[0.0, 1.0) の乱数を得るための“本当の”方法

レイトレ合宿9
hole (@h013)

＊2023/09/03時点での調査結果に基づいています。
＊2023/09/08 “除算法2”追記。(@Reputelessさんありがとうございました)

イントロダクション

- 様々な用途で(疑似)乱数を得たいことは多い。
  - シミュレーション
  - 暗号
  - 確率的アルゴリズム
  - ゲーム
  - コンピュータグラフィックス
  - etc.

イントロダクション

- 整数乱数を得る手法は無数にあり、かつ、一般的な用途においては十分な速度と品質を達成している（とされる）
  - 線形合同法
  - メルセンヌ･ツイスタ
  - xorshift 系列
  - Permuted Congruential Generator
  - Random123
  - etc.

イントロダクション

- 一方、浮動小数の乱数を得る手法は自明ではない。
- 今回、もっとも単純な問題設定である、以下について考察する。

`[0.0, 1.0)` の範囲の一様乱数を `float` で得るための方法

“間違っている”可能性が高い！
（かもしれない）

考察する手法

- 今回は以下の手法について考察する。
  - std::generate_canonical
  - 除算法
  - mantissa(仮数部)法
  - 除算法2
  - exponent+mantissa法
  - Abseilの最適化
  - Downeyの方法

- 名称は便宜的に付けたものも含まれる。

std::generate_canonical

std::generate_canonical

- まずは標準C++ライブラリから始める。標準C++ライブラリには[0.0, 1.0) 区間で一様乱数を得る関数が用意されている。

  実数区間[0.0, 1.0)に展開(事実上正規化)された一様分布乱数を得るための関数テンプレート。 テンプレート引数に与える RealType 型について、bits だけの分解能を持つ仮数部を URBG g を必要な回数だけ繰り返し呼び出して生成する。

  https://cpprefjp.github.io/reference/random/generate_canonical.html

- std::uniform_real_distribution の実装に使われていることが多い。

std::generate_canonical

- 使い方
  - std::generate_canonical<float, 23>(random_generator);
  - std::generate_canonical<double, 52>(random_generator);
  - 返却する型(float,double等)と、仮数部の分解能の最低要求(float, doubleのフルスペックなら23bit, 52bit)を指定する。

- random_generator の生成ビット数と、分解能要求に応じて、複数回の乱数生成が行われる。
  - 32bit乱数生成器、52bit分解能の場合、二回乱数生成が行われる。

- 今回はfloatに限定しているので、32bit乱数生成器、23bit分解能で良い。

std::generate_canonicalの問題

- これを使えばいいのか？ → ダメ！
- 幾つかの問題が存在している。
  - 問題1:一部のライブラリ実装において、[0.0, 1.0)が保証されない。
  - 問題2:一部のライブラリ実装において、非常に低速。
  - 問題3:一部のライブラリ実装において、[0.0, 1.0)の範囲における、表現可能な全てのfloat値を得ることが出来ない。

問題1:一部のライブラリ実装において、[0.0, 1.0) が保証されない

- ライブラリ実装についてエッジケースを観察する。
  - 0xffffffffを乱数として与える。1.0fが出力されなければOK。
- 実験方法
  - 返却型として、floatを使用。
  - 乱数生成器として、32bitを使用。
  - 以下のライブラリコードを抽出、共通のコンパイラ上でコンパイル・実行。ターゲットはx64。
- 対象ライブラリ
  - msvc new (>=19.32)
  - msvc old (<= 19.31)
  - gcc libstdc++ (2023/08/22 時点でのrepo最新)
  - clang libc++ (2023/08/22 時点でのrepo最新)
  - Abseil (2023/08/22 時点でのrepo最新)
    - Abseil については同様の機能を持つ別の関数、GenerateRealFromBitsを使用。

結果

- msvc new
  - 32bit rng/float: 1.000000000000000e+00
- msvc old
  - 32bit rng/float: 1.000000000000000e+00
- gcc libstdc++
  - 32bit rng/float: 9.999994039535522e-01
- clang libc++
  - 32bit rng/float: 1.000000000000000e+00
- Abseil
  - 64bit rng/float: 9.999994039535522e-01

結果

- msvc new
  - 32bit rng/float: 1.000000000000000e+00
- msvc old
  - 32bit rng/float: 1.000000000000000e+00
- gcc libstdc++
  - 32bit rng/float: 9.999994039535522e-01
- clang libc++
  - 32bit rng/float: 1.000000000000000e+00
- Abseil
  - 64bit rng/float: 9.999994039535522e-01

