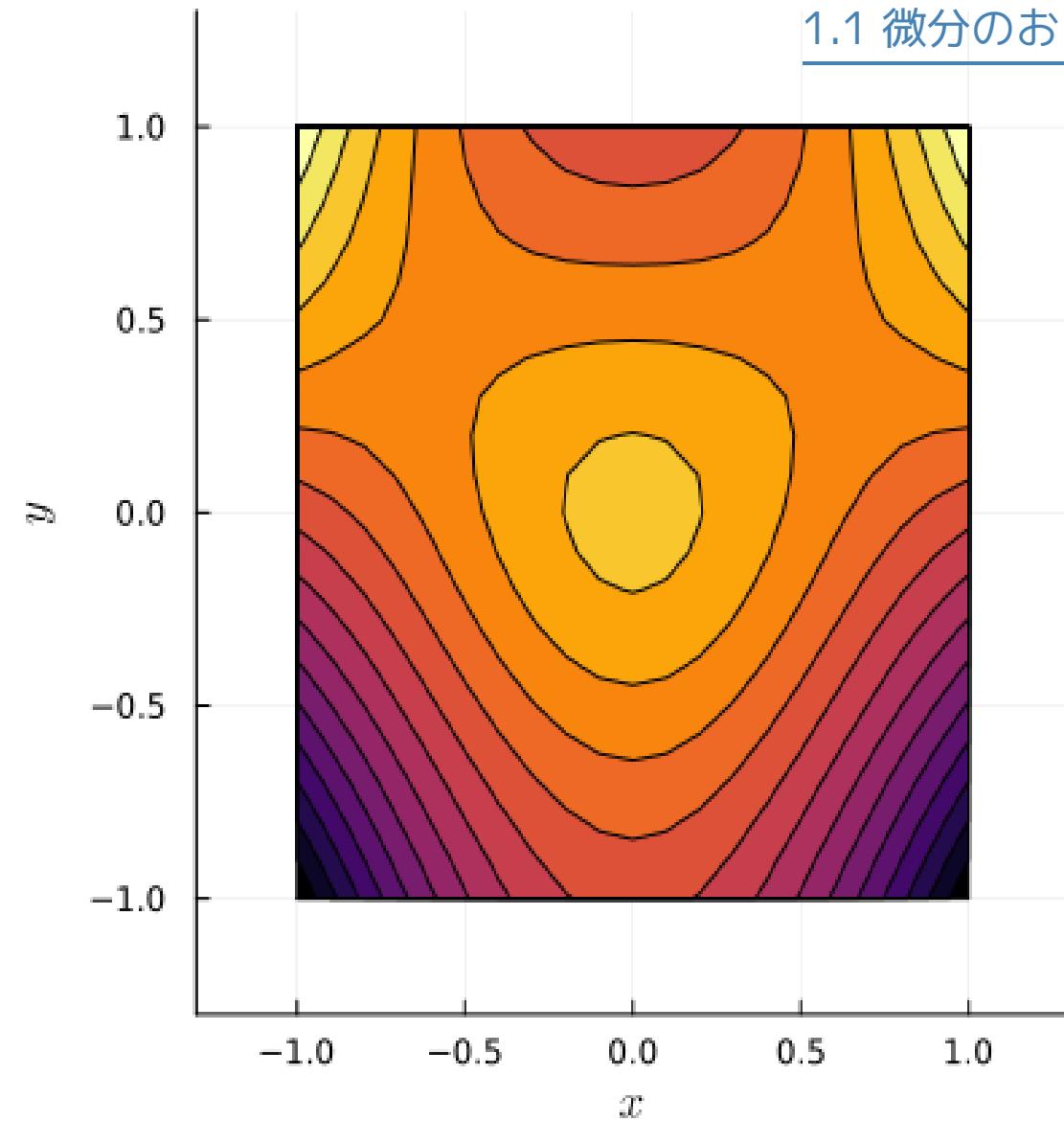
の最大値を与える。

... せっかく Julia を使っているので視覚的に確かめてみる

実例で見てみる



($f(x, y) = x^2y + \frac{1}{x^2+y^2+1}$ のプロット)



実例で見てみる

$$f(x, y) = x^2y + \frac{1}{x^2+y^2+1}$$

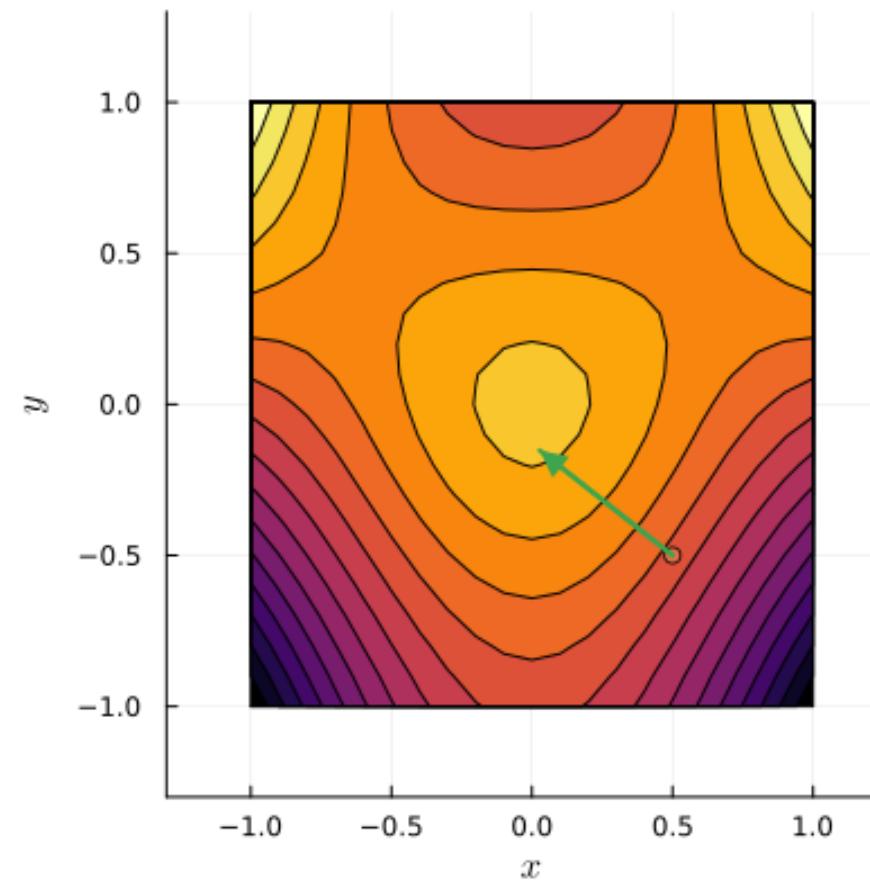
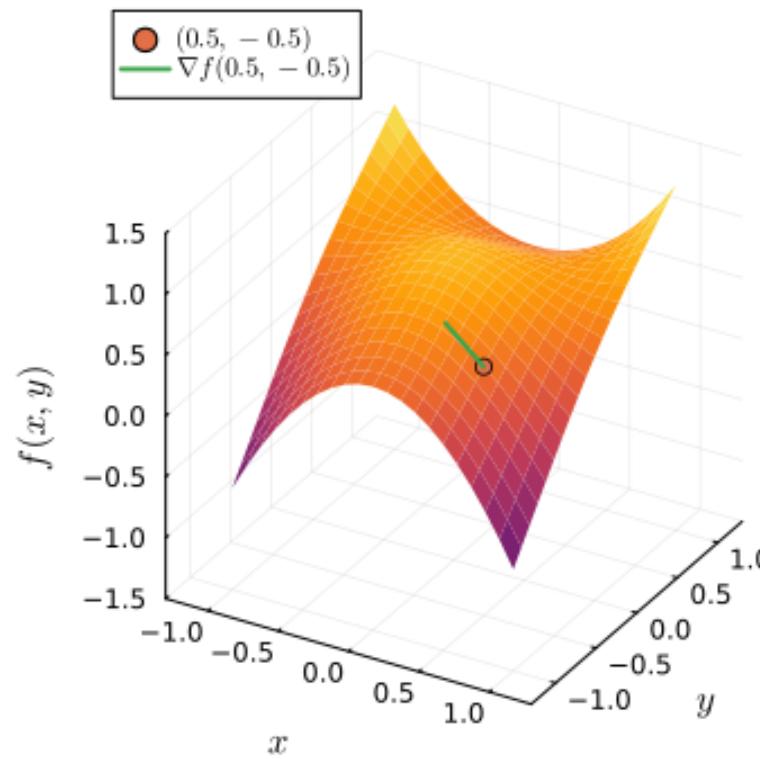
計算すると、

$$\begin{cases} \frac{\partial f}{\partial x} = 2xy - \frac{2x}{(x^2 + y^2 + 1)^2} \\ \frac{\partial f}{\partial y} = x^2 - \frac{2y}{(x^2 + y^2 + 1)^2} \end{cases}$$

なので

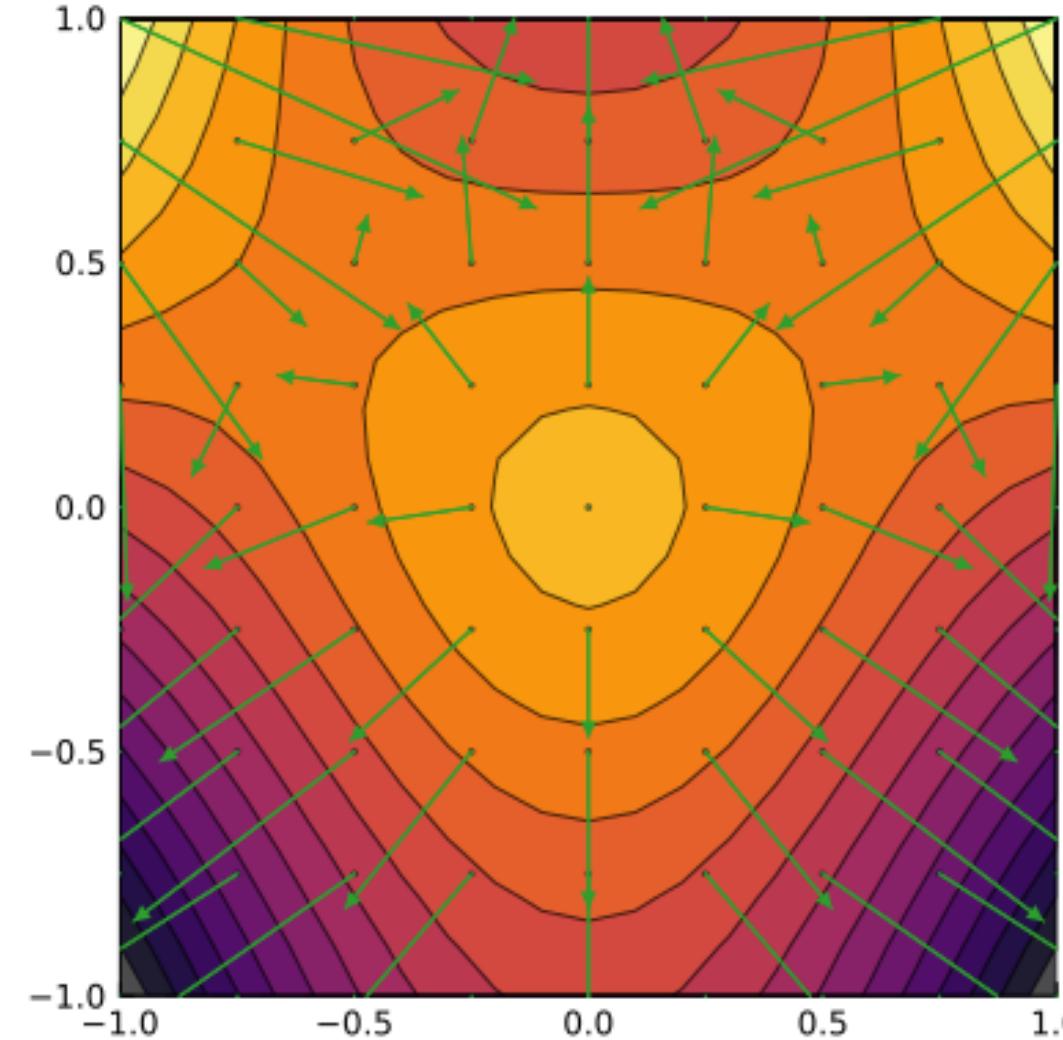
$$\nabla f(0.5, -0.5) = \left(\frac{-17}{18}, \frac{25}{36} \right) = (-0.94, 0.694)$$

実例で見てみる



$\nabla f(0.5, -0.5) = \left(\frac{-17}{18}, \frac{25}{36} \right) = (-0.94, 0.694)$ のプロット

実例で見てみる

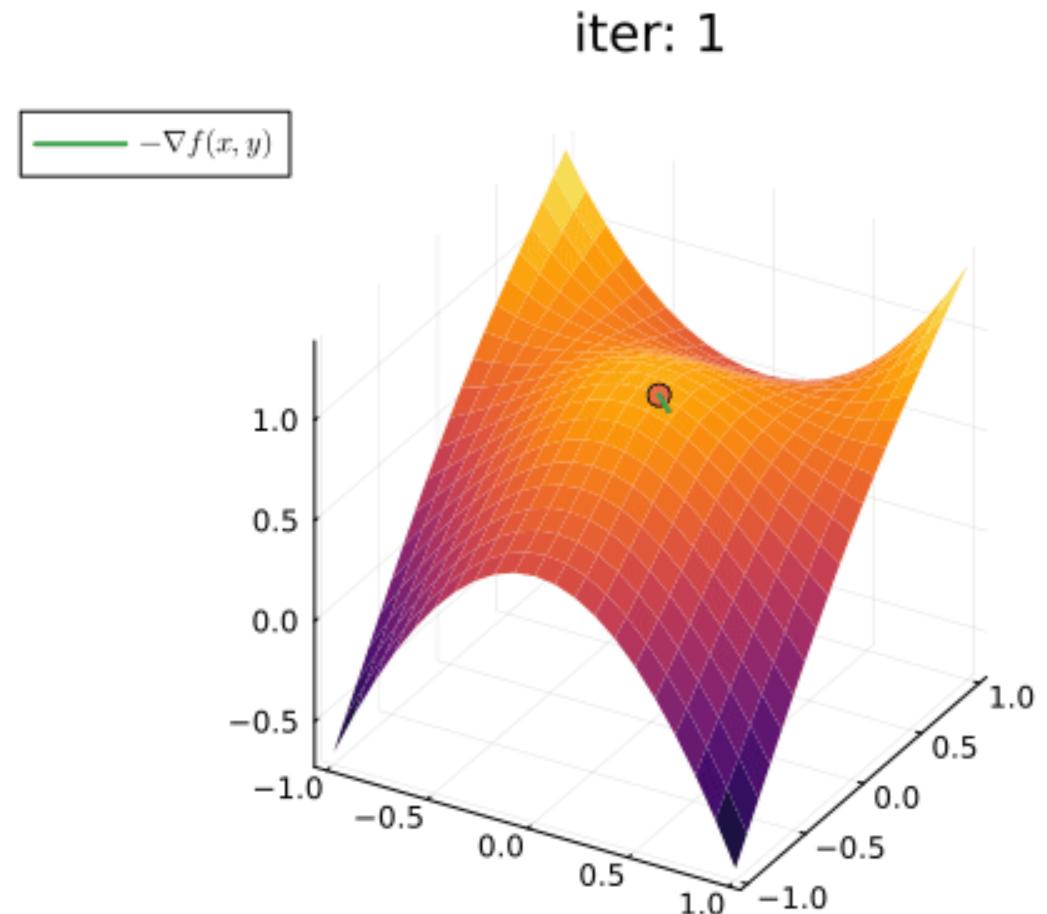


勾配降下法

✓ $-\nabla f(x)$ は小さくなる方を指す



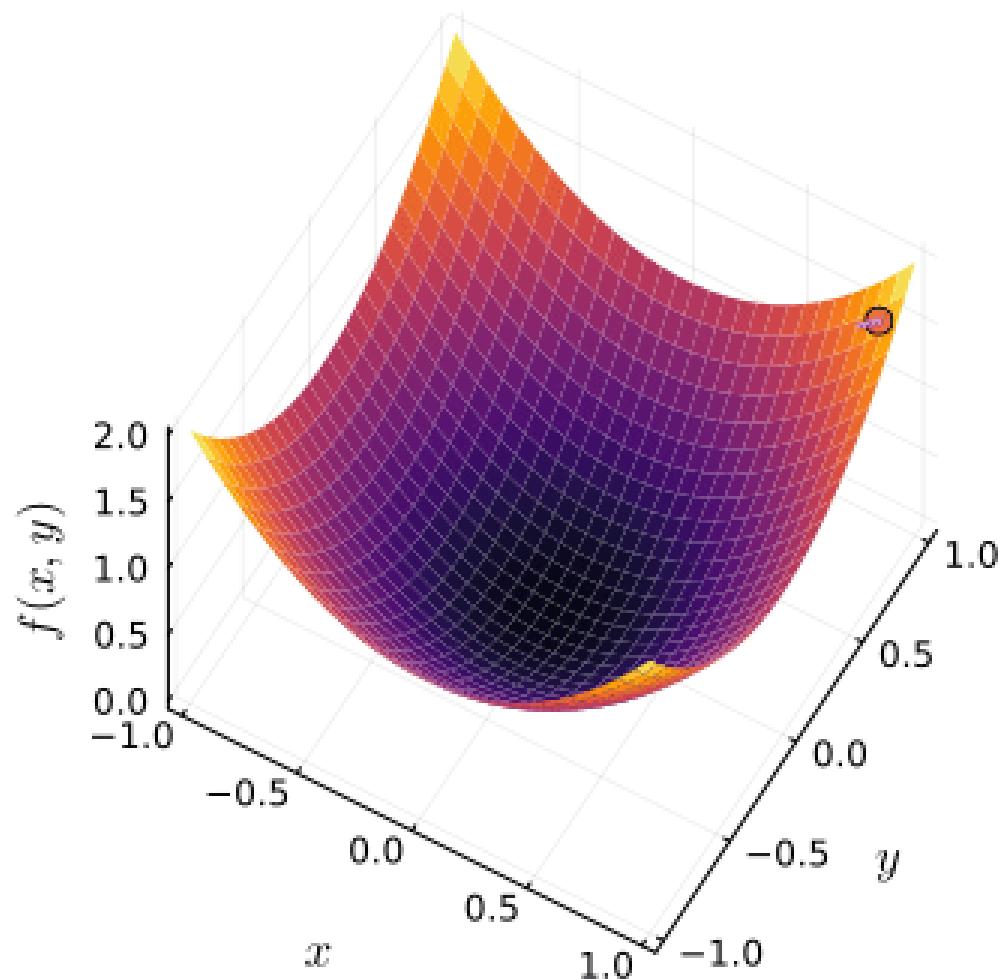
$-\nabla f(x, y)$ の方向にちょっとづつ点を動かしていくけば関数のそこそこ小さい値を取る点を探しに行ける



