浮動小数点数

荒田 実樹

2017 年 6 月 22 日

目次

数の表現：（固定長）整数

16 ビット符号なし整数

- $3 = [\mathtt{0003}]_{16} = [\mathtt{0000\ \ 0000\ \ 0000\ \ 0011}]_2$
- $2017 = [\mathtt{07E1}]_{16} = [\mathtt{0000\ \ 0111\ \ 1110\ \ 0001}]_2$
- 表せる最大の整数：$2^{16} - 1 = 65535 = [\mathrm{FFFF}]_{16} = [\mathtt{1111\ \ 1111\ \ 1111\ \ 1111}]_2$

16 ビット符号あり整数（2 の補数表現）

- $-1 = [\mathrm{FFFF}]_{16} = [\mathtt{1111\ \ 1111\ \ 1111\ \ 1111}]_2$
- $-2 = [\mathrm{FFFE}]_{16} = [\mathtt{1111\ \ 1111\ \ 1111\ \ 1110}]_2$
- 表せる最小の整数：$-2^{15} = [\mathtt{8000}]_{16} = [\mathtt{1000\ \ 0000\ \ 0000\ \ 0000}]_2$
- 表せる最大の整数：$2^{15} - 1 = [\mathtt{7FFF}]_{16} = [\mathtt{0111\ \ 1111\ \ 1111\ \ 1111}]_2$

32 ビットや 64 ビットの整数が使われることが多い

数の表現：固定小数点数

小数点の位置を決め、小数点以下 N 桁を表せるようにする。

例：小数点以下（2 進で）8 桁

- $0.5 = 128 \times 2^{-8} = [\texttt{00.80}]_{16} = [\texttt{0000 0000.1000 0000}]_2$
- $1.25 = 320 \times 2^{-8} = [\texttt{01.40}]_{16} = [\texttt{0000 0001.0100 0000}]_2$
- $0.1 = 25.6 \times 2^{-8} \overset{\text{丸め}}{\rightsquigarrow} 26 \times 2^{-8} = [\texttt{00.1A}]_{16} = [\texttt{0000 0000.0001 1010}]_2$

固定小数点数の演算

- 足し算、引き算は整数と同じように計算できる
- かけ算は、整数の掛け算とビットシフトを組み合わせる
- 整数演算を流用できる

数の表現：浮動小数点数

精度 p と底 b （2 か 10 のことが多い）を決め、数を

$$[X_0.X_1X_2\dots X_{p-1}]_b \times b^e \quad (X_0 \neq 0)$$

の形で表し、仮数部 X_i と指数部 e をそれぞれ保存する。

例：2 進小数の場合

$$[1.X_1X_2X_3\dots X_{p-1}]_2 \times 2^e$$

- 先頭の桁は必ず 1 にできる

浮動小数点数の演算

- 整数や固定小数点数よりも複雑

固定小数点数 vs 浮動小数点数

	固定小数点数	浮動小数点数
メリット	実装が簡単	表せる数の範囲が広い
デメリット	表せる数の範囲が狭い	実装が複雑

- コンピューター上で普通に計算する際は、浮動小数点数が使われる場合が多い
- 固定小数点数を採用している言語としては、TEX などがある
  - TEX が作られたのは 1970 年代で、IEEE754 が登場したのは 1985 年
- 貧弱な CPU だと浮動小数点数がハードウェア的に実装されていないケースもある

余談：それ以外の数の表し方

- これまで紹介したのは、使用するビット数が決まっている表し方
  - 普通の用途ではこれで十分
- 速度やメモリを犠牲にしてでも、数を「もっと広い範囲で」「もっと正確に」表したい用途では
  - 多倍長整数：メモリの許す限り何桁でも
  - 有理数：（多倍長）整数の比で表す
  - 任意精度計算：精度 100 桁でも 1000 桁でも（無限精度ではない）