多くのライブラリで、1.0fを出力してしまう！

原因:generate_canonicalの実装

- いずれのライブラリにおいても、本質的には以下の様になっている(Abseil除く)
  - `(return_type)random() / ((return_type)(RandomMax)+(return_type)1)`

- たとえばreturn_type=float、32bit乱数生成器の場合、丸め誤差により以下のような計算結果になる！
  - `RandomMax = 0xffffffff = 4294967295`
  - 分子: `(float)4294967295 → 4294967296.0f`
  - 分母: `((float)(4294967295) + (float)1) → 4294967296.0f`

- よって、random()の結果がRandomMaxだと最終結果が1.0fとなる。

原因:generate_canonicalの実装

- gccにおける対策:next_after
  - if (__ret >= _RealType(1))
  - {
  - __ret = std::nextafter(_RealType(1), _RealType(0));
  - }

指定方向への次の表現可能な値を取得する。この関数は、パラメータxの値をパラメータyの値の方向に対して、その環境で表現可能な最小の値だけ進める。

https://cpprefjp.github.io/reference/cmath/nextafter.html

- 1より小さい、最大の浮動小数値を返す。
  - 実用上は問題にならないと考えられるが、該当の値の生成確率がバイアスされる。

原因:generate_canonicalの実装

- Abseilにおける対策:exponent+mantissa法
  - あとで詳しく説明。
  - やや複雑な手法を使い、1.0が出ないようになっている。

問題2：一部のライブラリ実装において、非常に低速

- msvc old (<= 19.31) における実装は非常に低速。
  - 1億回の実行に要する平均時間(Intel®Xeon®Gold 6140 CPU @ 2.30GHz)
    - msvc new:　　0.23 [sec]
    - msvc old:　　4.18 [sec]
    - gcc libstdc++:　　0.36 [sec]
    - clang libc++:　　0.21 [sec]
    - Abseil　　0.41 [sec]

- 呼び出しごとに、log2(bits)を計算しており遅い。が、bitsはコンパイル時に決定できるため、無駄。
  - 他の実装はコンパイル時に計算するための工夫が行われている。
  - msvc 19.32からコンパイル時に計算するようになり、高速化された！
    - かなり最近。

問題3:一部のライブラリ実装において、[0.0, 1.0) の範囲における、表現可能な全てのfloat値を得ることが出来ない

- Abseil以外の実装は全てこの問題を抱える。
  - この問題については、あとで詳しく考察する。

std::generate_canonicalまとめ

- ○ 標準ライブラリに含まれているため使いやすい。
- × 1.0（端）付近の挙動がライブラリ実装に依存しており、規格外の挙動をすることがある。（[0.0, 1.0)が保証されると限らない）
- × 低速なライブラリ実装が存在する。
- × 一部のライブラリ実装において、[0.0, 1.0) の範囲における、表現可能な全てのfloat値を得ることが出来ない。

除算法

除算法

- generate_canonicalのアルゴリズムをそのまま取り出す。
  - `float r = (float)random() / ((float)(RandomMax)+(float)1);`

- 先述したような問題がある。
  - `[0.0, 1.0)` が保証されない。
    - 丸め誤差によって`1.0f`が出てしまう。これはすでに述べた通り。
    - 解決策は、エッジが出たら再生成 or クランプ、など。
  - `[0.0, 1.0)` の範囲における、表現可能な全ての`float`値を得ることが出来ない。
    - この問題について考察する。

除算法:浮動小数の構造について

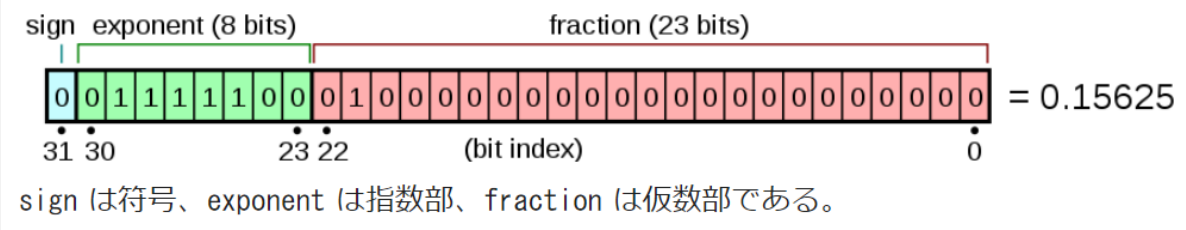
- IEEE754において浮動小数に関する規格が定められている。バイナリ表現と、対応する浮動小数の値も決められている。
  - https://float.exposed/