勾配降下法

$$f(x, y) = x^2 + y^2$$

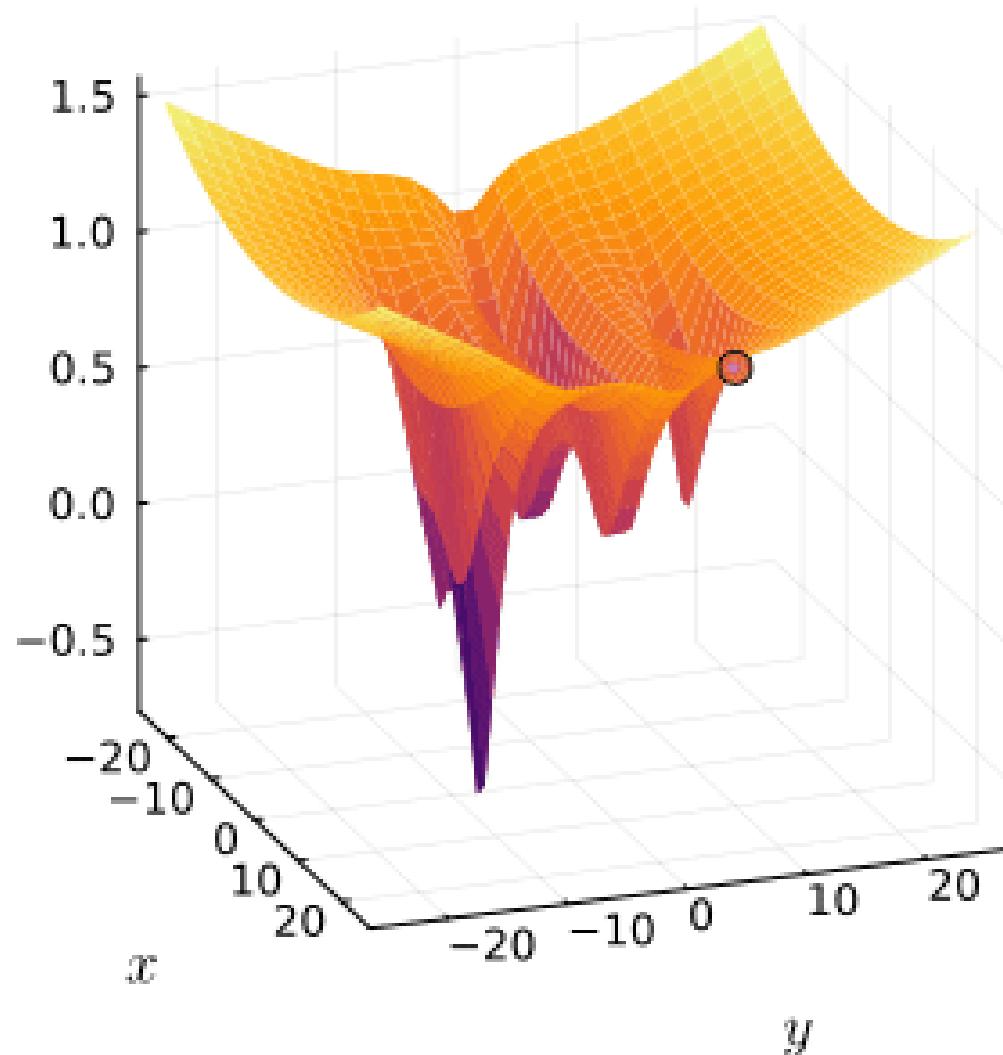
iter: 0



iter: 0

勾配降下法

$$f(x, y) = \left(1 - \frac{1}{1 + 0.05 \cdot x^2 + (y - 10)^2} \right. \\ - \frac{1}{1 + 0.05 \cdot (x - 10)^2 + y^2} \\ - \frac{1.5}{1 + 0.03 \cdot (x + 10)^2 + y^2} \\ - \frac{2}{1 + 0.05 \cdot (x - 5)^2 + (y + 10)^2} \\ - \frac{1}{1 + 0.1 \cdot (x + 5)^2 + (y + 10)^2} \\ \left. \cdot \left(1 + 0.0001 \cdot (x^2 + y^2)^{1.2} \right) \right)$$



元ネタ: Ilya Pavlyukevich, "Levy flights, non-local search and simulated annealing", Journal of Computational Physics 226 (2007) 1830-1844.

勾配降下法を機械学習に応用する

機械学習で解きたくなる問題

データ $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ があるので、
パラメータ θ を変化させて 損失 $L(\mathcal{D}; \theta)$ をなるべく小さくせよ

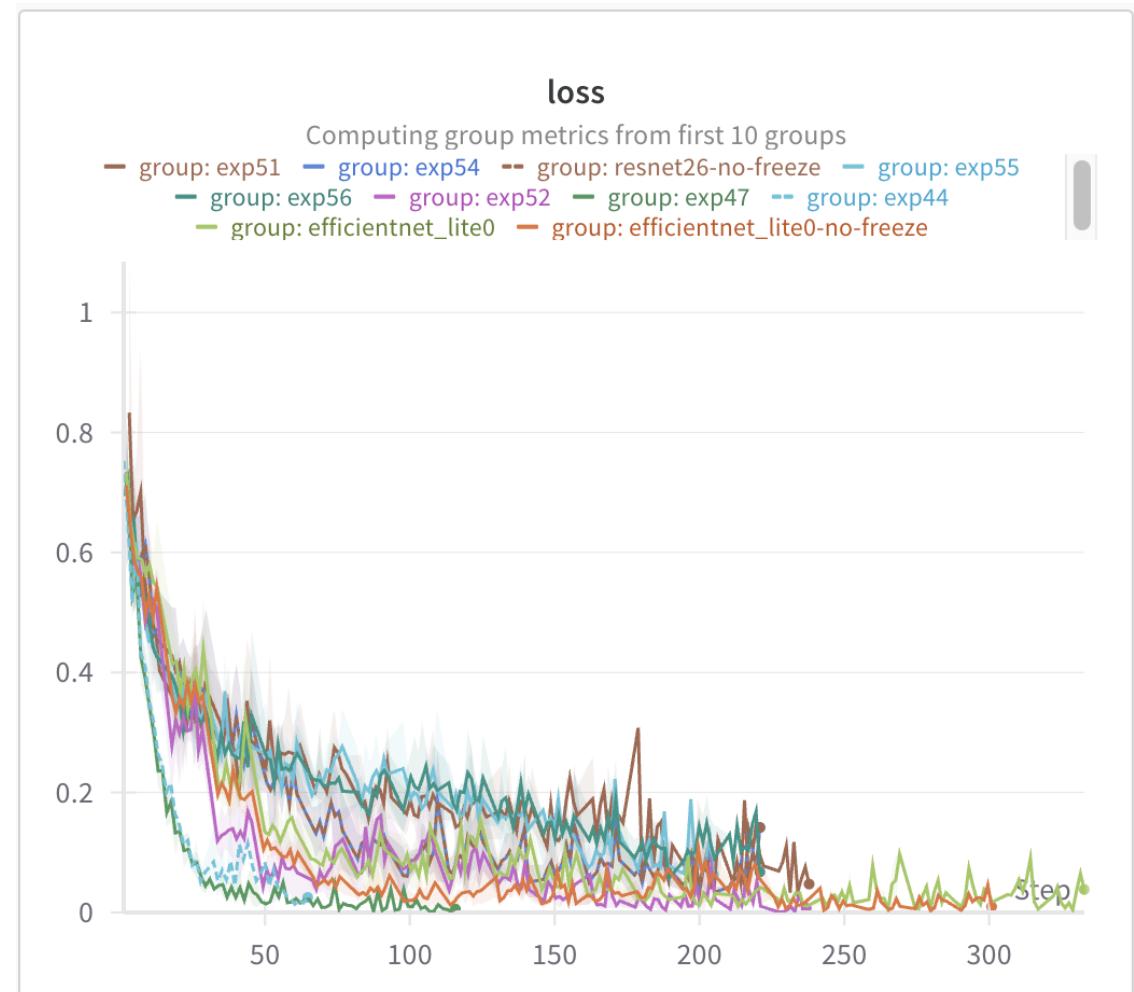


関数の小さい値を探しに行く問題

↔ 勾配降下法チャンス！

勾配降下法と深層学習

勾配降下法を使った深層学習モデルのパラメータの最適化は、
● ● ● ● ● ● ● ●
実際やってみると 非常に上手くいく



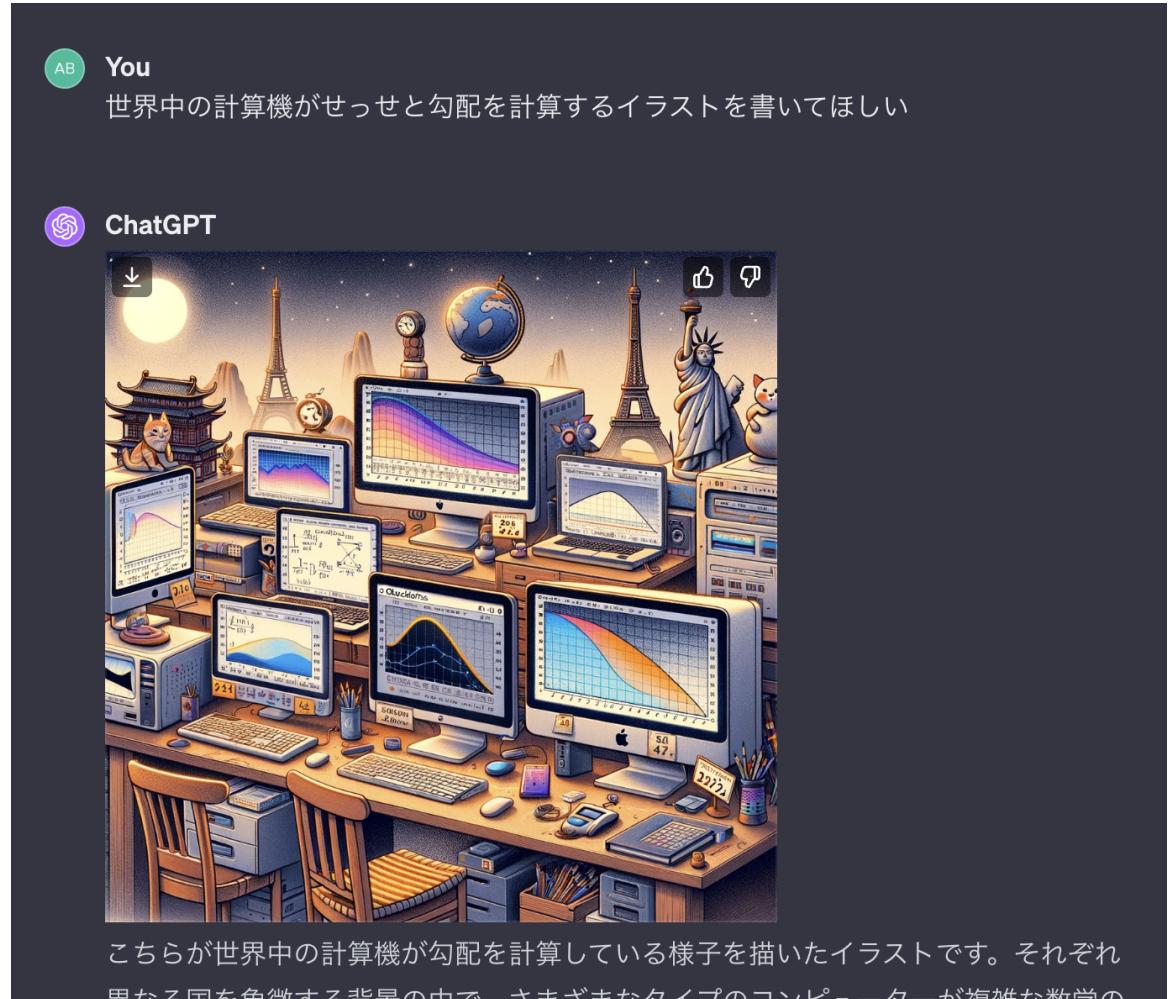
勾配下降法と深層学習

基本的に、深層学習モデルは
勾配下降法を使って訓練

⇒ 今この瞬間も世界中の計算機が
せっせと勾配ベクトルを計算中

2050年にはAI業務サーバの消費電力は 3000 Twh にのぼると予測されているらしいです。[\(https://www.jst.go.jp/lcs/pdf/fy2020-pp-03.pdf\)](https://www.jst.go.jp/lcs/pdf/fy2020-pp-03.pdf)

このうちどれだけの電力が学習(+そのうちの勾配の計算)に使われているかはわかりませんが、上の電力では日産リーフ五億六千万台を地球一周させることができます。そう考えると勾配の計算の効率化を考えることに多少は時間を使っても良さそうな気になってきます。



データ $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ があるので、

パラメータ θ を変化させて 損失 $L(\mathcal{D}; \theta)$ をなるべく小さくせよ



勾配降下法で解くには...

∇L を使って θ を更新して小さい値を探索していく

データ $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ があるので、

パラメータ θ を変化させて 損失 $L(\mathcal{D}; \theta)$ をなるべく小さくせよ



勾配降下法で解くには...

$\nabla L(\theta)$ の値を使って θ を更新して小さい値を探索していく

... $\nabla L(\theta)$ をどうやって計算する？

勾配の計算法を考える

$$\text{さっきは } f(x, y) = x^2y + \frac{1}{x^2+y^2+1}$$

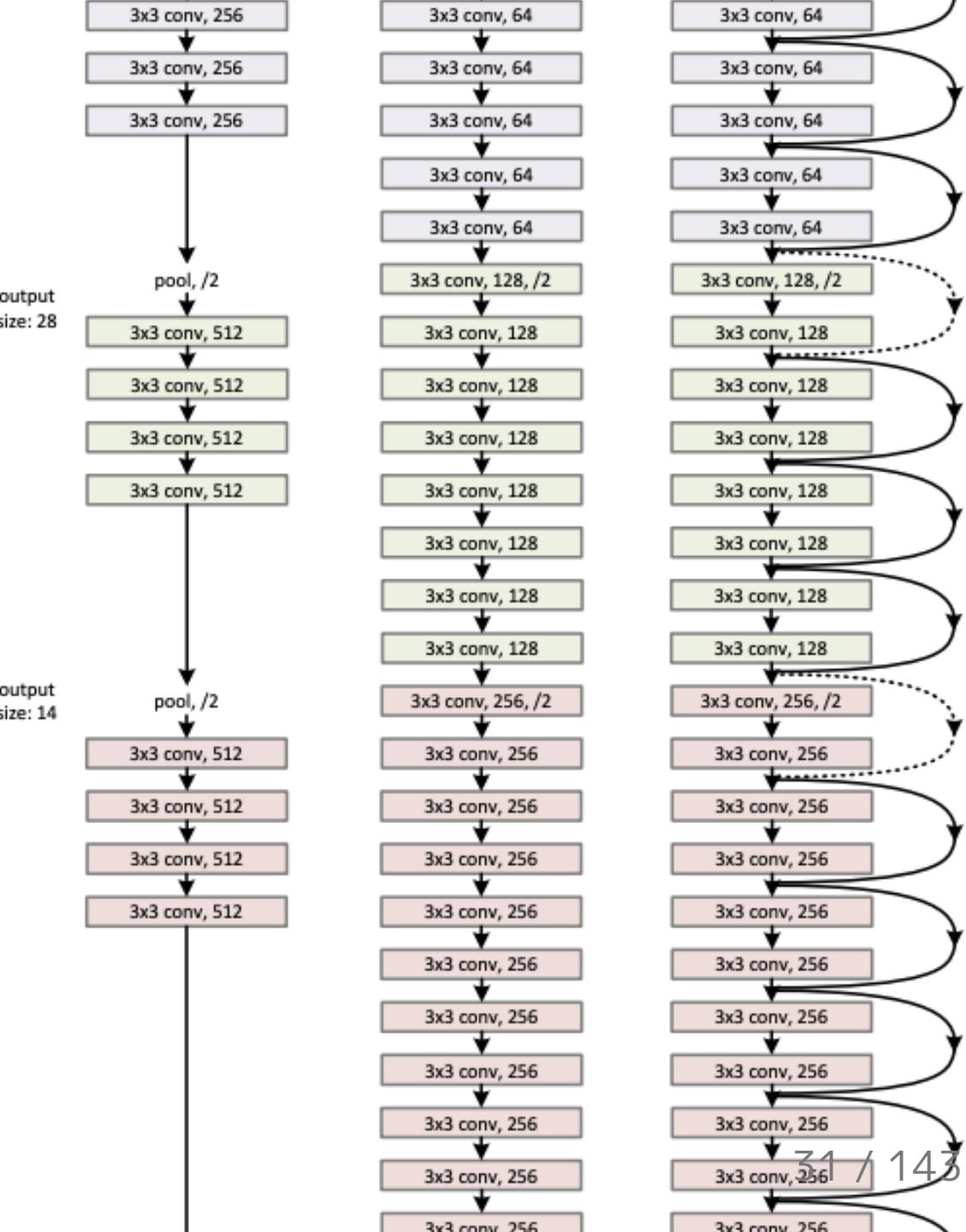
⇒ 頑張って手で ∇f を求められた

深層学習の複雑なモデル...

$$L(\theta; x, y) = \text{VeryComplicatedf}(\theta; x, y)$$



とてもつらい。



画像: He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep Residual Learning for Image Recognition. ArXiv. /abs/1512.03385

アイデア1. 近似によって求める？

$$f'(x) = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

⇒ 実際に小さい h をとって計算する。

```
function diff(f, x; h=1e-8)
    return (f(x + h) - f(x)) / h
end
```

これでもそれなりに近い値を得られる.