  などを使う

コストを払ってでも高精度計算が必要な状況の例

「OS 標準の電卓アプリで 10 進小数の計算が正しくできない！」みたいなクレームの対策

IEEE 754

アイトリプルイー

- 浮動小数点数の規格
  - 初版は IEEE754-1985、最新版は IEEE754-2008
  - うちの学内ネットワークからはタダで見れるみたい
- 同じ規格に従っていれば、言語やマシンが違っても同じ計算結果になる
  - 例：パソコン vs スマートフォン
- データ交換の形式（どういう数にどういうビット列を対応させるか）を定める
- 2 進小数 (binary) と 10 進小数 (decimal) が定義されているが、この小話では以後 2 進小数 (binary) を扱う

IEEE754 binary

全体のビット数によって、何種類か定められている

	binary16	binary32	binary64	binary128
全体のビット数	16	32	64	128
仮数部の精度 p	11	24	53	113
指数部の範囲	$[-14, 15]$	$[-126, 127]$	$[-1022, 1023]$	$[-16382, 16383]$
指数部のバイアス	15	127	1023	16383
指数部のビット数	5	8	11	15
仮数部のビット数	10	23	52	112

どれを使う？

- よく使われるのは binary64（倍精度）と binary32（単精度）
- 最近は、画像処理や機械学習等の用途で 16 ビット浮動小数点数（半精度）の需要があるらしい（質より量？）
- binary128 （四倍精度）は ~~誰も使ってない~~ 一般的とは言い難い
- ~~x87 には 80 ビットの⋯~~いや何でもない

IEEE754 における数の種類

IEEE754 における浮動小数点数は、以下の 5 つのどれかに該当する：

- 正規化数 (normal)
- ゼロ (zero)
- 非正規化数 (subnormal)
- 無限大 (infinity)
- 非数 (NaN; Not a Number)

binary64 の場合にそれぞれ見ていく

IEEE754 における数の種類：正規化数 (normal)

正規化数とは

$[1.X_1X_2\dots X_{52}]_2 \times 2^e$ $(-1022 \le e \le 1023)$ の形で表せる数

例

$1.25 \times 2^{1023} = [\mathtt{1.4}]_{16} \times 2^{1023} = [\mathtt{1.01}]_2 \times 2^{1023}$

- 符号：正 (0)
- 指数部のビット列：$1023 + 1023 = 2046 = [\mathtt{7fe}]_{16} = [\mathtt{111\ \ 1111\ \ 1110}]_2$
    - 指数部をビット列として表す際には、バイアス（この場合 1023）を加える。
- 仮数部のビット列：$[\mathtt{4000\ \ 0000\ \ 0000\ \ 0}]_{16} = [\mathtt{0100}\ \ \underbrace{00\dots00}_{48\mathit{bits}}]_2$

まとめると

$$1.25 \times 2^{1023} = \mathrm{binary64}([\underbrace{0}_{\text{符号}}\underbrace{11111111110}_{\text{指数部}}\underbrace{00100000\dots0000}_{\text{仮数部 (52 ビット)}}]_2)$$

IEEE754 における数の種類：正規化数 (normal)

正規化数で表せる範囲

最小の正の正規化数：

$$1 \times 2^{-1022} = [1.\underbrace{000\ldots000}_{52 bits}]_2 \times 2^{-1022}$$

$$= \mathrm{binary64}([\underbrace{0}_{\text{符号}}\underbrace{00000000001}_{\text{指数部}}\underbrace{00000000\ldots0000}_{\text{仮数部 (52 ビット)}}]_2)$$