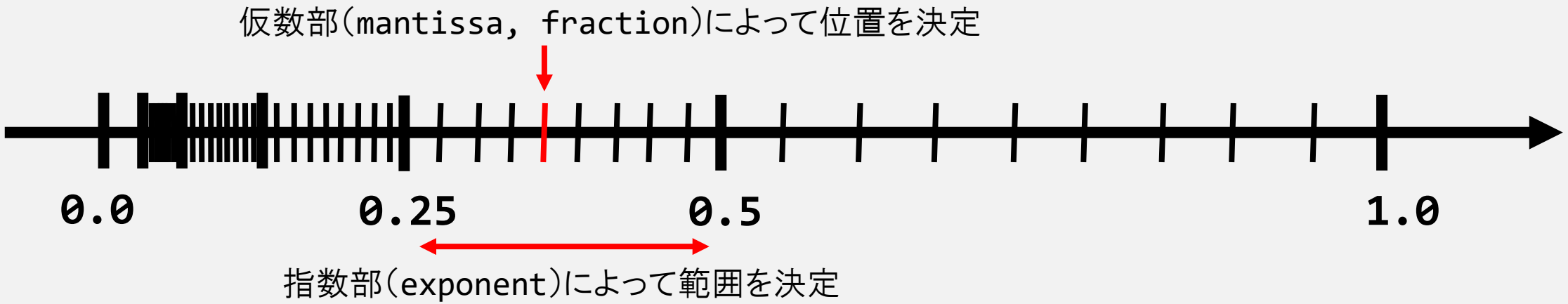
バイナリ表現	対応する値
0x3f800000	1.0f
0x3db634fb	0.08896823972463607788O9f
0x52f77c6a	531472121856.0f

除算法：浮動小数の構造について

- ビット上の構造
  - fractionはmantissaとも呼ぶ

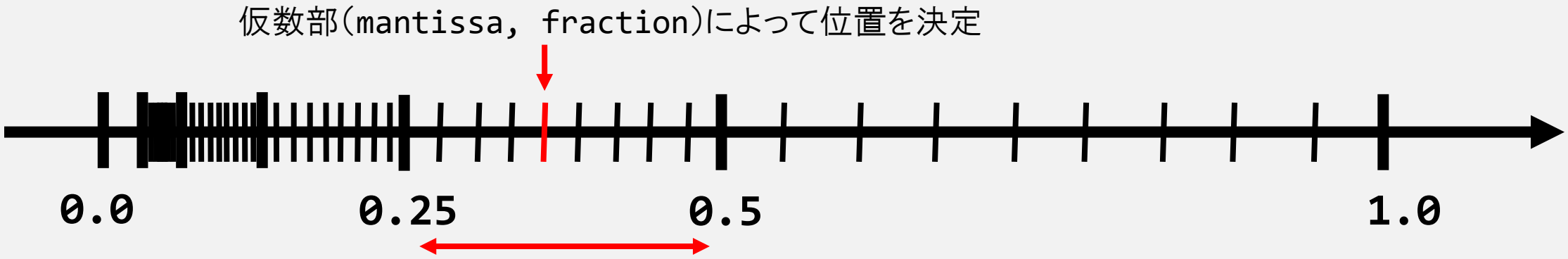
$$(-1)^{sign} \cdot 2^{exponent-127} \cdot (1.fraction)$$