例) $f(x) = x^2$ の $x = 2$ における微分係数 4 を求める.

```
julia> function diff(f, x; h=1e-8)
           return (f(x + h) - f(x)) / h
       end
```

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

```
julia> diff(x → x^2, 2)    # おしい!
3.99999975690116
```

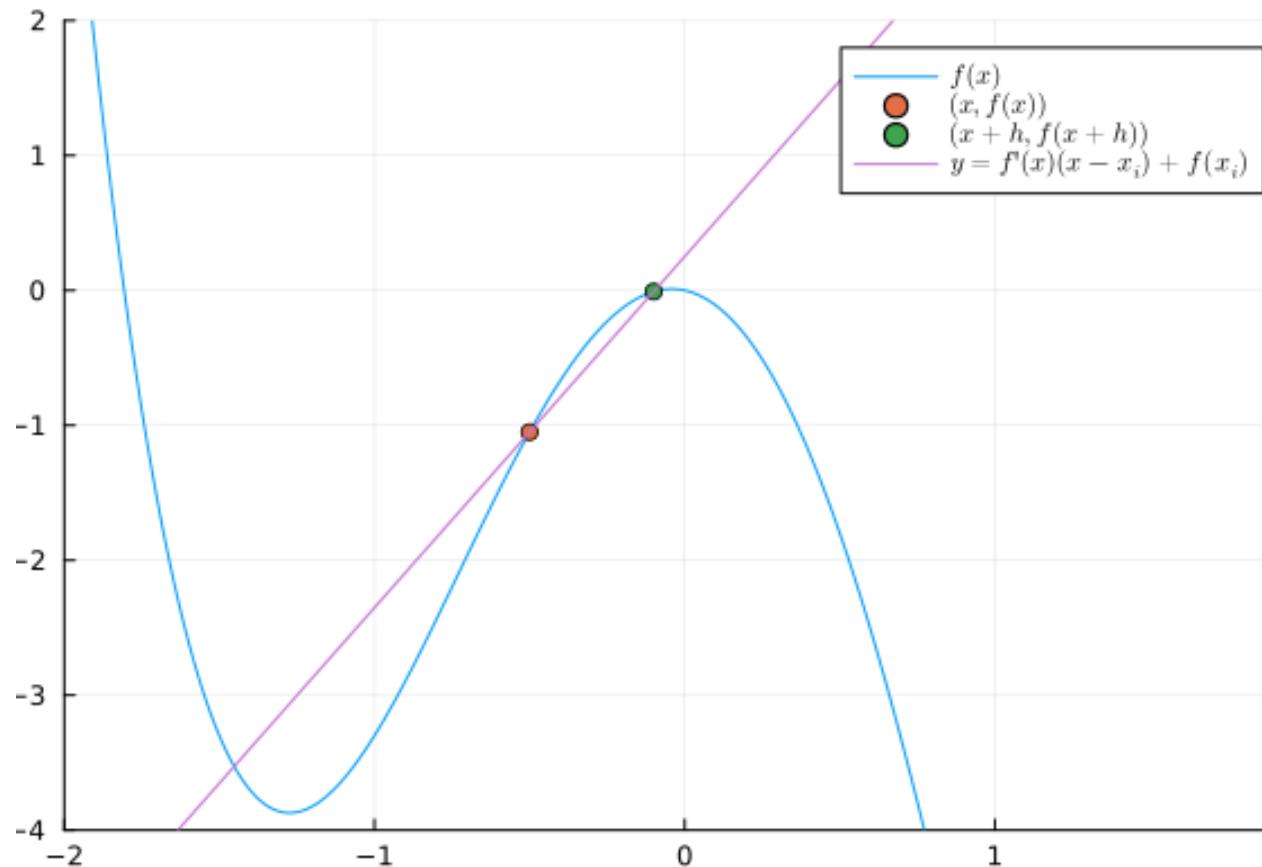
数値微分

実際に小さい h をとって計算

「数値微分」

お手軽だが...

- 誤差が出る
- 勾配ベクトルの計算が非効率



問題点①. 誤差が出る

1. 本来極限をとるのに、小さい h をとって計算しているので誤差が出る
2. 分子が極めて近い値同士の引き算になっていて、 $\left(\frac{f(x+h)-f(x)}{h} \right)$ 衍落ちによって精度が大幅に悪化。

問題点②. 勾配ベクトルの計算が非効率

1. n 変数関数の勾配ベクトル
 $\nabla f(\mathbf{x}) \in \mathbb{R}^n$ を計算するには、各 x_i について「少し動かす → 計算」を繰り返すので n 回 f を評価する。
2. 応用では n がとても大きくなり、 f の評価が重くなりがちで **致命的**

- 微分をすることによる誤差なく
- 高次元の勾配ベクトルを効率よく計算できなか?

 <できますよ



 自動微分の世界へ

1. 微分を求めることでなにが嬉しくなるのか, なぜ今微分が必要なのか理解する



3. いろいろな微分をする手法のメリット・デメリットを理解する



4. Julia でそれぞれを利用 / 拡張する方法を理解する

1. 微分を求めることでなにが嬉しくなるのか, なぜ今微分が必要なのか理解する



2. いろいろな微分をする手法のメリット・デメリットを理解する



3. Julia でそれぞれを利用 / 拡張する方法を理解する

[2] 自動で微分

2.1 自動微分の枠組み

2.2 数式微分 — 式の表現と微分と連鎖律

2.3 自動微分 — 式からアルゴリズムへ

2.4 自動微分とトレース

2.5 自動微分とソースコード変換

2.1 自動微分の枠組み

- ✓ 計算機上で微分するためには、計算機上で関数を表現しないといけない。

[定義. 自動微分]

(数学的な関数を表すように定義された) **計算機上のアルゴリズム** を入力とし,
その関数の任意の点の微分係数を無限精度の計算モデル上で正確に計算できる

計算機上のアルゴリズム を出力するアルゴリズムを

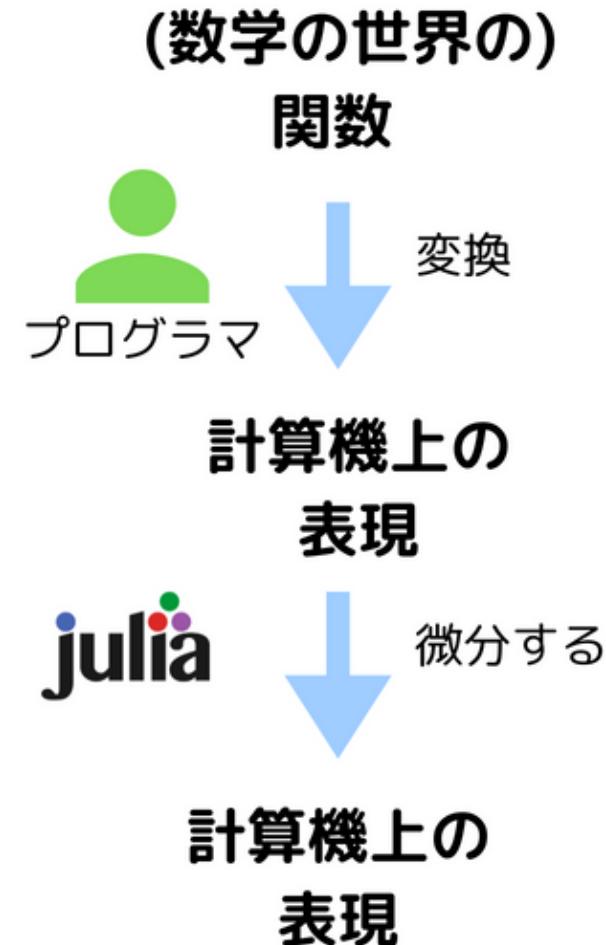
「**自動微分(Auto Differentiation, Algorithmic Differentiation)**」

と呼ぶ。

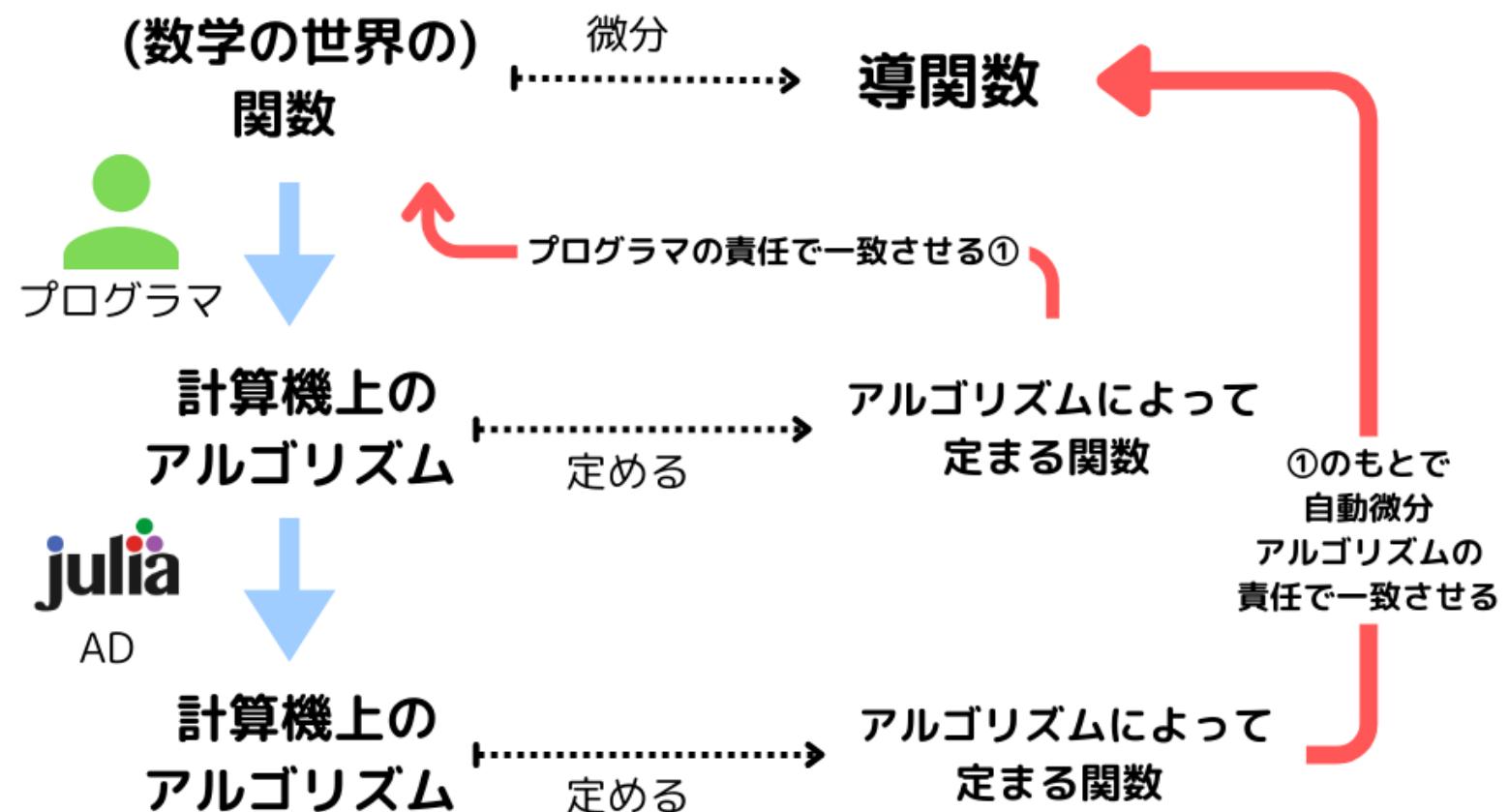
自動微分の枠組み

計算機は、

- 計算機上の表現をもらって
- 計算機上の表現を返す。



自動微分



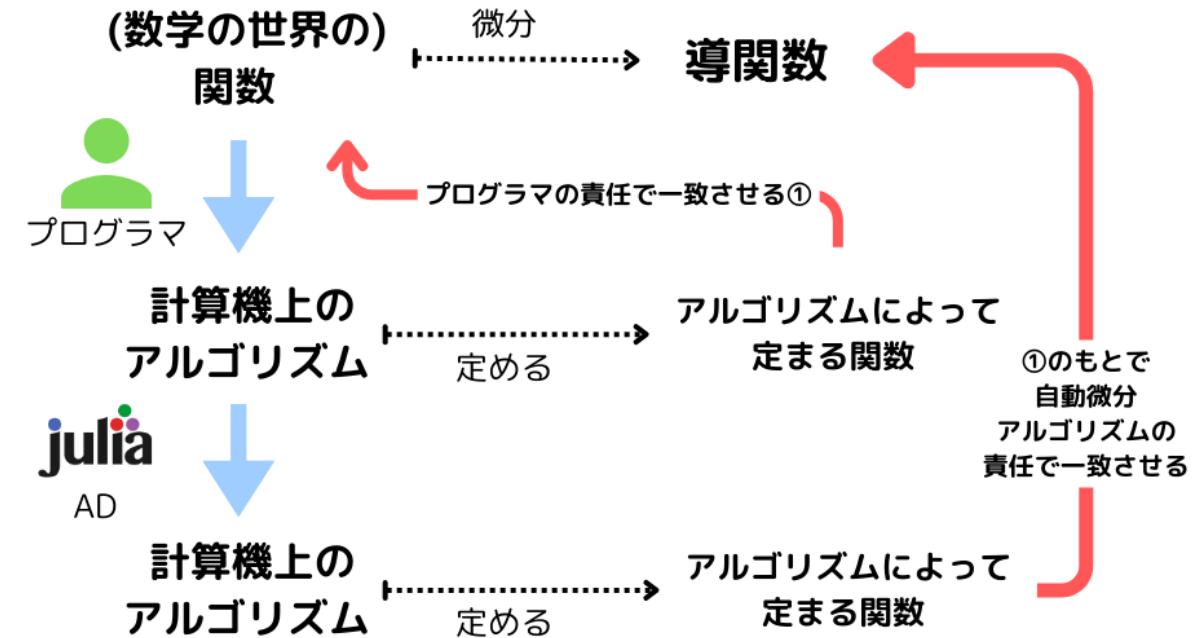
[例: 二次関数の微分]

 < $f(x) = x^2$ の微分がわからなので、自動微分で計算したい

例：二次関数の微分

1. 関数 → アルゴリズム
by プログラマー

```
function f(x::InfinityPrecisionFloat)
    return x^2
end
```



例：二次関数の微分

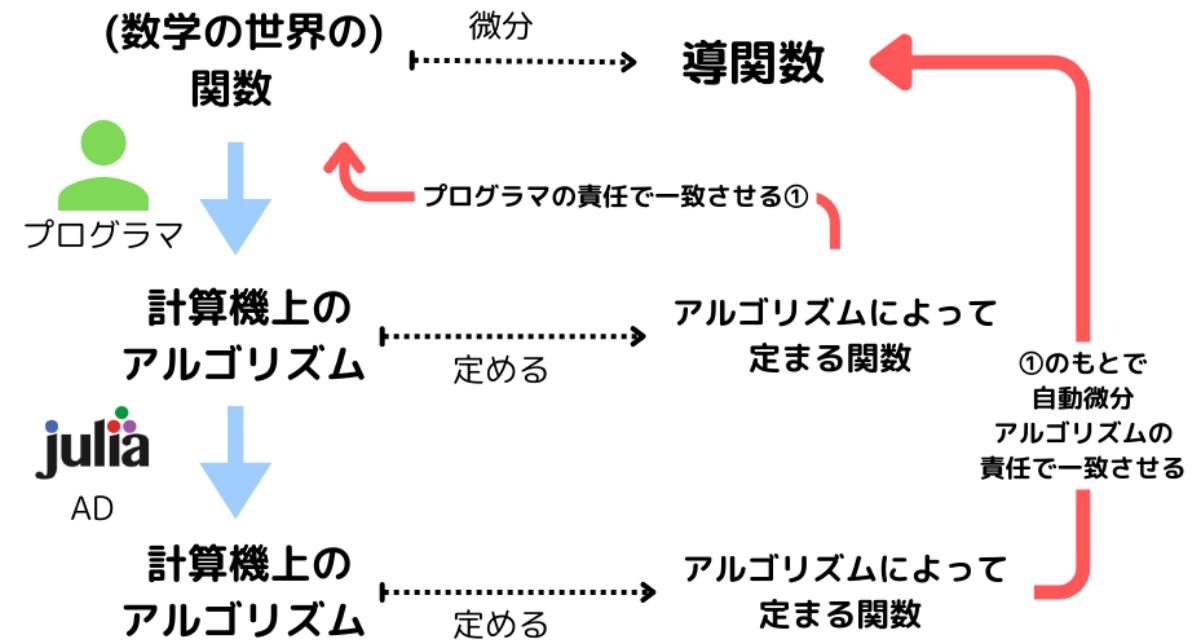
2. アルゴリズム → アルゴリズム
by 自動微分ライブラリ

```
using AutoDiffLib # ※ 存在ないです！

function f(x::InfinityPrecisionFloat)
    return x^2
end

df = AutoDiffLib.differentiate(f)

df(2.0) # 4.0
df(3.0) # 6.0
```



[例：二次関数の微分]