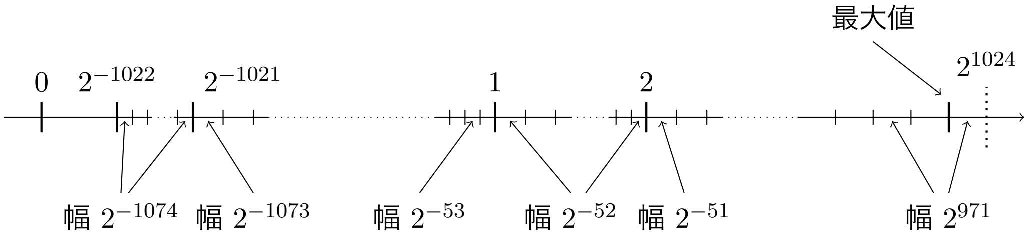
最大の正の正規化数：

$$(2 - 2^{-52}) \times 2^{1023} = [1.\underbrace{111\ldots111}_{52 bits}]_2 \times 2^{1023}$$

$$= \mathrm{binary64}([\underbrace{0}_{\text{符号}}\underbrace{11111111110}_{\text{指数部}}\underbrace{11111111\ldots1111}_{\text{仮数部 (52 ビット)}}]_2)$$

IEEE754 における数の種類：正規化数 (normal)

数直線に図示してみる：



- 絶対値が小さいほど刻み幅が小さい

IEEE754 における数の種類：ゼロ (zero)

浮動小数点数におけるゼロとは

計算結果がゼロだった、または絶対値が（非）正規化数で表現できないほど小さかったことを表す

ゼロは正規化数では表せないので、専用のビット列を使って表す：

$$+0 = \mathrm{binary64}([\underbrace{0}_{\text{符号}}\underbrace{00000000000}_{\text{指数部}}\underbrace{00000000\ldots0000}_{\text{仮数部 (52 ビット)}}]_2),$$

$$-0 = \mathrm{binary64}([\underbrace{1}_{\text{符号}}\underbrace{00000000000}_{\text{指数部}}\underbrace{00000000\ldots0000}_{\text{仮数部 (52 ビット)}}]_2)$$

注意

- ゼロにも符号がある！
- $+0$ と -0 は比較演算では同一視される

IEEE754 における数の種類：非正規化数 (subnormal)

非正規化数とは

「指数部のビット列が 0 で、なおかつ仮数部のビット列が 0 でない」ものを使って、0 より大きく 1×2^{-1022} 未満の数をコードしたもの

$$[0.X_1X_2\dots X_{52}]_2 \times 2^{-1022}$$

例

$$1.25 \times 2^{-1024} = [\mathtt{1.4}]_{16} \times 2^{-1024} = [\mathtt{1.01}]_2 \times 2^{-1024} = [\mathtt{0.0101}]_2 \times 2^{-1022}$$
$$= \mathrm{binary64}([\underbrace{0}_{\text{符号}}\underbrace{00000000000}_{\text{指数部}}\underbrace{01010000\dots0000}_{\text{仮数部 (52 ビット)}}]_2)$$

- 仮数部のビット列：$[\mathtt{5000\ \ 0000\ \ 0000\ \ 0}]_{16} = [\mathtt{0101}\ \ \underbrace{00\dots00}_{48 bits}]_2$

IEEE754 における数の種類：非正規化数 (subnormal)

非正規化数で表せる範囲

最小の正の非正規化数：

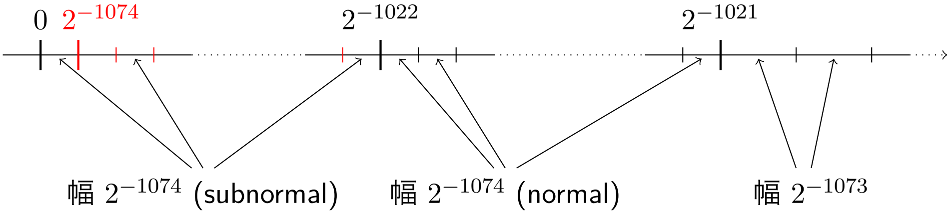
$$1 \times 2^{-1074} = 1 \times 2^{-1022-52} = [0.00\ldots0001]_2 \times 2^{-1022}$$
$$= \mathrm{binary64}([\underbrace{\ 0\ }_{\text{符号}}\underbrace{00000000000}_{\text{指数部}}\underbrace{0000\ldots0001}_{\text{仮数部 (52 ビット)}}]_2)$$