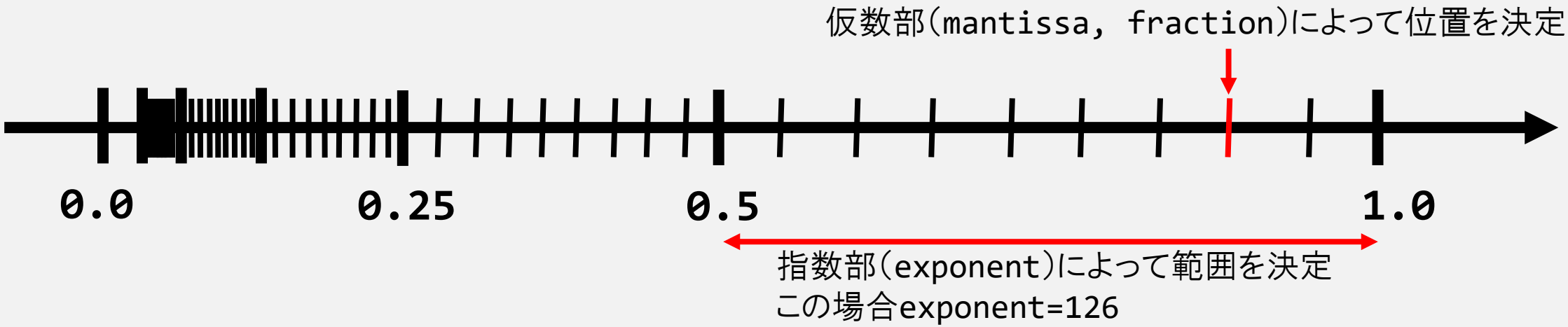




除算法：浮動小数の構造について

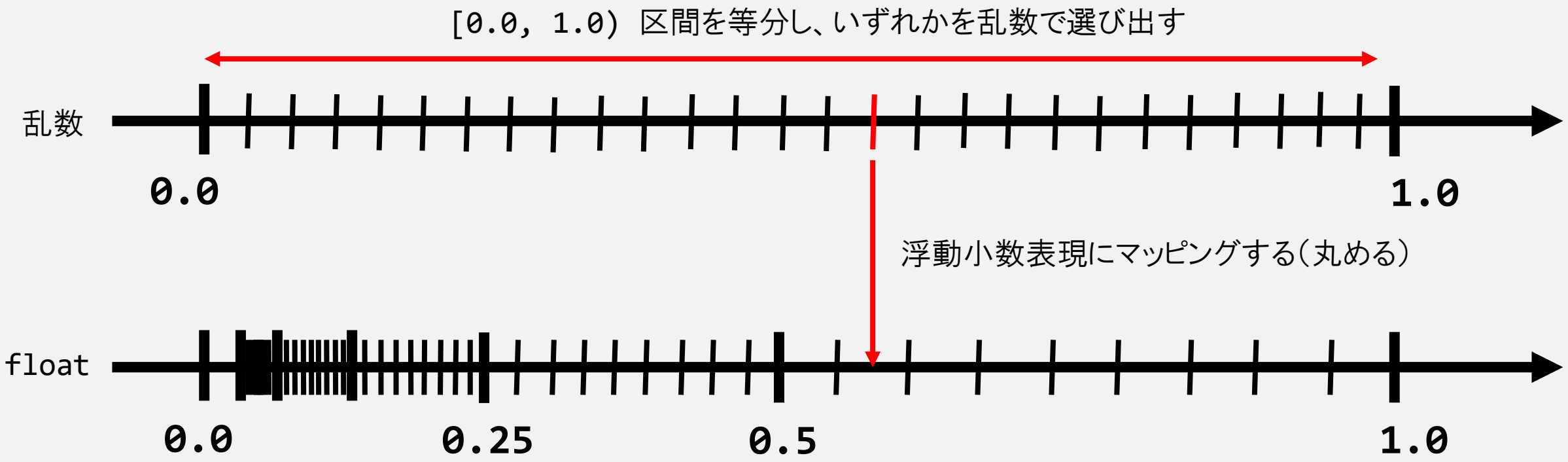
- ビット上の構造





除算法

- 除算法は以下のようなイメージとなる。



表現可能な値

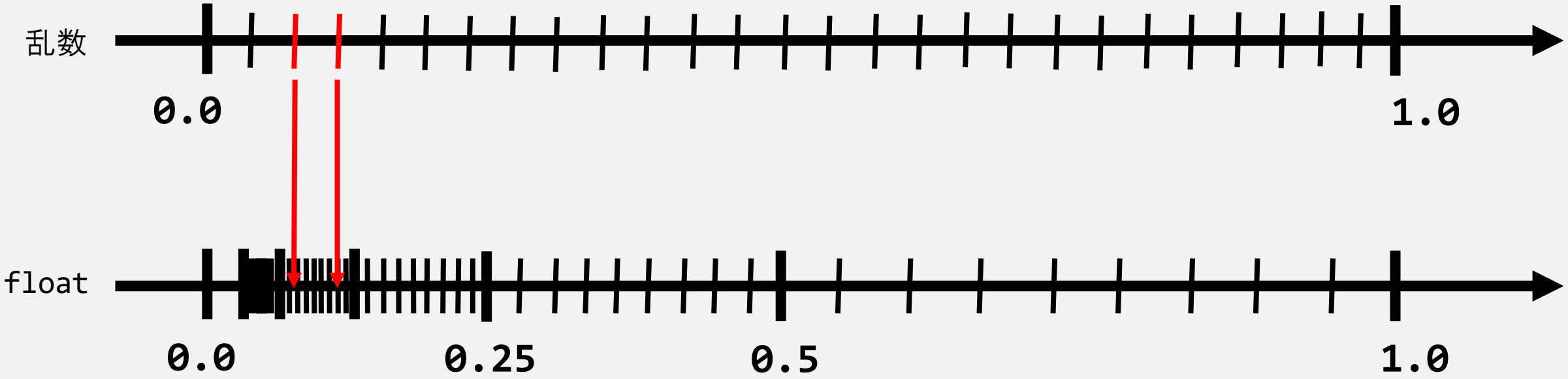
- 除算法によって得られる、0より大きい最小の値
  - `float f = (float)1 / ((float)0xffffffff + (float)1);`
  - これは `2.32830644e-10` となる。

- 一方、floatで表現可能な、0より大きい最小の値は
  - 正規化数の範囲なら `1.17549435e-38`
  - 非正規化数も含むと `1.40129846e-45`

- floatの指数部は正規化数の範囲で2^-126(=1.17549435e-38)まで小さくなる。しかし、除算法で得られる、`2.32830644e-10` というのは指数でいえば2^-29から2^-30の間くらいの精度しかない。

表現可能な値

- すなわち、本来floatで表現可能な0付近の値を、除算法では得ることが出来ない。
  - [0.0, 1.0) の範囲における、表現可能な全てのfloat値を得ることが出来ない。
  - 乱数による量子化と、浮動小数表現による量子化のズレ。



問題になるのか？

- 一般的な一様乱数の用途においては、問題になることは少ないと考えられる。しかし、問題になるケースもありえるかもしれない。

表現可能な値に関する問題

- exponential distribution
  - 乱数ξを使って、u=-logn(ξ)によって得られる分布。ボリュームレンダリングとかでも良く使う。
  - ξは(0.0, 1.0]の乱数が使われる。

- ξとして理想的なfloatの範囲をとるならば、最小値は（既に述べたように）正規化数で1.17549435e-38となり、uの最大値は87.34となる。しかし、除算法ではuの最大値は22.18になる。
  - 本来、uの最大値は∞なので、いずれも真の分布になっていないという本質的な問題は別途存在する。