🐶 < $f(x) = x^2$ の微分がわからぬので、自動微分で計算したい

- ✓ プログラムに直したプログラマがミスっていなければ
- ✓ 自動微分ライブラリがバグっていなければ

正しい微分係数を計算できるアルゴリズムを入手できた

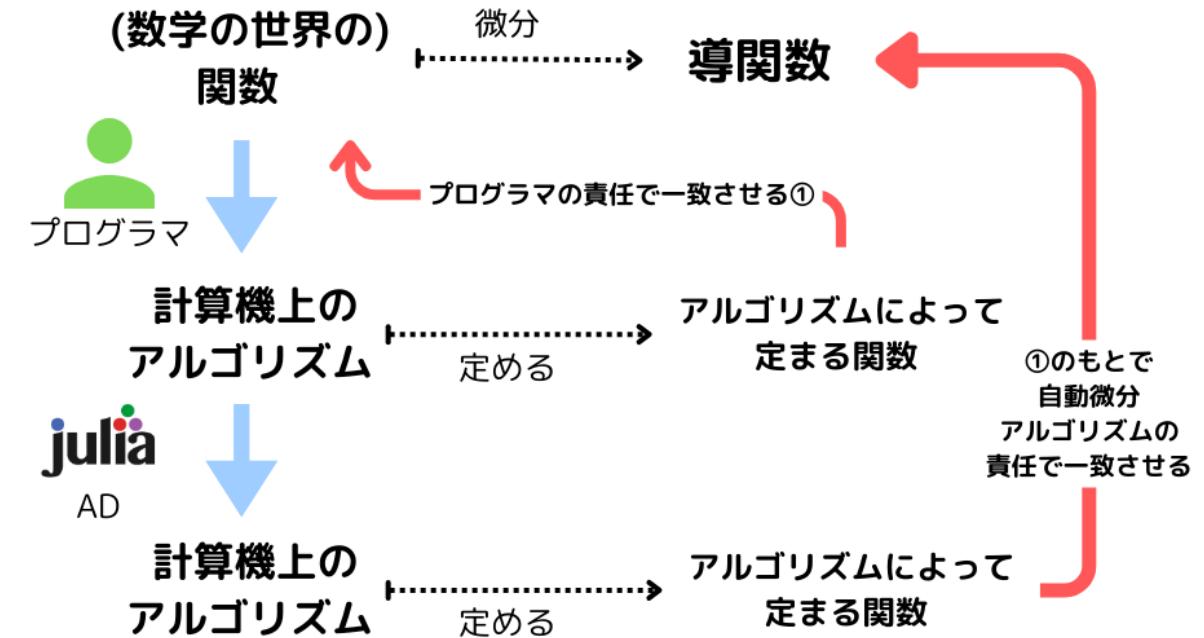
自動微分の枠組み

で、実際に

どうやって微分する？



自動微分の実装へ



2.2 数式微分 一式の表現と微分

```
function symbolic_derivative(f::Function)::Function
    g = symbolic_operation(f)
    return g
end
```

[定義. 自動微分]

(数学的な関数を表すように定義された) **計算機上のアルゴリズム** を入力とし,
その関数の任意の点の微分係数を無限精度の計算モデル上で正確に計算できる

計算機上のアルゴリズム を出力するアルゴリズムを

「**自動微分(Auto Differentiation, Algorithmic Differentiation)**」

と呼ぶ。

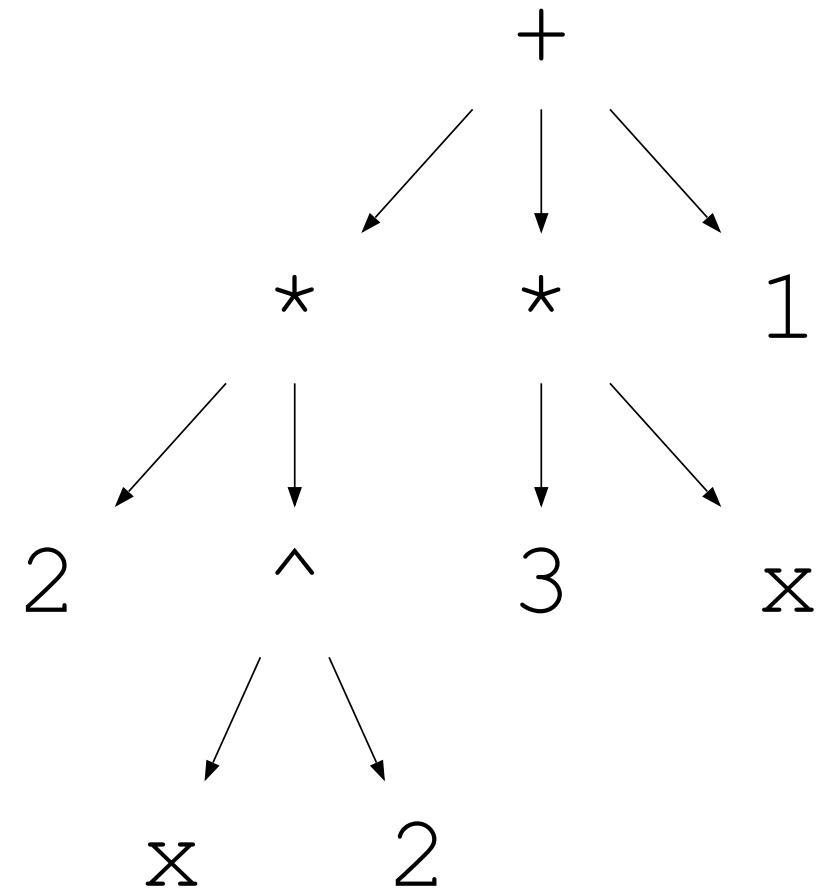


アルゴリズム を計算機上でどう表現するか？

数式微分のアイデア

単純・解析しやすい表現
... 式をそのまま木で表す

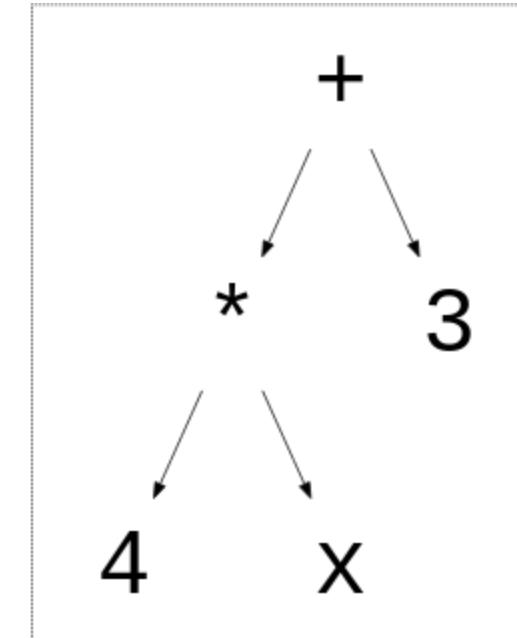
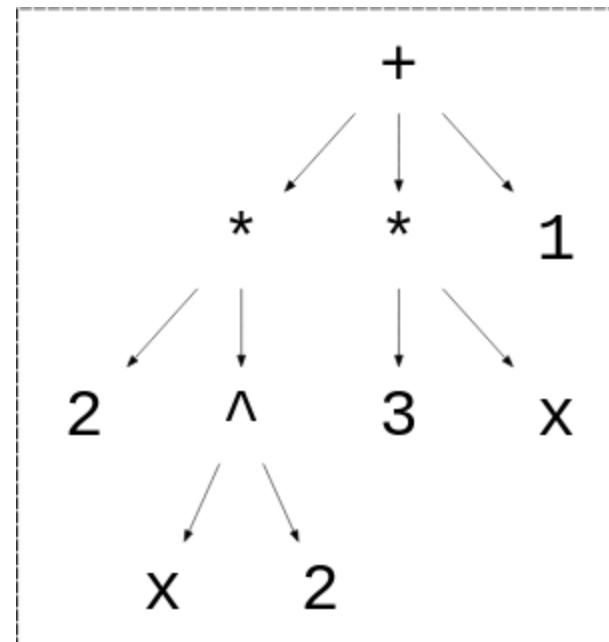
$$2x^2 + 3x + 1 \Rightarrow$$



数式微分のアイデア

2.2 数式微分 - 式の表現と微分

この木をもとに導関数を表現する木を得たい！



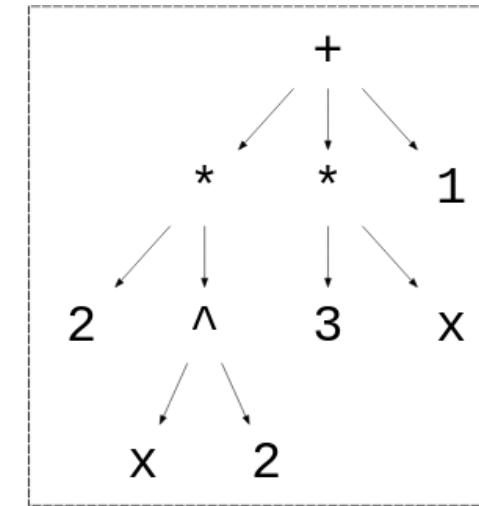
$$2x^2 + 3x + 1$$

$$4x + 3$$

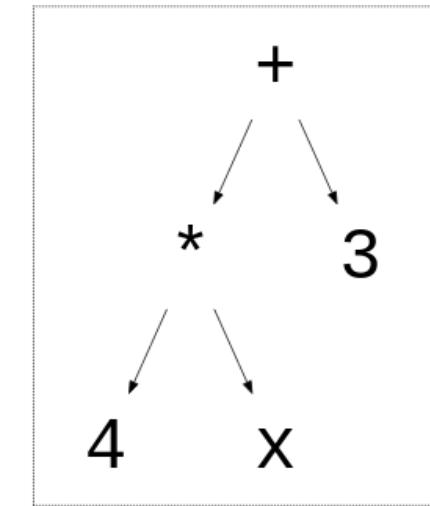
数式微分のアイデア

$$f(x) = 2x^2 + 3x + 1$$

変換



操作



- ✓ Juliaなら簡単に式の木構造による表現を得られる。

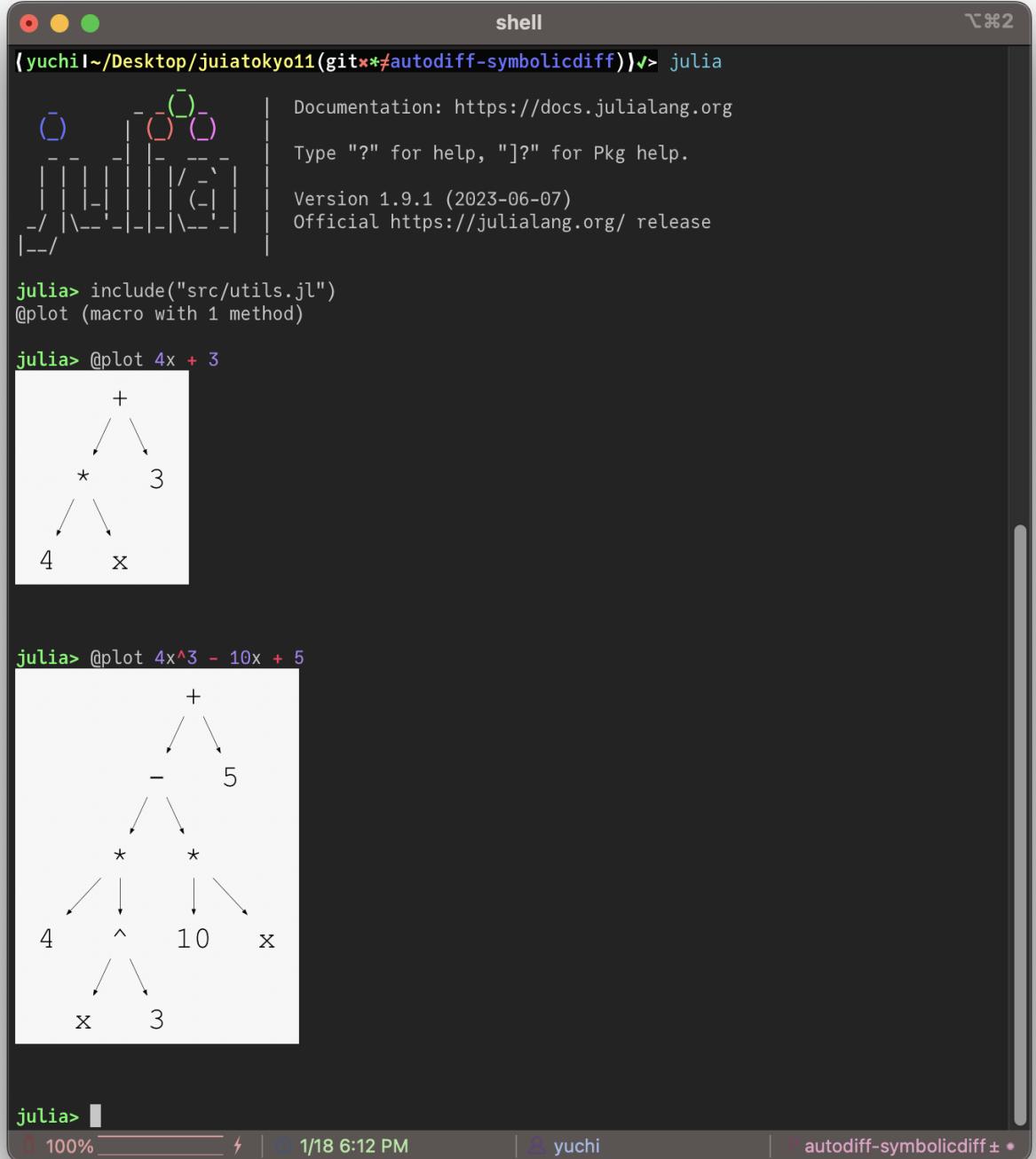
```
julia> f = :(4x + 3) # or Meta.parse("4x + 3")
:(4x + 3)
```

```
julia> dump(f)
Expr
  head: Symbol call
  args: Array{Any}((3,))
    1: Symbol +
    2: Expr
      head: Symbol call
      args: Array{Any}((3,))
        1: Symbol *
        2: Int64 4
        3: Symbol x
    3: Int64 3
```

Expr 型

Expr 型の可視化

— 構造が保持されてる



The screenshot shows a Julia shell window with the following content:

```
(yuchi@~/Desktop/juiatokyo11(git***autodiff-symbolicdiff))> julia
julia> include("src/utils.jl")
@plot (macro with 1 method)

julia> @plot 4x + 3
+---+
|   |
*--- 3
|   |
4   x
```

The second part of the screenshot shows a more complex expression:

```
julia> @plot 4x^3 - 10x + 5
+---+
|   |
---- 5
|   |
*--- *
|   |
4   ^   10   x
|   |
x   3
```

1. 定数を微分できるようにする

$$\frac{d}{dx}(c) = 0$$

```
julia> derivative(ex::Int64) = 0
```

2. x についての微分は 1

$$\frac{d}{dx}(x) = 1$$

```
derivative(ex::Symbol) = 1
```

3. 足し算に関する微分

$$\frac{d}{dx}(f(x) + g(x)) = f'(x) + g'(x)$$

```
function derivative(ex::Expr)::Expr
    op = ex.args[1]
    if op == :+
        return Expr(
            :call,
            :+,
            derivative(ex.args[2]),
            derivative(ex.args[3])
        )
    end
end
```

4. 掛け算に関する微分

$$\frac{d}{dx}(f(x) \cdot g(x)) = f'(x) \cdot g(x) + f(x) \cdot g'(x)$$

```
function derivative(ex::Expr)::Expr
    op = ex.args[1]
    if op == :+
        ...
    elseif op == :*
        return Expr(
            :call,
            :+,
            Expr(:call, :*, ex.args[2], derivative(ex.args[3])),
            Expr(:call, :*, derivative(ex.args[2]), ex.args[3])
        )
    end
end
```

```
derivative(ex::Symbol) = 1      # dx/dx = 1
derivative(ex::Int64) = 0        # 定数の微分は 0

function derivative(ex::Expr)::Expr
    op = ex.args[1]
    if op == :+
        return Expr(:call, :+, derivative(ex.args[2]), derivative(ex.args[3]))
    elseif op == :*
        return Expr(
            :call,
            :+,
            Expr(:call, :*, ex.args[2], derivative(ex.args[3])),
            Expr(:call, :*, derivative(ex.args[2]), ex.args[3])
        )
    end
end
```

※ Juliaは `2 * x * x` のような式を、`(2 * x) * x` でなく `*(2, x, x)` として表現するのでこのような式については上は正しい結果を返しません。(スペースが足りませんでした)
このあたりもちゃんとやるやつは付録のソースコードを見てください。基本的には二項演算の合成とみて順にやっていくだけで良いです。

例) $f(x) = x^2 + 3$ の導関数 $f'(x) = 2x$ を求めて $x = 2, 10$ での微分係数を計算

```
julia> f = :(x * x + 3)
:(x * x + 3)
```

```
julia> df = derivative(f)
:((x * 1 + 1x) + 0)
```

```
julia> x = 2; eval(df)
4
```

```
julia> x = 10; eval(df)
20
```



数式微分の改良 ~ 複雑な表現

```
df = ((x * 1 + 1x) + 0) ... 2x にはなっているが冗長?
```

自明な式の簡約を行ってみる

- 足し算の引数から 0 を除く.
- 掛け算の引数から 1 を除く.

```
function add(args)
    args = filter(x → x != 0, args)
    if length(args) == 0
        return 0
    elseif length(args) == 1
        return args[1]
    else
        return Expr(:call, :+, args...)
    end
end
```

- 掛け算の引数から 1 を取り除く.

```
function mul(args)
    args = filter(x → x != 1, args)
    if length(args) == 0
        return 1
    elseif length(args) == 1
        return args[1]
    else
        return Expr(:call, :*, args...)
    end
end
```

数式微分 + 自明な簡約

```
derivative(ex::Symbol) = 1
derivative(ex::Int64) = 0

function derivative(ex::Expr)
    op = ex.args[1]
    if op == :+
        return add([derivative(ex.args[2]), derivative(ex.args[3])])
    elseif op == :*
        return add([
            mul([ex.args[2], derivative(ex.args[3])]),
            mul([derivative(ex.args[2]), ex.args[3]])])
    end
end
```

✓ 簡単な式を得られた

```
julia> derivative(:(:x * x + 3))
:(x + x)
```

⇒ ではこれでうまくいく？

```
julia> derivative(:((1 + x) / (2 * x^2)))
:((2 * x ^ 2 - (1 + x) * (2 * (2x))) / (2 * x ^ 2) ^ 2)
```