最大の正の非正規化数：

$$(1 - 2^{52}) \times 2^{-1022} = [0.111\ldots111]_2 \times 2^{-1022}$$
$$= \mathrm{binary64}([\underbrace{\ 0\ }_{\text{符号}}\underbrace{00000000000}_{\text{指数部}}\underbrace{1111\ldots1111}_{\text{仮数部 (52 ビット)}}]_2)$$

IEEE754 における数の種類：非正規化数 (subnormal)

数直線に図示してみる：



- 刻み幅は一定（固定小数点数っぽい）

IEEE754 における数の種類：無限大 (infinity)

浮動小数点数における無限大とは

計算結果の絶対値が正規化数で表現できないほど大きかった（2^{1024} 以上)、あるいは、$1/0$ や $\log 0$ を計算しようとしたことを表す

binary64 における無限大は

$$+\infty = \mathrm{binary64}([\underbrace{\ 0\ }_{\text{符号}}\underbrace{11111111111}_{\text{指数部}}\underbrace{00000000\ldots0000}_{\text{仮数部 (52 ビット)}}]_2),$$

$$-\infty = \mathrm{binary64}([\underbrace{\ 1\ }_{\text{符号}}\underbrace{11111111111}_{\text{指数部}}\underbrace{00000000\ldots0000}_{\text{仮数部 (52 ビット)}}]_2)$$

の 2 つ

$+0$ と -0 の逆数は、それぞれの符号の無限大になる

IEEE754 における数の種類：非数 (NaN; Not a Number)

NaN とは

計算結果が実数として ill-defined だったことを表す（例：$0/0$, $\sqrt{-1}$, $\infty - \infty$）

性質

- NaN が絡む演算の結果は NaN となる
  - 例：$\mathit{NaN} \times 0 = \mathit{NaN}$
- 比較演算では「自身と同一でない」と判断される
  - これを利用して計算結果が NaN かどうかを判断できる

ビットパターンの例

$$\text{binary64}([\underbrace{0}_{\text{符号}}\underbrace{11111111111}_{\text{指数部}}\underbrace{10000000\ldots0000}_{\text{仮数部 (52 ビット)}}]_2)$$

IEEE754 における数の種類：非数 (NaN; Not a Number)

余談：NaN の応用

$$\left[\underbrace{0}_{\text{符号}}\ \underbrace{11111111111}_{\text{指数部}}\ \underbrace{\overbrace{1}^{\text{NaN の種類}}\quad * * * * * * * \ldots * * * *}_{\text{仮数部 (52 ビット)}}\right]_2$$

- 仮数部におよそ 51 ビット分の情報を持てる
  - 仮数部は 52 ビットあるが、先頭ビットは NaN の種類 (quiet/signaling) を表すのに使われる
  - 仮数部が完全に 0 であってはいけない
- NaN tagging / NaN trick （スクリプト言語処理系の実装に使われるテクニック）
  - 一つの 64 ビット値に、スクリプト言語における値を保持できる（普通はデータの種類を表すのに数ビット、実際のデータを表すのに 64 ビット必要）
  - LuaJIT が発祥（のはず；2009 年ごろ）で、その後 JavaScript の処理系などでも採用されているらしい

IEEE754 における数の種類

まとめ

指数部のビット列	仮数部	種類	値
$[000\ 0000\ 0000]_2$	0	ゼロ	± 0
$[000\ 0000\ 0000]_2$	$\neq 0$	非正規化数	$\pm 0.[\text{仮数部}] \times 2^{-1022}$
$[000\ 0000\ 0001]_2, \ldots$ $\ldots, [111\ 1111\ 1110]_2$		正規化数	$\pm 1.[\text{仮数部}] \times 2^{(\text{指数部のビット列})-1023}$
$[111\ 1111\ 1111]_2$	0	無限大	$\pm\infty$
$[111\ 1111\ 1111]_2$	$\neq 0$	NaN	