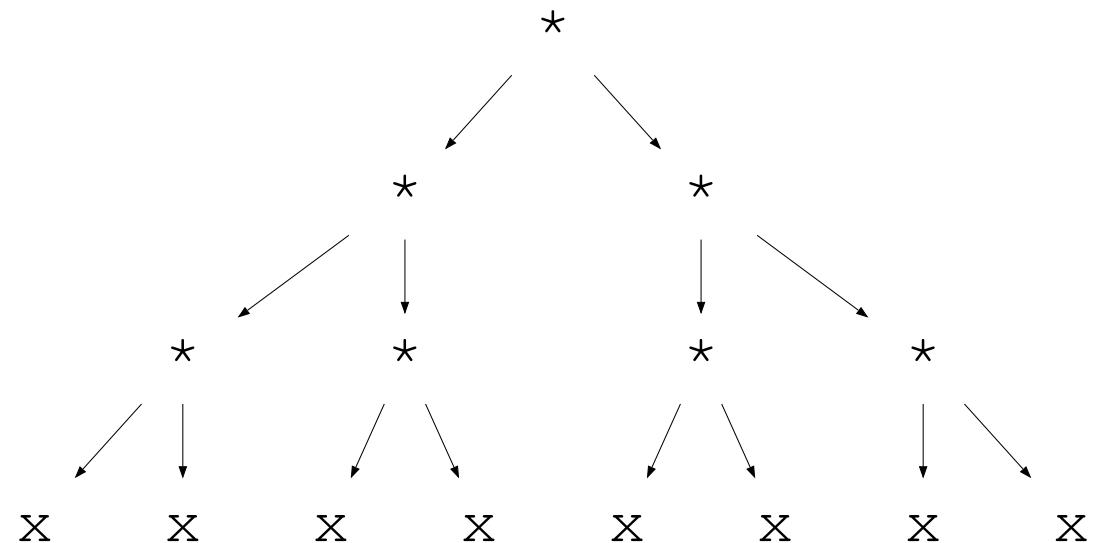
$$= \frac{(2 \cdot x^2 - (1 + x) \cdot 2 \cdot 2 \cdot x)}{(2 \cdot x^2)^2} = -\frac{x + 2}{2x^3}$$



式の表現法を考える

```
julia> t1 = :(x * x)
julia> t2 = :($t1 * $t1)
julia> f = :($t2 * $t2)
:(((x * x) * (x * x)) * ((x * x) * (x * x)))
```

という f は、木で表現すると…



```
julia> t1 = :(x * x)
julia> t2 = :($t1 * $t1)
julia> f = :($t2 * $t2)
:(((x * x) * (x * x)) * ((x * x) * (x * x)))
```

？作るときは単純な関数が、なぜこんなに複雑になってしまったのか？

- ⇒ (木構造で表す) 式には、**代入・束縛がない** ので、共通のものを参照できない。
- ⇒ **アルゴリズムを記述する言語として、数式(木構造)は貧弱**

式の表現法を考える



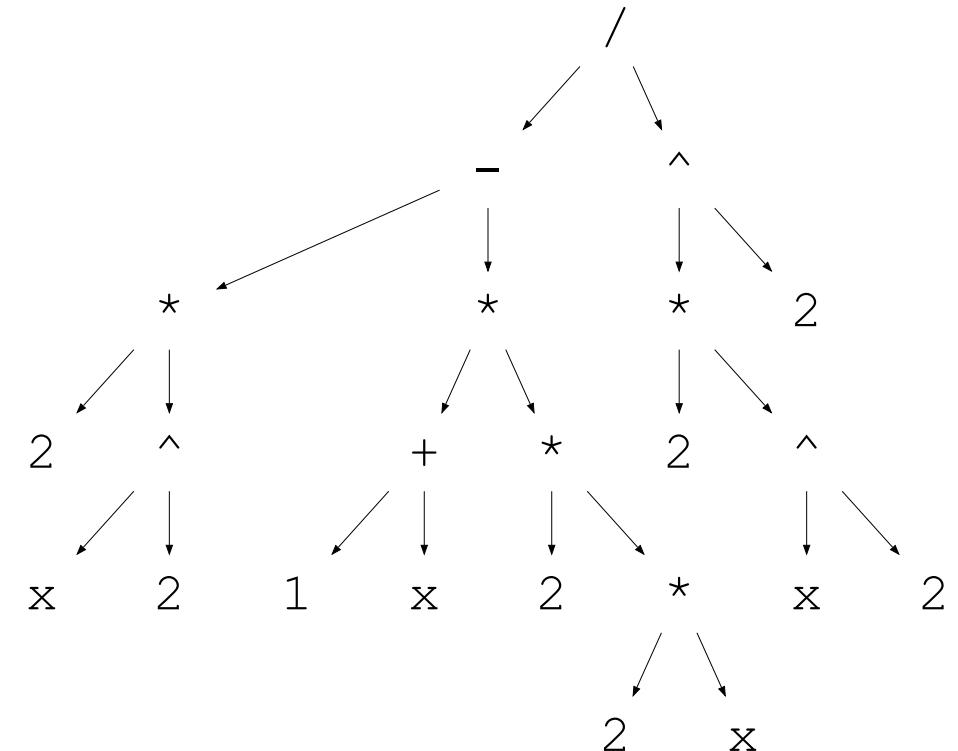
- ✖ 数式微分は微分すると式が肥大化してうまくいかない.
- 木で式を表現するのがそもそもうまくいかない 

式の表現法を考える

```
:((2 * x ^ 2 - (1 + x) * (2 *  
(2x))) / (2 * x ^ 2) ^ 2)
```

$$= \frac{(2 \cdot x^2 - (1 + x) \cdot 2 \cdot 2 \cdot x)}{(2 \cdot x^2)^2}$$

も、



式の表現法を考える

$$y_1 = x^2$$

$$y_2 = 2 \cdot y_1$$

$$y_3 = (1 + x)$$

$$y_4 = 2 \cdot x$$

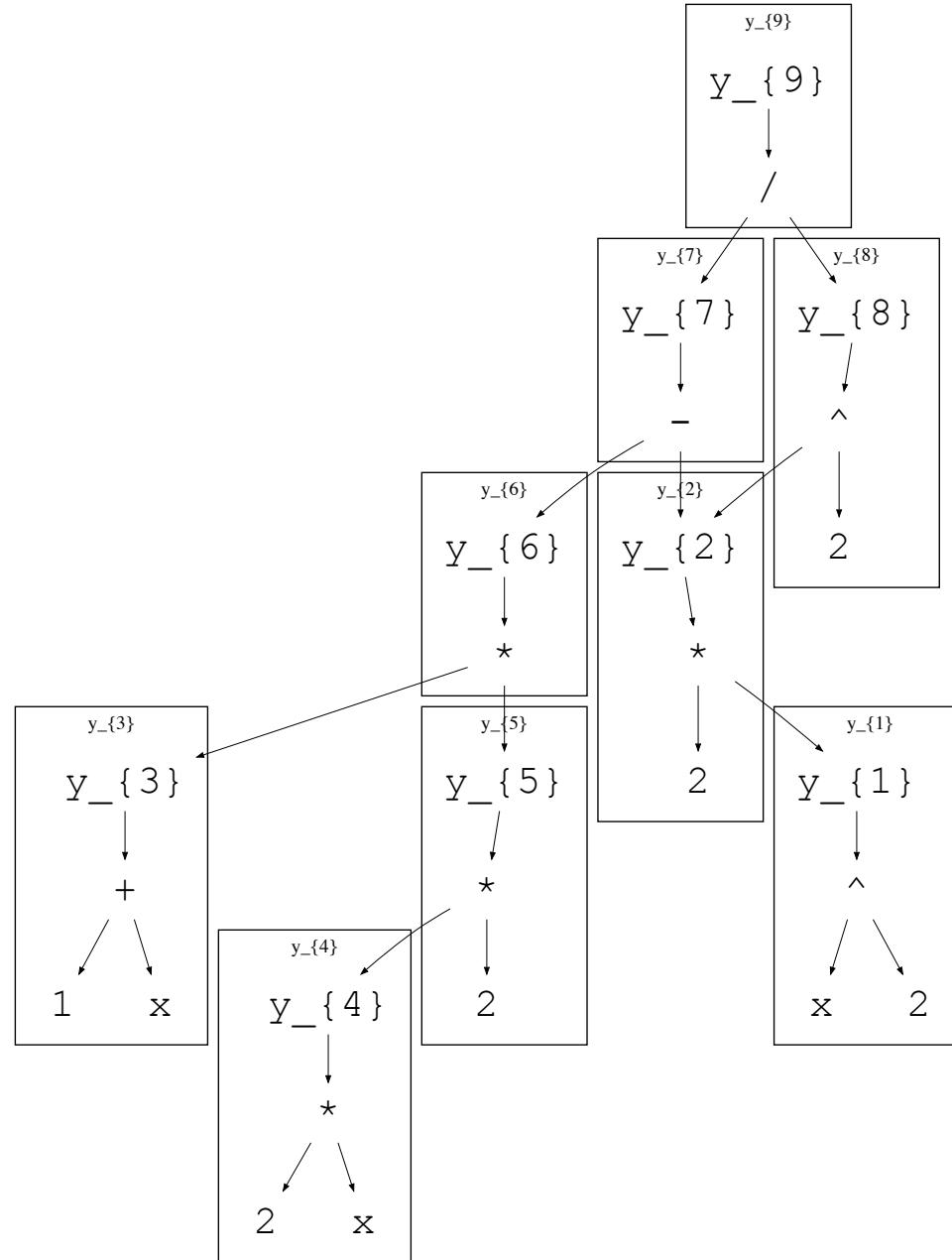
$$y_5 = 2 \cdot y_4$$

$$y_6 = y_3 \cdot y_5$$

$$y_7 = (y_2 - y_6)$$

$$y_8 = (y_2)^2$$

$$y_9 = \frac{y_7}{y_8}$$



```
x = [1, 2, 3]
y = [2, 4, 6]
```

```
function linear_regression_error(coef)
    pred = x * coef
    error = 0.
    for i in eachindex(y)
        error += (y[i] - pred[i])^2
    end
    return error
end
```

[需要]

制御構文・関数呼び出し etc...

一般的なプログラミング言語によって記述されたアルゴリズムに対しても、微分したい

$$f(x) = (2 - x)^2 + (4 - 2x)^2 + (6 - 3x)^2$$

木構造の式 から 木構造の式



(ふつうの) プログラム から プログラム へ

ヒューリスティックにやってそれなりに簡単な式を得られれば実用的には大丈夫なので与太話になりますが、簡約化を頑張れば最もシンプルな式を得られるか考えてみます。簡単さの定義にもよるかもしれません、 $\forall x$ で $f(x) = 0$ な f は $f(x) = 0$ と簡約化されるべきでしょう。

ところが、 f が四則演算と \exp, \sin, abs と有理数、 $\pi, \ln 2$ で作れる式のとき、 $\forall x, f(x) = 0$ か判定する問題は決定不能であることが知られています。(Richardson's theorem)
したがって、一般的の式を入力として、最も簡単な式を出力するようなアルゴリズムは存在しないとわかります。

2.3 自動微分 式からアルゴリズムへ

おさらい

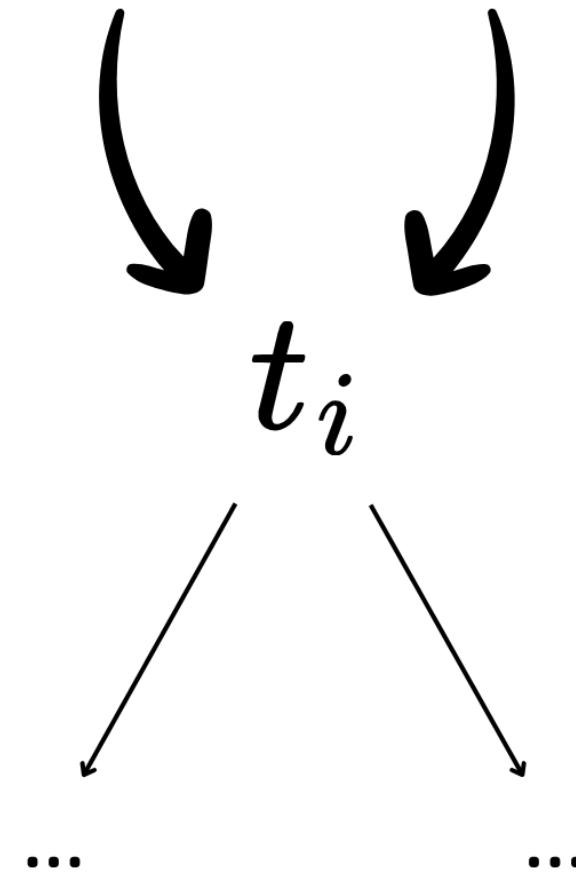
- 木構造で関数を表現しようとすると、簡単なものでも木が複雑になる.
- 原因は、木で表現された数式は束縛がないこと
- 束縛ができる（中間変数が導入された）場合にどうなるか考えてみる

DAG による表現

… あるものに名前をつけて
いくらでも参照できるようになった

$$t_1 = x * x$$

$$t_2 = t_1 * t_1$$

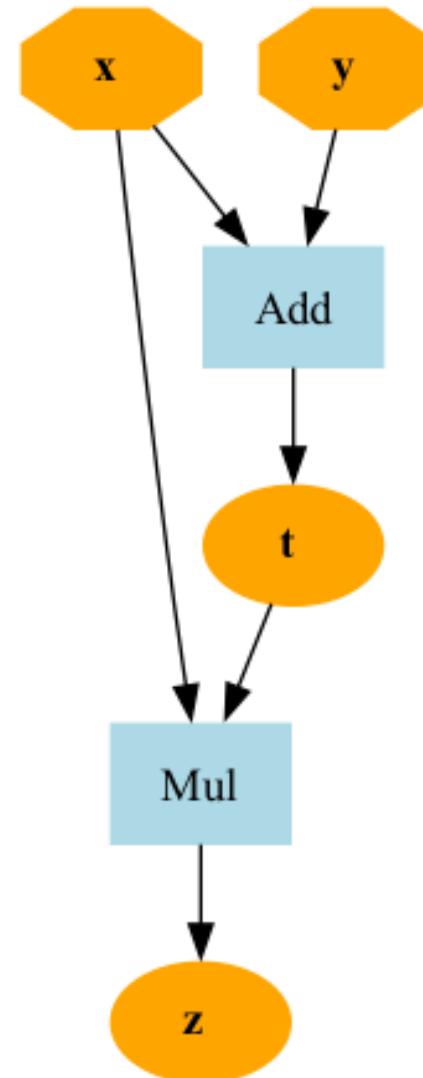


DAG による表現

有効非巡回グラフ(DAG)
でのアルゴリズムの表現

計算グラフ

(Kantrovich グラフ)



計算グラフによる表現

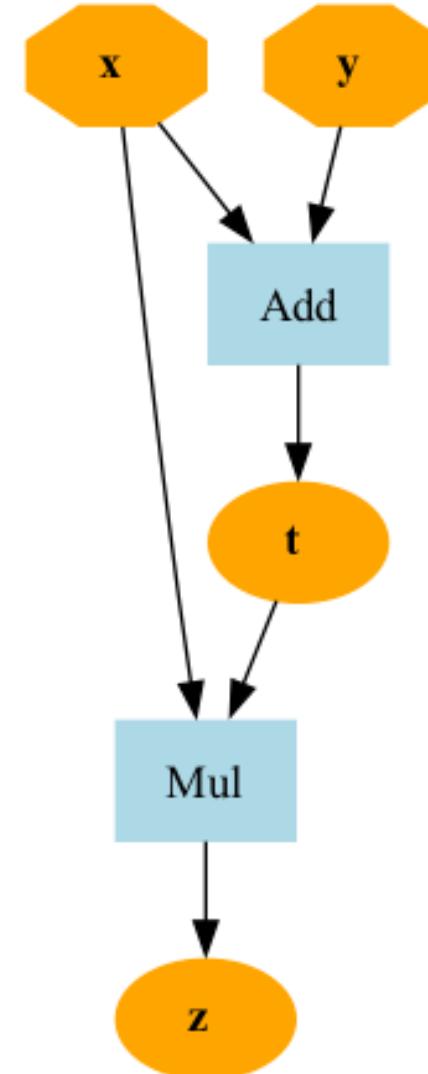
[計算グラフ]

計算過程をDAGで表現

$$t = x + y$$

$$z = x \cdot t$$

単に計算過程を表しただけのものを Kantrovich グラフなどと呼び、
これに偏導関数などの情報を加えたものを計算グラフと呼ぶような定義もあります。
(伊里, 久保田 (1998) に詳しく形式的な定義があります)
ただ、単に計算グラフというだけで計算過程を表現するグラフを指すという用法はかなり普及していて一般的と思われます。そのためここでもそれに従って計算過程を表現するグラフを計算グラフと呼びます。



計算グラフによる表現

2.3 自動微分 一式からアルゴリズムへ

(一旦計算グラフを得たものとして、)
この構造から導関数を得ることを考えてみる。

[連鎖律]

u, v の関数 x, y による合成関数 $z(x(u, v), y(u, v))$ に対して、

$$\frac{\partial z}{\partial u} = \frac{\partial z}{\partial x} \cdot \frac{\partial x}{\partial u} + \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial u}$$

$$\frac{\partial z}{\partial v} = \frac{\partial z}{\partial x} \cdot \frac{\partial x}{\partial v} + \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial v}$$