演算と丸め方向

計算結果を正確に表せない場合にどうするか？

例

0.1 は 2 進法だと循環小数になる：

$$0.1 = \frac{1}{10} = [\texttt{1.9999}\cdots]_{16} \times 2^{-4} = [\texttt{1.1001 1001 1001}\cdots]_2 \times 2^{-4}$$

計算結果を正確に表せない場合は、近い値へ丸める：

- 最近接丸め：$0.1 \rightsquigarrow [\texttt{1.1001} \cdots \texttt{1001 1010}]_2 \times 2^{-4}$
- 負の無限大方向：$0.1 \rightsquigarrow [\texttt{1.1001} \cdots \texttt{1001 1001}]_2 \times 2^{-4}$
- 正の無限大方向：$0.1 \rightsquigarrow [\texttt{1.1001} \cdots \texttt{1001 1010}]_2 \times 2^{-4}$
- ゼロ方向：$0.1 \rightsquigarrow [\texttt{1.1001} \cdots \texttt{1001 1001}]_2 \times 2^{-4}$

丸め方向の利用：区間演算と精度保証

- 丸めが発生すると、正確な計算はできない
- それでも、丸め方向を上手く制御すると、計算結果の上界や下界を与えることはできる
- 詳しくは「区間演算」や「精度保証」で調べて

(2 進) 浮動小数点数の罠：10 進小数との兼ね合い

最近接丸めで $(0.1 + 0.1) + 0.1$ を計算してみよう
まず、 $0.1 + 0.1$ は

$$
\begin{array}{rlr}
 & 0.1 \overset{\text{丸め}}{\rightsquigarrow} & [\texttt{1.1001 1001} \cdots \texttt{1001 1010}]_2 \times 2^{-4} \\
+) & 0.1 \overset{\text{丸め}}{\rightsquigarrow} & [\texttt{1.1001 1001} \cdots \texttt{1001 1010}]_2 \times 2^{-4} \\
\hline
 & & [\texttt{11.0011 0011} \cdots \texttt{0011 0100}]_2 \times 2^{-4} \\
 & & \overset{\text{丸め}}{\rightsquigarrow} [\texttt{11.0011 0011} \cdots \texttt{0011 010 }]_2 \times 2^{-4}
\end{array}
$$

なので、 $(0.1 + 0.1) + 0.1$ は

$$
\begin{array}{rlr}
 & & [\texttt{11.0011 0011} \cdots \texttt{0011 010 }]_2 \times 2^{-4} \\
+) & 0.1 \overset{\text{丸め}}{\rightsquigarrow} & [\texttt{1.1001 1001} \cdots \texttt{1001 1010}]_2 \times 2^{-4} \\
\hline
 & & [\texttt{100.1100 1100} \cdots \texttt{1100 1110}]_2 \times 2^{-4} \\
 & & \overset{\text{丸め}}{\rightsquigarrow} [\texttt{1.0011 0011} \cdots \texttt{0011 0100}]_2 \times 2^{-2}
\end{array}
$$

となる

（2 進）浮動小数点数の罠：10 進小数との兼ね合い

一方、

$$0.3 = 3/10 = [\mathtt{1.3333}\cdots]_{16} \times 2^{-2} = [\mathtt{1.0011\ \ 0011\ \ 0011}\cdots]_2 \times 2^{-2}$$
$$\overset{\text{丸め}}{\rightsquigarrow} [\mathtt{1.0011\ \ 0011}\ \ \cdots\ \ \mathtt{0011\ \ 0011}]_2 \times 2^{-2}$$