連鎖律と計算グラフの対応

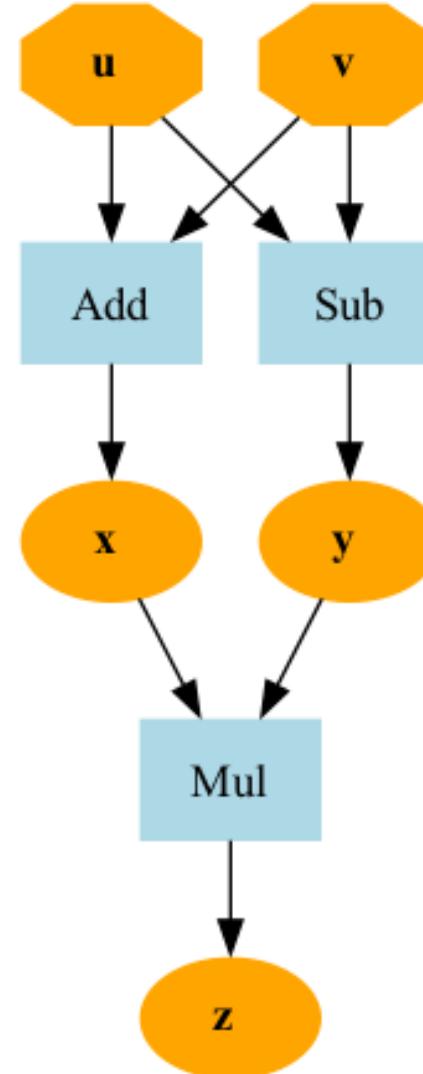
目標

$$x = u + v$$

$$y = u - v$$

$$z = x \cdot y$$

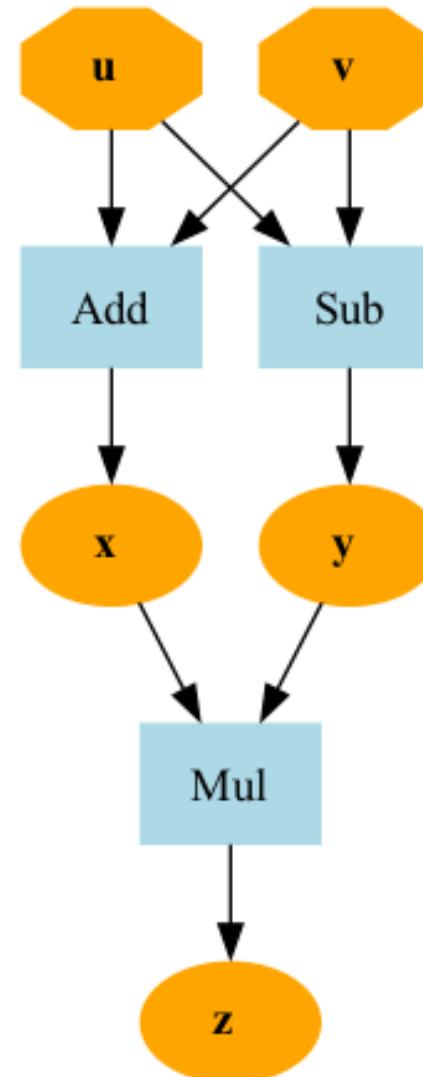
のとき、 $\frac{\partial z}{\partial u}$ を求める



連鎖律と計算グラフの対応

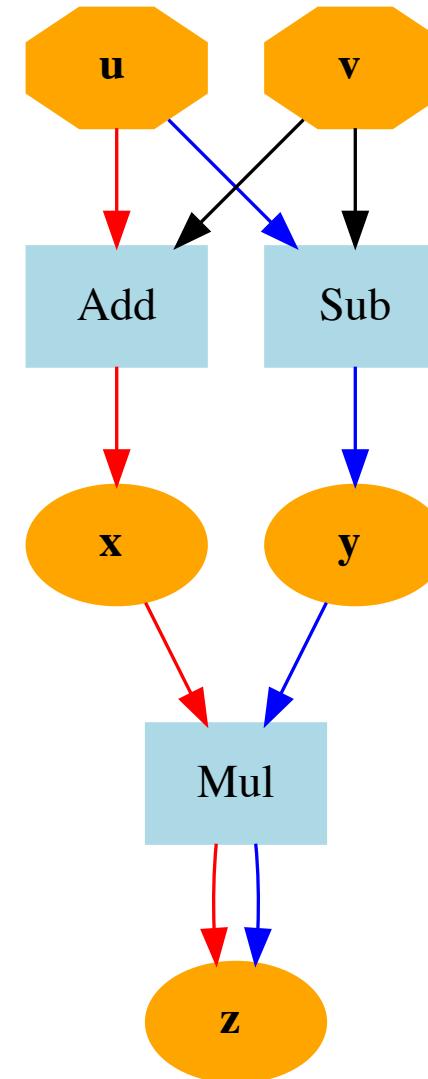
$$\frac{\partial z}{\partial u} = \frac{\partial z}{\partial x} \cdot \frac{\partial x}{\partial u} + \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial u}$$

との対応は、



連鎖律と計算グラフの対応

$$\frac{\partial z}{\partial u} = \frac{\partial z}{\partial x} \cdot \frac{\partial x}{\partial u} + \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial u}$$



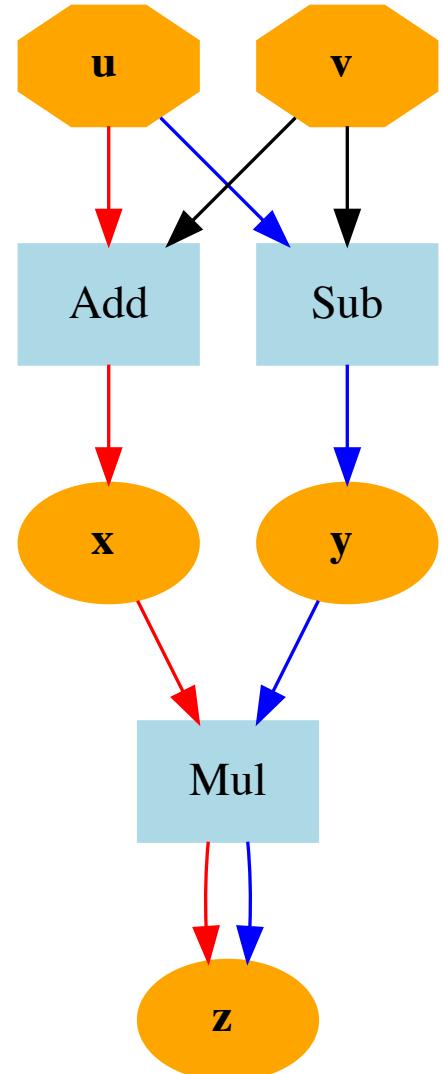
連鎖律と計算グラフの対応

✓ 変数 z に対する u による偏微分の
計算グラフ上の表現

↔ u から z への全ての経路の偏微分の総積の総和

$$\frac{\partial z}{\partial u} = \sum_{p \in \hat{P}(u, z)} \left(\prod_{(s, t) \in p} \frac{\partial t}{\partial s} \right)$$

$\hat{P}(u, z)$ は u から z への全ての経路の集合. (s, t) は変数 s から変数 t への辺を表す.



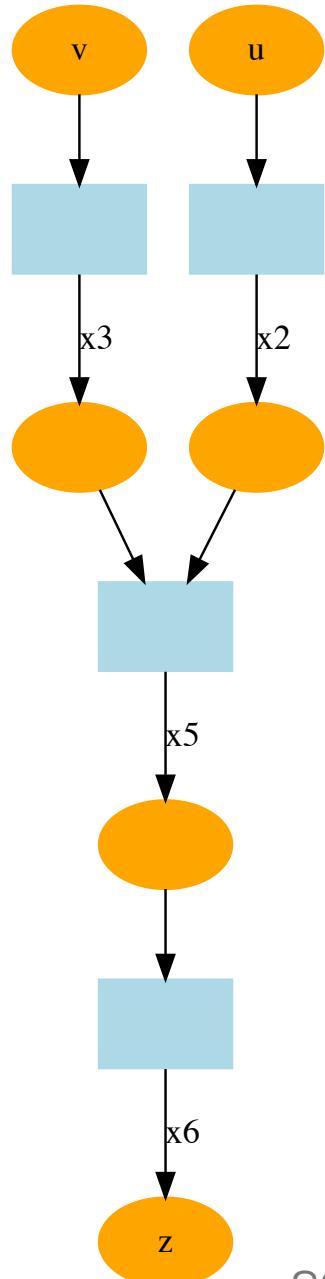
キャッシュ

一番簡単なやりかた

$\frac{\partial z}{\partial u}$ を求める:

```
graph = ComputationalGraph(f)

∂z_∂u = 0
for path in all_paths(graph, u, z)
    ∂z_∂u += prod(grad(s, t) for (s, t) in path)
end
```

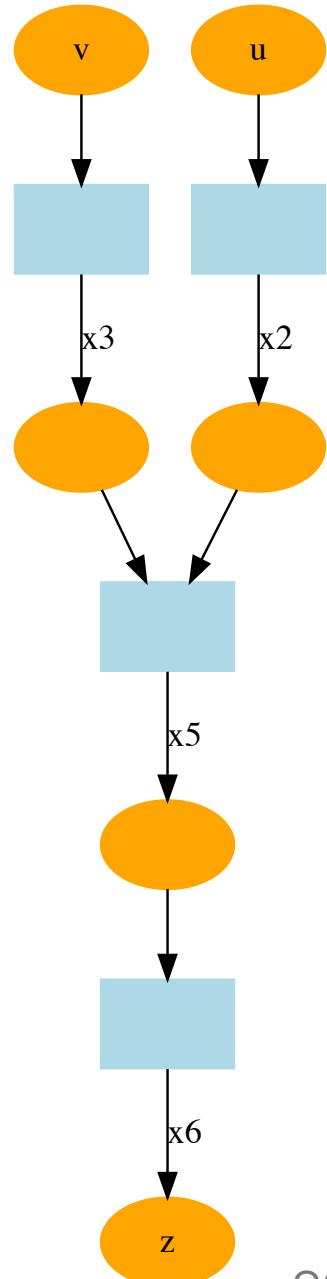


キャッシュ

続いて $\frac{\partial z}{\partial v}$ を求める:

$$\partial z / \partial v = 0$$

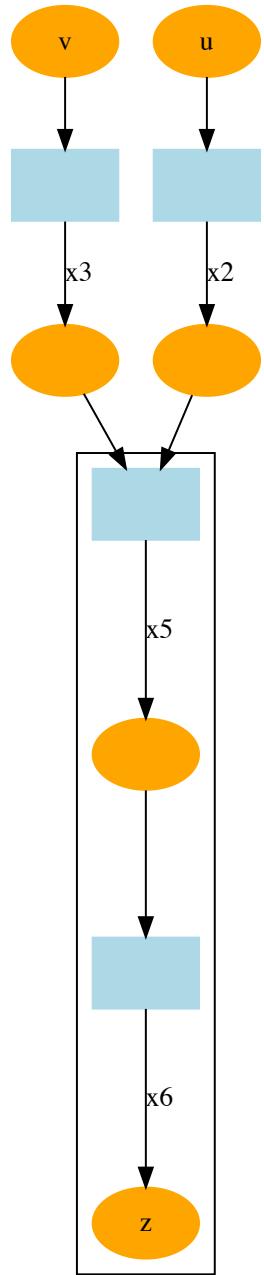
```
for path in all_paths(graph, v, z)
    ∂z_∂v += prod(grad(s, t) for (s, t) in path)
end
```



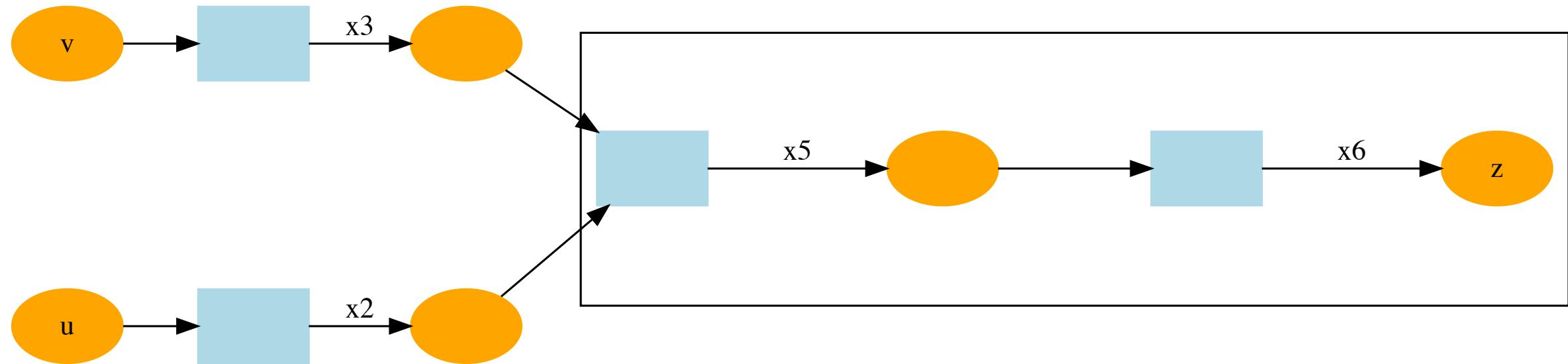
自動微分とキャッシュ

共通部がある！ \leftrightarrow 独立して計算するのは非効率.

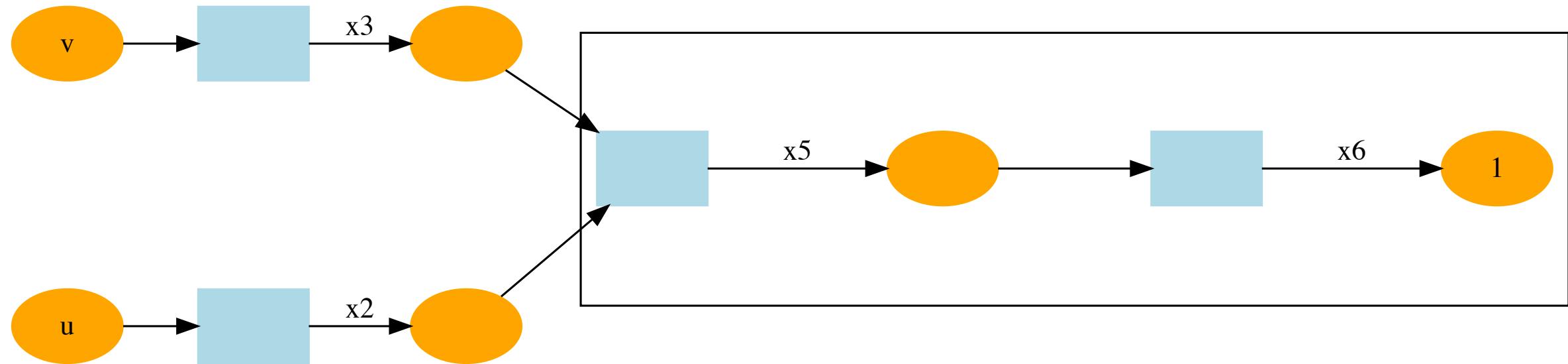
\Rightarrow うまく複数のノードからの経路を計算する.



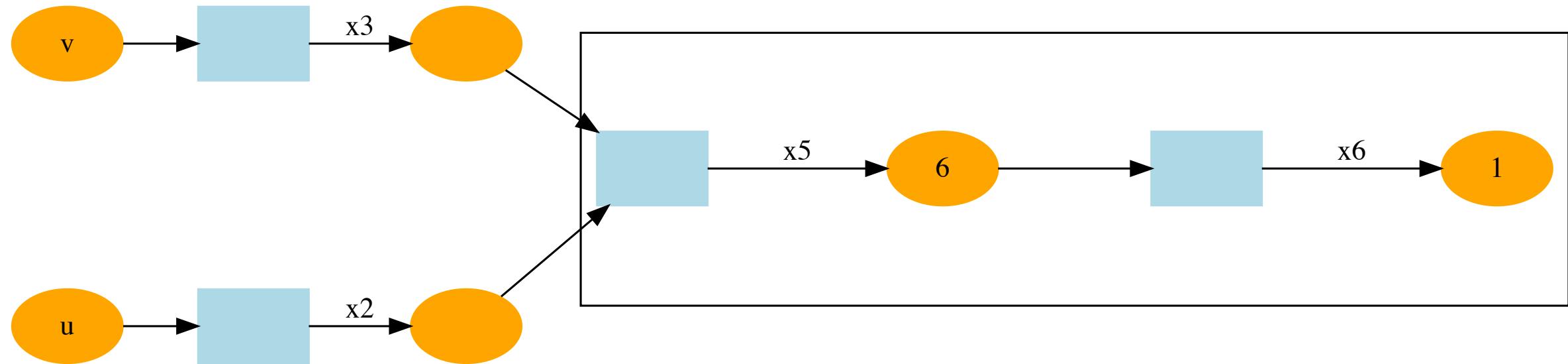
Backward-Mode AD



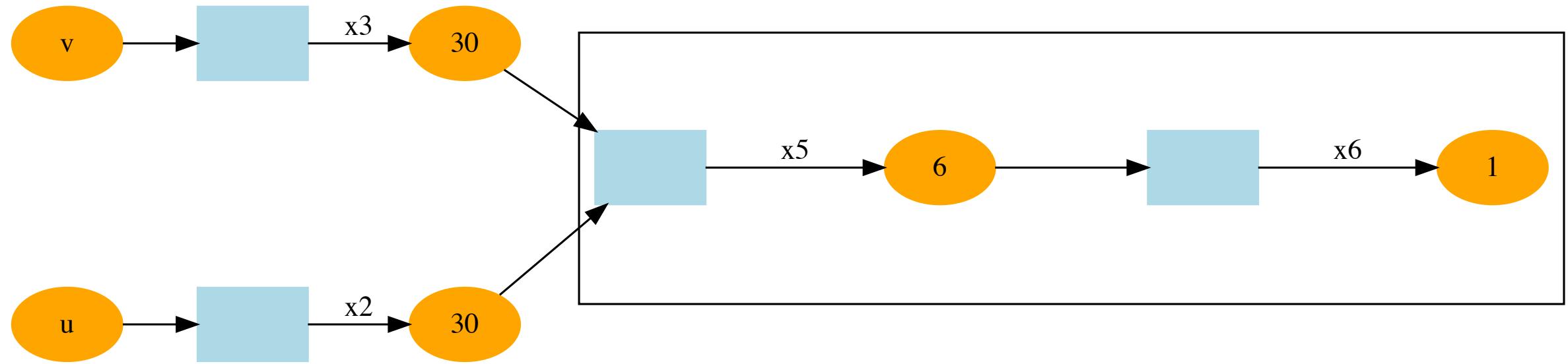
Backward-Mode AD



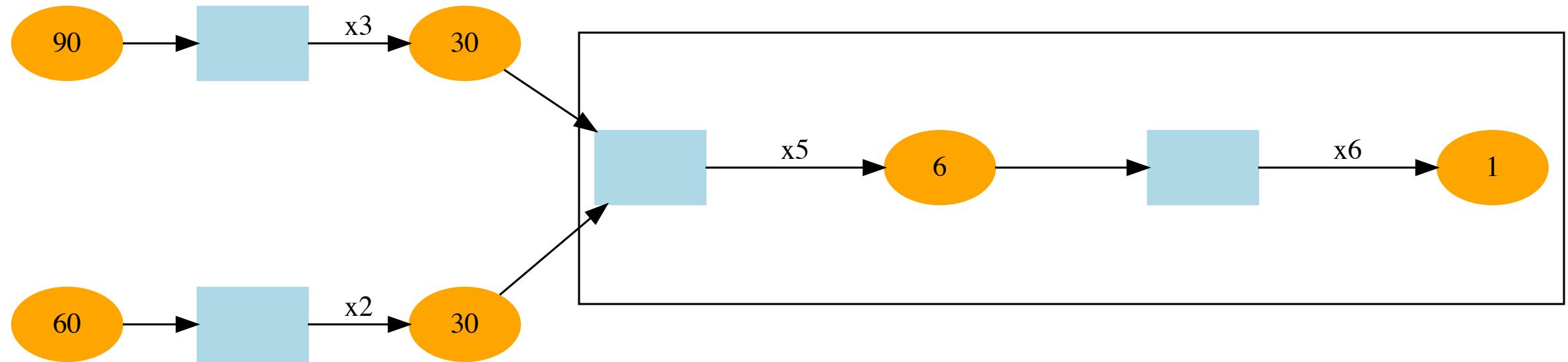
Backward-Mode AD



Backward-Mode AD



Backward-Mode AD



✓ 計算グラフを辿っていくことで、共通部の計算を共有しながら、
「複数の偏微分係数をグラフ一回の走査で」
「中間変数の偏微分係数を共有しながら」 計算できた！

この微分のアルゴリズムを

後退型自動微分 (Backward-Mode AD)、 高速微分(fast differentiation)、
逆向き自動微分(Reverse-Mode AD)、 高速自動微分(fast AD) などと呼ぶ。

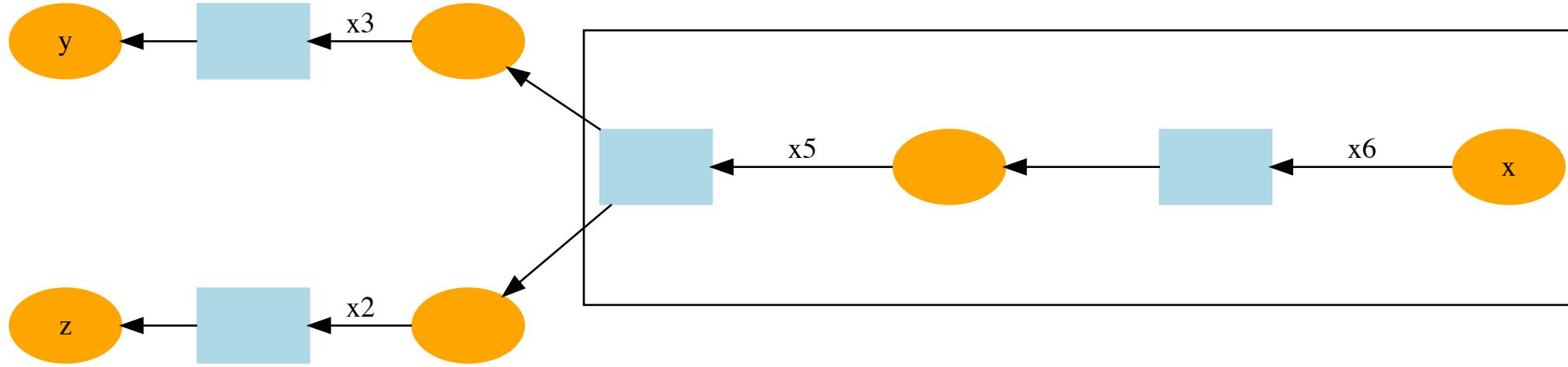
？一般の $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ について、常に逆向きに微分を辿るのがよい？



$m > n$ の場合を考えてみる

Forward-Mode AD

2.3 自動微分 一式からアルゴリズムへ



... x から辿っていくことで、**共通部を共有しつつ、複数の出力に対する偏微分係数を一度に計算できる**

⇒ 前進型自動微分 (Forward-Mode AD)

- 逆向き自動微分 (Backward-Mode AD)

- $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ に対して、 $m < n$ の場合に効率的
- 勾配を一回グラフを走査するだけで計算可能

- 前進型自動微分 (Forward-Mode AD)

- $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ に対して、 $m > n$ の場合に効率的
- ヤコビ行列の一列を一回グラフを走査するだけで計算可能
- 実装では定数倍が軽くなりがちなため、
 n, m が小さい場合には効率的な可能性が高い

時間がないため割愛しましたが、Forward-Mode AD の話でよく出てくる **二重数** というものがあります。 $\varepsilon^2 = 0$ かつ $\varepsilon \neq 0$ なる $\varepsilon \notin \mathbb{R}$ を考え、これと実数からなる集合上の演算を素直に定義すると一見、不思議なことに微分が計算される... というような面白い数です。興味があるかたは 「2乗してはじめて0になる数」とかあったら面白くないですか？ですよね - アジマティクス や ForwardDiff.jlのドキュメント などおすすめです。

✓ 計算グラフは DAG

↔ トポロジカルソート可能

演算の適用順序を適切に持てば、
単に演算の列を持って同様に
グラフを辿るのと同じことが可能。

この列を、

Wengert List, Gradient Tape
と呼ぶ。

$$y_1 = x^2$$

$$y_2 = 2 \cdot y_1$$

$$y_3 = (1 + x)$$

$$y_4 = 2 \cdot x$$

$$y_5 = 2 \cdot y_4$$

$$y_6 = y_3 \cdot y_5$$

$$y_7 = (y_2 - y_6)$$

$$y_8 = (y_2)^2$$

$$y_9 = \frac{y_7}{y_8}$$

- PyTorch / Chainer は Wengert List ではなく計算グラフを使っている. [1]

No tape. Traditional reverse-mode differentiation records a tape (also known as a Wengert list) describing the order in which operations were originally executed; <中略>

An added benefit of structuring graphs this way is that when a portion of the graph becomes dead, it is automatically freed; an important consideration when we want to free large memory chunks as quickly as possible.

- Zygote.jl, Tensorflow などは Wengert List を使っている.

[1] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. & Lerer, A. (2017). Automatic Differentiation in PyTorch. NIPS 2017 Workshop on Autodiff, .

[2] 計算グラフとメモリの解放周辺で、Chainer の Aggressive Buffer Release という仕組みがとても面白いです: [Aggressive buffer release #2368](#)

計算グラフをどう得るか？

2.3 自動微分 一式からアルゴリズムへ

 < 計算グラフさえあれば計算ができることがわかった。
では計算グラフをどう得るか？

計算グラフをどう得るか？

2.3 自動微分 一式からアルゴリズムへ

- ✓ 一般的なプログラムを **直接解析** して
(微分が計算できる) 計算グラフを得る
プログラムを実装するのはとても難易度
が高い.



```
x = [1, 2, 3]
y = [2, 4, 6]
```

```
function linear_regression_error(coef)
    pred = x * coef
    error = 0.
    for i in eachindex(y)
        error += (y[i] - pred[i])^2
    end
    return error
end
```

トレースによる計算グラフの獲得

2.3 自動微分 一式からアルゴリズムへ

- ✓ トレース … 実際にプログラムを実行し、その過程を記録することで計算グラフを得る

[典型的なトレースの実装]

1. `Variable` 型を用意
2. 基本的な演算を表す関数について、`Variable` 型用のメソッドを実装し、このメソッドの中で計算グラフも構築する

トレースの OO による典型的な実装

2.3 自動微分 一式からアルゴリズムへ

```
import Base

mutable struct Scalar
    values
    creator
    grad
    generation
    name
end

Base.:+(x1::Scalar, x2::Scalar) = calc_and_build_graph(+, x1, x2)
Base.:(*)(x1::Scalar, x2::Scalar) = calc_and_build_graph(*, x1, x2)
...
```

トレースの利点

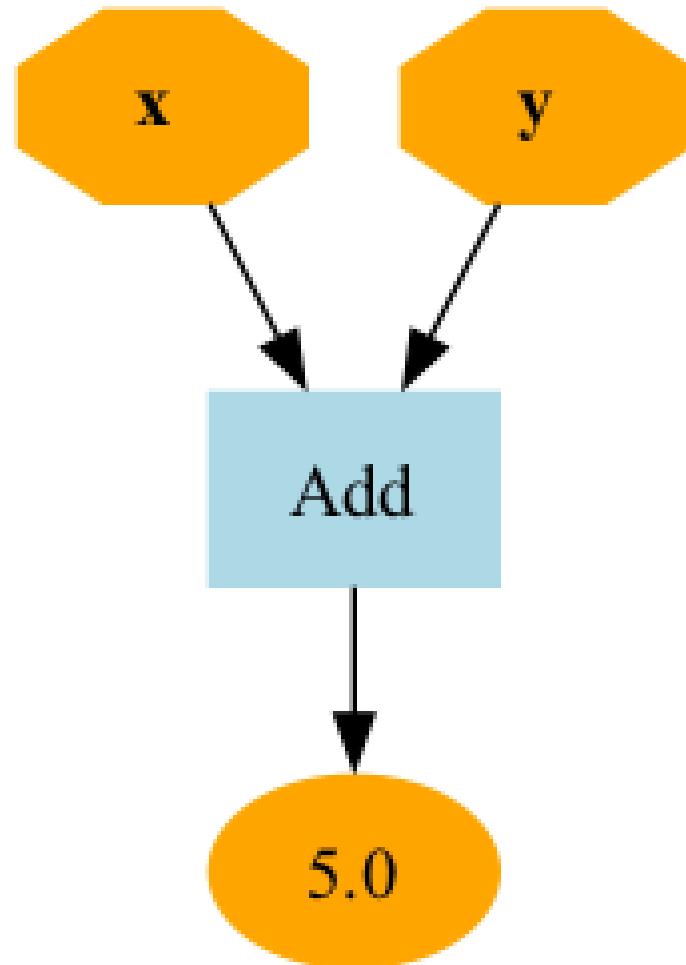
「実際そのときあった演算」のみが記録され問題になる
⇒ 制御構文がいくらあってもOK

```
function crazy_function(x, y)
    rand() < 0.5 ? x + y : x - y
end

x = Scalar(2.0, name="x")
y = Scalar(3.0, name="y")

z = crazy_function(x, y)

JITrench.plot_graph(z, var_label=:name)
```



- 計算時にグラフを作ることによるオーバーヘッド
- コンパイラの最適化の情報が消えてしまい恩恵をうけにくい

- コンパイラと深く関わったレベルで自動微分をやっていこう！



Zygote.jl, Enzyme, Swift for Tensorflow etc...



[3] Juila に微分させる

3.1 FiniteDiff.jl/FiniteDifferences.jl

- どちらも数値微分のパッケージ
- 概ね機能は同じだが、スペースなヤコビ行列を求めたいときやメモリのアロケーションを気にしたいときは FiniteDiff.jl を使うといいかもしれない
- 詳しい比較は <https://github.com/JuliaDiff/FiniteDifferences.jl>

```
julia> using FiniteDifferences  
  
julia> central_fdm(5, 1)(sin, π / 3)  
0.4999999999999536
```

ForwardDiff.jl

ForwardDiff.jl

- Forward-Mode の自動微分を実装したパッケージ
- 小規模な関数の微分を行う場合は高速ことが多い

```
julia> using ForwardDiff

julia> f(x) = x^2 + 4x
f (generic function with 1 method)

julia> df(x) = ForwardDiff.derivative(f, x) # 2x + 4
df (generic function with 1 method)

julia> df(2)
8
```

Zygote.jl

- コンパイラと深く関わったレベルで自動微分をやっていこう！



Zygote.jl, Enzyme, Swift for Tensorflow etc...



Zygote.jl

- ✓ ソースコード変換ベースのAD
- ✓ JuliaのコードをSSA形式のIRに変換して導関数を計算するコード(Adjoint Code) を生成

```
julia> f(x) = 3x^2
f (generic function with 1 method)

julia> Zygote.@code_ir f(0.)
1: (%1, %2)
%3 = Main.:^
%4 = Core.apply_type(Base.Val, 2)
%5 = (%4)()
%6 = Base.literal_pow(%3, %2, %5)
%7 = 3 * %6
return %7
```

Zygote.jl

- Julia のコンパイラと連携・最適なコードを生成
- SSA形式からの最適化 ... という方向性はJuliaに限らない
⇒ Enzyme, Diffractor.jl...

その他のパッケージ



- Enzyme
 - LLVMのレイヤーで自動微分を行う.
 - Julia をはじめ、 LLVMを中間表現に使っている色々な言語で動作させられる
- Diffractor.jl
 - より進んで Juliaの型推論を活用し効率的なコードを生成を目指す
 - まだ experimental だが Keno さんが開発中で期待大！

まとめ

- 微分をするアルゴリズムはさまざまあり、それぞれ特徴があった
 - 数値微分... 容易に実装可能
 - 自動微分... 高速で精度よく計算できる
- 状況(入出力変数の数, 時間 etc...) に応じて適切なアルゴリズムを選ぶのが大事！
- メタプログラミングやコンパイラのあれこれに介入しやすいJuliaが楽しい！
- 自動微分は奥が深い！

おまけ

数値微分

```
function numerical_derivative(f::Function, x::Number)::Number
    g = numerical_operation(f, x)
    return g
end
```

数値微分のアイデア

微分の定義

$$\lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

をそのまま近似する

数值微分の実装

```
function numerical_derivative(f, x; h=1e-8)
    g = (f(x+h) - f(x)) / h
    return g
end

f(x) = sin(x)
f'(x) = cos(x)
x = π / 3

numerical_derivative(f, x) # 0.4999999969612645
f'(x)                      # 0.5000000000000001
```

数値微分のデメリット

1. 打ち切り誤差が生じる
2. 衍落ちも起こる
3. 計算コストが高い

数值微分の誤差 ~ 打ち切り誤差

本来は極限を取るのに小さい値で
誤魔化すので誤差が発生



実際どれくらいの誤差が発生する？

$$\lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

数值微分の誤差

[定理2. 数値微分の誤差]

$$f'(x) - \frac{f(x+h) - f(x)}{h} = O(h)$$

[証明]

テイラー展開すると、

$$\begin{aligned} f'(x) - \frac{1}{h}(f(x+h) - f(x)) &= f'(x) - \frac{1}{h}(f(x) + f'(x)h + O(h^2) - f(x)) \\ &= O(h) \end{aligned}$$

誤差の最小化

実験:

$O(h)$ なら、 h をどんどん小さくすればいくらでも精度が良くなるはず？

```
H = [0.1^i for i in 4:0.5:10]
E = similar(H)

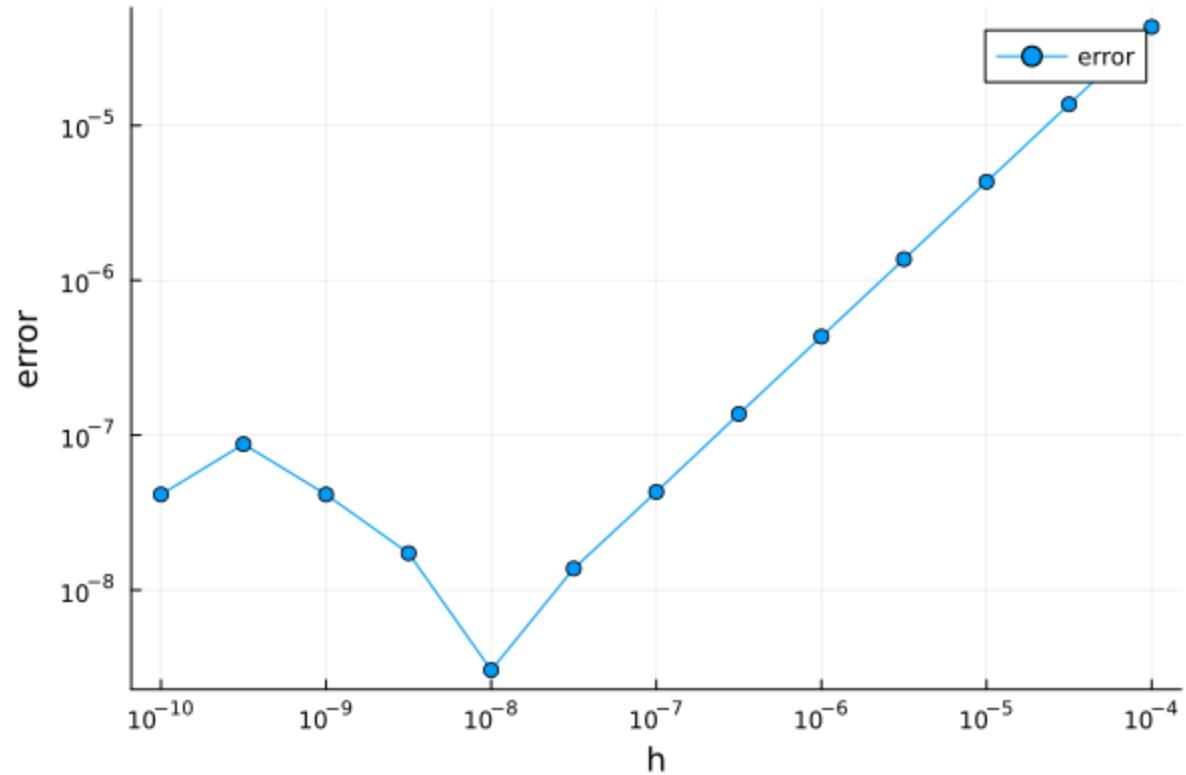
for i in eachindex(H)
    d = numerical_derivative(f, x, h=H[i])
    E[i] = abs(d - f'(x))
end

plot(H, E)
```

誤差の最小化

実際

$h < 10^{-8}$ くらいになるとむしろ精度が悪化する



丸め誤差と打ち切り誤差のトレードオフ

h が小さくなると、分子の引き算が非常に近い値の引き算になる

⇒ 衍落ちが発生し全体として悪化

$$\frac{f(x + h) - f(x)}{h}$$

数值微分の改良

誤差への対応

1. 打ち切り誤差 \Rightarrow 計算式の変更
2. 衍落ち $\Rightarrow h$ の調整？

数値微分の改良 ~ 打ち切り誤差の改善

1. 打ち切り誤差への対応

微分の(一般的な)定義をそのまま計算する方法:

$$f'(x) \approx \frac{f(x + h) - f(x)}{h}$$

は **前進差分** と呼ばれる

数値微分の改良 ~ 打ち切り誤差の改善

ところで、

$$\lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x - h)}{2h}$$



これを近似してみても良さそう？

中心差分による2次精度の数値微分

実はこれの方が精度がよい！

[定理3. 中心差分の誤差]

$$f'(x) - \frac{f(x+h) - f(x-h)}{2h} = O(h^2)$$

同じようにテイラー展開をするとわかる

また、簡単な計算で一般の n について n 点評価で $O(h^n)$ の近似式を得られる

中心差分と同様に x から左右に $\frac{n}{2}$ 個ずつ点とってこれらの評価の重みつき和を考えてみます。

すると、テイラー展開の各項を足し合わせて $f'(x)$ 以外の係数を 0 にすることを考えることで公比が各列 $-\frac{n}{2}, -\frac{n-1}{2}, \dots, \frac{n}{2}$ で初項 1 のヴァンデルモンド行列を A として $Ax = e_2$ を満たす x を各成分 h で割ったのが求めたい重みとわかります。あとはこの重み付き和をとればいいです。同様にして k 階微分の近似式も得られます。

数值微分の改良 ~ 衍落ちへの対応

2. 衍落ちへの対応

Q. 打ち切り誤差と丸め誤差のトレードオフで h を小さくすればいいというものじゃないことはわかった。じゃあ、最適な h は見積もれる？

A. 最適な h は f の n 階微分の大きさに依存するから簡単ではない。

例) 中心差分 $\frac{f(x+h) - f(x-h)}{h}$ は $h_{best} \approx \sqrt[3]{\frac{3\sqrt{2}\varepsilon}{|f'''(x)|}}$ くらい？

⇒ $f'(x)$ がわからないのに $f'''(x)$ を使った式を使うのは現実的でない。

しょうがないので $h = \left(\frac{(n+1)!}{\sqrt{n}f(x)}\varepsilon\right)^{\frac{1}{n+1}}$ に線を引いてみると…

[導出?]

n 点評価で $O(h^n)$ の近似をしたときの誤差の期待値を最小化する.