なので、binary64 では $(0.1 + 0.1) + 0.1 \neq 0.3$ となる

解決策

10 進小数を正確に扱いたいなら 2 進の浮動小数点数ではなくて 10 進の（浮動）小数点数を使え

浮動小数点数の罠：演算の結合法則

$1 + 2^{-53} + 2^{-53}$ を（binary64 における最近接丸めで）計算してみよう

$(1 + 2^{-53}) + 2^{-53}$ の場合：

$$(1 + 2^{-53}) + 2^{-53}$$

$1 + (2^{-53} + 2^{-53})$ の場合：

$$1 + (2^{-53} + 2^{-53})$$

浮動小数点数の罠：演算の結合法則

$1 + 2^{-53} + 2^{-53}$ を（binary64 における最近接丸めで）計算してみよう

$(1 + 2^{-53}) + 2^{-53}$ の場合：

$$(1 + 2^{-53}) + 2^{-53} \overset{\text{丸め}}{\rightsquigarrow} 1 + 2^{-53}$$

$1 + (2^{-53} + 2^{-53})$ の場合：

$$1 + (2^{-53} + 2^{-53}) = 1 + 2^{-52}$$

浮動小数点数の罠：演算の結合法則

$1 + 2^{-53} + 2^{-53}$ を（binary64 における最近接丸めで）計算してみよう

$(1 + 2^{-53}) + 2^{-53}$ の場合：

$$(1 + 2^{-53}) + 2^{-53} \overset{\text{丸め}}{\leadsto} 1 + 2^{-53} \overset{\text{丸め}}{\leadsto} 1$$

$1 + (2^{-53} + 2^{-53})$ の場合：

$$1 + (2^{-53} + 2^{-53}) = 1 + 2^{-52} \quad \text{(これは binary64 で正確に表せる)}$$

なので、$(1 + 2^{-53}) + 2^{-53} \neq 1 + (2^{-53} + 2^{-53})$ となる（結合法則が成り立たない！）

浮動小数点数の罠：指数部のオーバーフロー・アンダーフロー

$\mathrm{hypot}(x, y) = \sqrt{x^2 + y^2}$ 関数を次のように素朴に実装したとする：

```
double naive_hypot(double x, double y) {
    return sqrt(x * x + y * y);
}
```

（注：hypot は直角三角形の斜辺 (hypotenuse) の略）

問題点：指数部のオーバーフロー・アンダーフロー

$x = 3 \times 2^{1000}$, $y = 4 \times 2^{1000}$ に対して $\mathrm{hypot}(x, y) = 5 \times 2^{1000}$ となるべきだが…？

浮動小数点数の罠：指数部のオーバーフロー・アンダーフロー

$\mathrm{hypot}(x, y) = \sqrt{x^2 + y^2}$ 関数を次のように素朴に実装したとする：

```
double naive_hypot(double x, double y) {
    return sqrt(x * x + y * y);
}
```

（注：hypot は直角三角形の斜辺 (hypotenuse) の略）

問題点：指数部のオーバーフロー・アンダーフロー

$x = 3 \times 2^{1000}$, $y = 4 \times 2^{1000}$ に対して $\mathrm{hypot}(x, y) = 5 \times 2^{1000}$ となるべきだが…？

$x^2 = 9 \times 2^{2000}$, $y^2 = 16 \times 2^{2000}$

浮動小数点数の罠：指数部のオーバーフロー・アンダーフロー

$\mathrm{hypot}(x, y) = \sqrt{x^2 + y^2}$ 関数を次のように素朴に実装したとする：

```
double naive_hypot(double x, double y) {
    return sqrt(x * x + y * y);
}
```

（注：hypot は直角三角形の斜辺 (hypotenuse) の略）

問題点：指数部のオーバーフロー・アンダーフロー

$x = 3 \times 2^{1000}$, $y = 4 \times 2^{1000}$ に対して $\mathrm{hypot}(x, y) = 5 \times 2^{1000}$ となるべきだが…？

$x^2 = 9 \times 2^{2000} \overset{\text{丸め}}{\rightsquigarrow} +\infty$, $y^2 = 16 \times 2^{2000} \overset{\text{丸め}}{\rightsquigarrow} +\infty$ なので、

$$\texttt{naive\_hypot}(x, y) = \texttt{sqrt}((+\infty) + (+\infty)) = +\infty$$

となる（不適切！）

浮動小数点数の罠：指数部のオーバーフロー・アンダーフロー

解決策

- 計算前に x, y の指数部を適切な範囲に収める
（この場合なら $m = 1002$ として $2^m \cdot \sqrt{(x \cdot 2^{-m})^2 + (y \cdot 2^{-m})^2}$ と計算させる）
- または、割り算を使って

$$\begin{cases} |y| \cdot \sqrt{1 + (x/y)^2} & (|x| \le |y|) \\ |x| \cdot \sqrt{1 + (y/x)^2} & (|y| \le |x|) \end{cases}$$

と計算する

複素数の除算（逆数）にも似たような罠がある

浮動小数点数の罠：情報落ち・桁落ち

$\sinh x = \frac{\exp(x)-\exp(-x)}{2}$ を次のように素朴に実装したとする：

```
double naive_sinh(double x) {
    return (exp(x) - exp(-x)) / 2.0;
}
```

問題点：0 の近くでの挙動が不適切

$|x| \ll 1$ の場合 $\sinh x \approx x$ となるべきだが…？

浮動小数点数の罠：情報落ち・桁落ち

$\sinh x = \frac{\exp(x) - \exp(-x)}{2}$ を次のように素朴に実装したとする：

```
double naive_sinh(double x) {
    return (exp(x) - exp(-x)) / 2.0;
}
```

問題点：0 の近くでの挙動が不適切

$|x| \ll 1$ の場合 $\sinh x \approx x$ となるべきだが…？
$x = 2^{-100}$ の場合

$$\exp\left(2^{-100}\right) = 1 + 2^{-100} + \cdots \qquad , \qquad \exp\left(-2^{-100}\right) = 1 - 2^{-100} + \cdots$$

浮動小数点数の罠：情報落ち・桁落ち

$\sinh x = \frac{\exp(x)-\exp(-x)}{2}$ を次のように素朴に実装したとする：

```
double naive_sinh(double x) {
    return (exp(x) - exp(-x)) / 2.0;
}
```

問題点：0 の近くでの挙動が不適切

$|x| \ll 1$ の場合 $\sinh x \approx x$ となるべきだが…？
$x = 2^{-100}$ の場合

$$\exp\left(2^{-100}\right) = 1 + 2^{-100} + \cdots \overset{\text{丸め}}{\rightsquigarrow} 1, \qquad \exp\left(-2^{-100}\right) = 1 - 2^{-100} + \cdots \overset{\text{丸め}}{\rightsquigarrow} 1$$

なので、`naive_sinh`$\left(2^{-100}\right) = (1 - 1)/2 = 0$ となる

浮動小数点数の罠：情報落ち・桁落ち

解決策

- $\mathrm{expm1}(x) = \exp(x) - 1 = x + x^2/2 + \cdots$ を導入
- ```
  double sinh(double x) {
      return (expm1(x) - expm1(-x)) / 2.0;
  }
  ```
  と定義すれば良い
- $\frac{\mathrm{expm1}\left(2^{-100}\right) - \mathrm{expm1}\left(-2^{-100}\right)}{2.0} = \frac{2^{-100} - (-2^{-100})}{2.0} = 2^{-100}$
  ```

TL;DR

- コンピューターで計算をさせるときは浮動小数点数の性質を知っておこう
- 浮動小数点数の長所も短所も把握した上で、上手く付き合っていこう
- 「0.1 を 10 回足しても 1.0 にならない、これはバグだ」と大騒ぎするような大人にはならな