前のページで導出した \mathbf{x} を使うと $f_{estimate}(x) = \frac{1}{h} \sum_i^n x_i \hat{f}(x_i)$
 (\hat{f} は計算誤差を含む f の計算結果.)

ここで各 $\hat{f}(x_i)$ の計算結果が ε の誤差を生むとすると

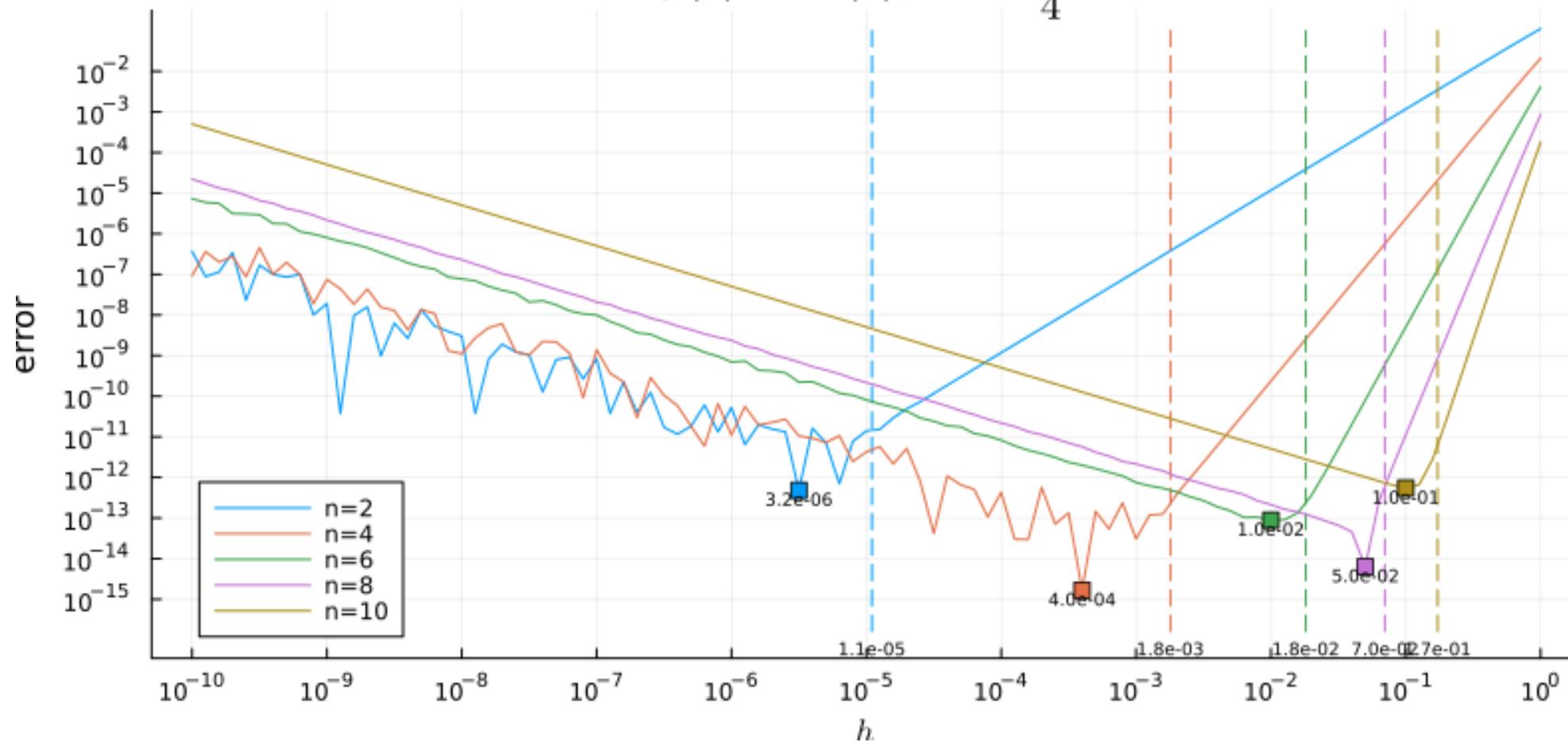
分子の誤差の期待値はランダムウォークの期待値を考えて $\sqrt{n}\varepsilon$.

すると $f'(x)$ との誤差の期待値 $E(h)$ は

$$E(h) \approx \sqrt{n}\varepsilon + \frac{nf^{(n+1)}(x)}{(n+1)!} h^{n-1} \quad (\because \text{テイラーの定理})$$

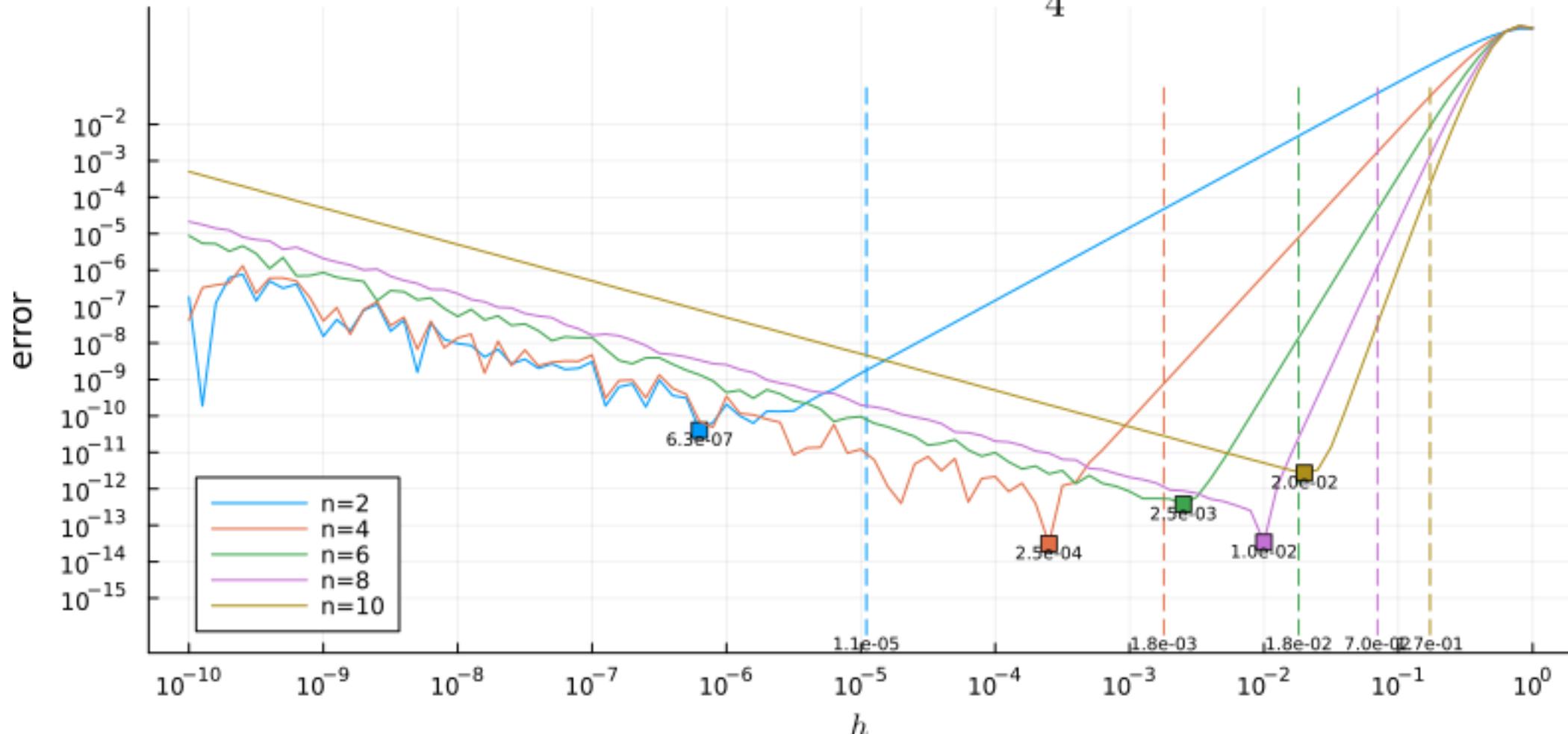
$$\text{この最小値を計算すると } h = \left(\frac{(n+1)!\varepsilon}{\sqrt{n}f^{(n+1)}(x)} \right)^{\frac{1}{n+1}}.$$

$$f(x) = \sin(x), \quad x = \frac{\pi}{4}$$



何回微分しても大きさが変わらないウルトラお行儀が良い関数.

$$f(x) = \sin(5x), \quad x = \frac{\pi}{4}$$



(微分するたび 5 が外に出る)

デメリット3. 計算コストが高い

多変数関数への拡張

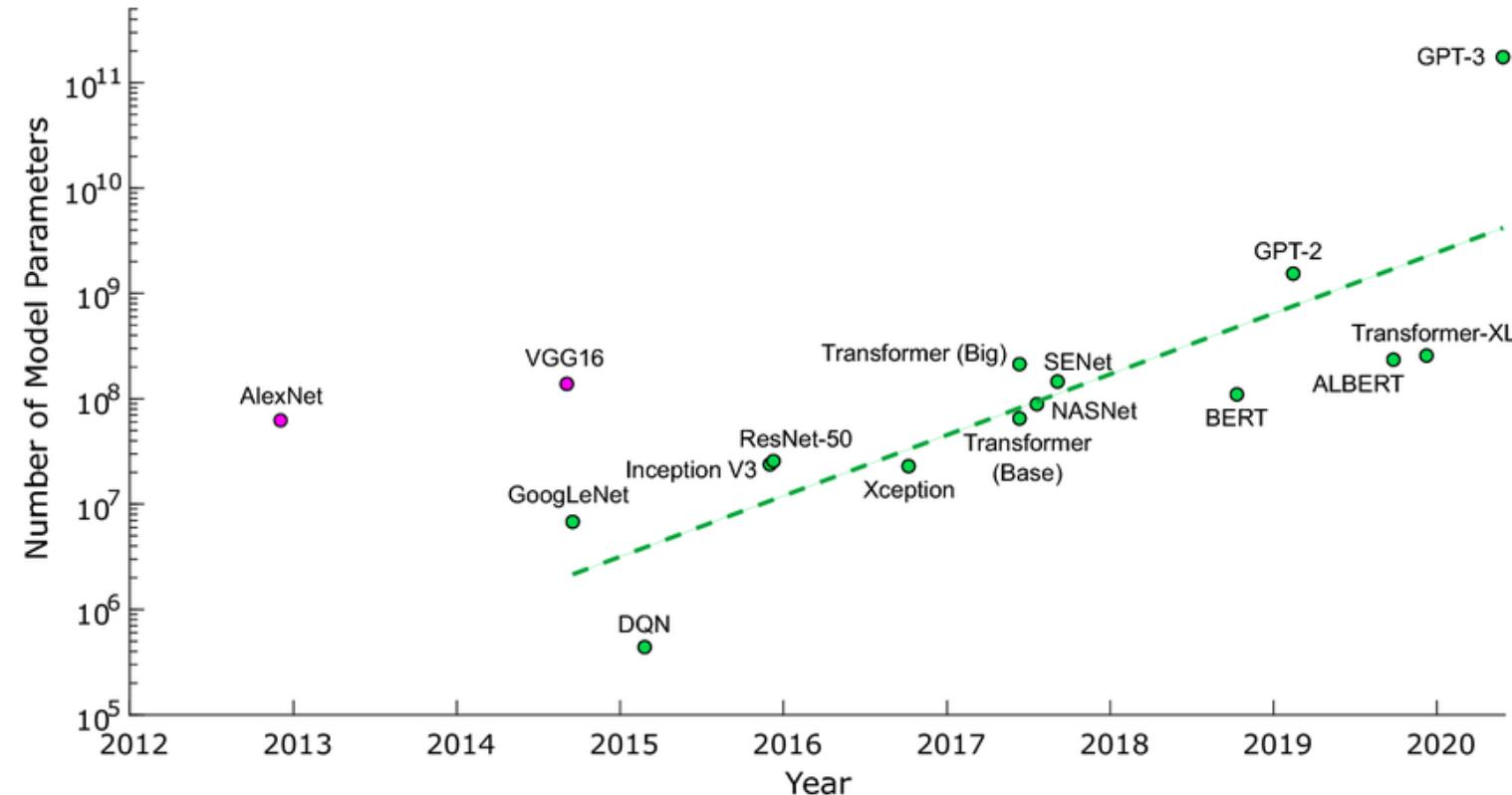
$f : \mathbb{R}^n \rightarrow \mathbb{R}$ の x における勾配ベクトル $\nabla f(x)$ を求める

```
function numerical_gradient(f, x::Vector; h=1e-8)
    n = length(x)
    g = zeros(n)
    y = f(x...)
    for i in 1:n
        x[i] += h
        g[i] = (f(x...) - y) / h
        x[i] -= h
    end
    return g
end
```

⇒ f を n 回評価する必要がある.

多変数関数への拡張

✓ 応用では f が重く, n が大きくなりがち $\Rightarrow n$ 回評価は高コスト



結論: 数値微分を機械学習などで使うのは難しそう.

一方、

- f に特別な準備なくなんでも計算できる
- 実装が容易でバグが混入しにくい

ため、他の微分アルゴリズムのテストに使われることが多い.

自動微分の勉強で参考になる文献

1. 久保田光一, 伊里正夫 「アルゴリズムの自動微分と応用」 コロナ社 (1998)
 - i. 自動微分そのものについて扱ったおそらく唯一の和書です。 詳しいです。
 - ii. 形式的な定義から、計算グラフの縮小のアルゴリズムや実装例と基礎から実用まで触れられています。
 - iii. サンプルコードは、FORTRAN, (昔の) C++ です。 😊
2. 斎藤康毅 「ゼロから作るDeep Learning ③」 O'Reilly Japan (2020)
 - i. トレースベースの Reverse AD を Python で実装します。
 - ii. Step by step で丁寧に進んでいくので、とてもおすすめです。
 - iii. 自動微分自体について扱った本ではないため、その辺りの説明は若干手薄かもしれません。
3. Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2015). Automatic differentiation in machine learning: A survey. ArXiv. /abs/1502.05767
 - i. 機械学習 x AD のサーベイですが、機械学習に限らず AD の歴史やトピックを広く取り上げてます。
 - ii. 少し内容が古くなっているかもしれません。
4. [Differentiation for Hackers](#)
 - i. Flux.jl や Zygote.jl の開発をしている Mike J Innes さんが書いた自動微分の解説です。 Juliaで動かしながら勉強できます。おすすめです。
5. Innes, M. (2018). Don't Unroll Adjoint: Differentiating SSA-Form Programs. ArXiv. /abs/1810.07951
 - i. Zygote.jl の論文です。かなりわかりやすいです。
6. Gebremedhin, A. H., & Walther, A. (2019). An introduction to algorithmic differentiation. Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 10(1), e1334.
<https://doi.org/10.1002/widm.1334>
 - i. 実装のパラダイムやCheckpoint, 並列化などかなり広く触れられています
7. [Zygote.jl のドキュメントの用語集](#)
 - i. 自動微分は必要になった応用の人がやったり、コンパイラの人がやったり、数学の人がやったりで用語が乱立しまくっているのでこちらを参照して整理すると良いです
8. [JuliaDiff](#)
 - i. Julia での微分についてまとまっています。
9. [Chainer のソースコード](#)
 - i. Chainer は Python 製の深層学習フレームワークですが、既存の巨大フレームワークと比較すると、裏も Python でとても読みやすいです。
 - ii. 気になる実装があったら当たるのがおすすめです。議論もたくさん残っているのでそれを巡回するだけでとても勉強になります。