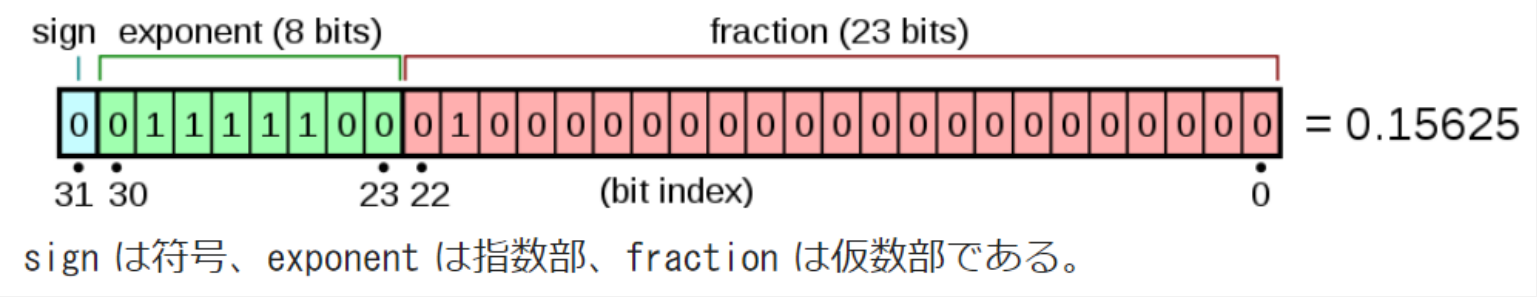
除算法まとめ

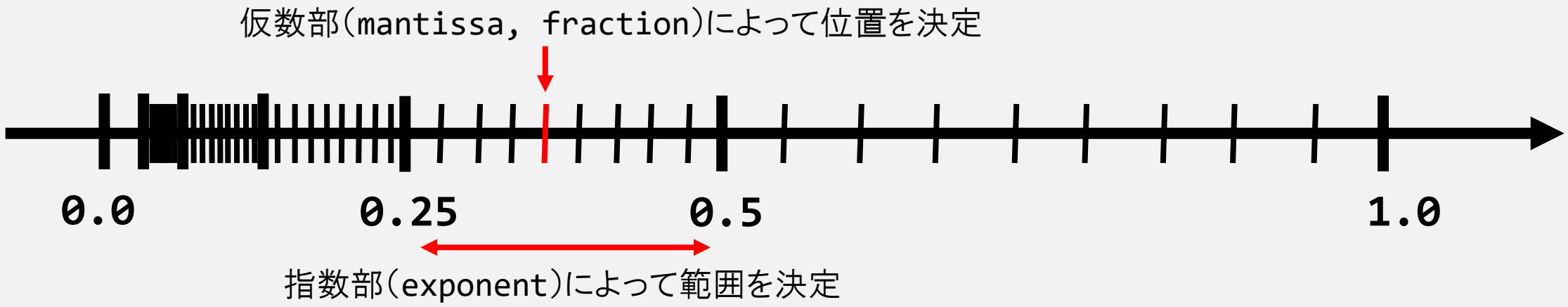
- ○ 実装が自明、かつ単純。
- × `[0.0, 1.0)` が保証されない。
- × `[0.0, 1.0)` の範囲における、表現可能な全ての`float`値を得ることが出来ない。

mantissa(仮数部)法

mantissa(仮数部)法

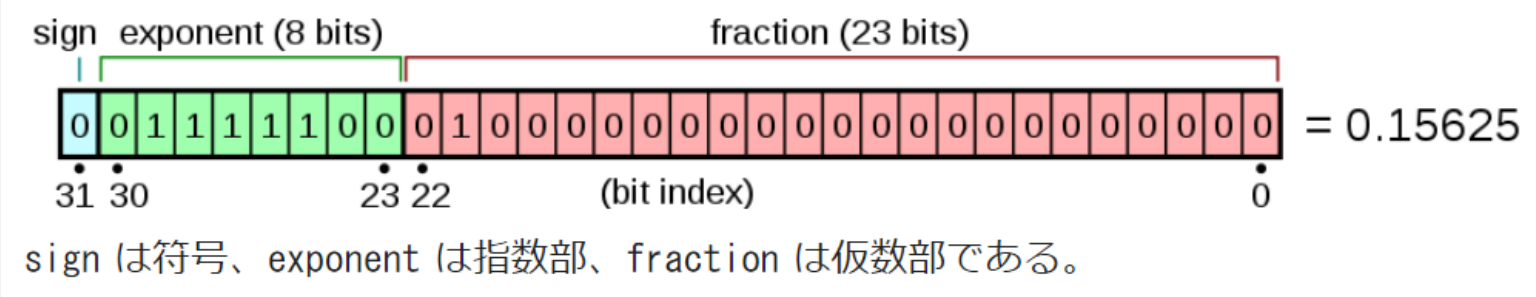
- [0.0, 1.0) の一様乱数を得る方法として、mantissa(仮数部)を直接埋める方法がある。





mantissa(仮数部)法

- まず、exponentとして[1.0, 2.0)の範囲をとる値を与える。
  - exponent=127
- 次に、mantissa(23bit)を乱数で埋める。この時点で、[1.0, 2.0)の範囲の一様乱数を完璧に得られる。
- 最後に、上記乱数から1.0を引くと、[0.0, 1.0) の範囲の一様乱数になる。



mantissa(仮数部)法

- 実際の実装例

```
template·<typename·rng_t>
float·rand01_mantissa(rng_t&·rng)
{
····const·uint32_t·u·=·rng();
····const·uint32_t·mantissa·=·u·&·0x7fffff;
····const·uint32_t·exponent·=·127;
····//·[1.0,·2.0)
····const·float·r12·=·bitToFloat((exponent·<<·23)·|·mantissa);
····//·[0.0,·1.0)
····const·float·r01·=·r12·-·1.0f;
····return·r01;
}
```

mantissa(仮数部)法

- ちゃんと開区間を表現できる。
- 除算法などと違って、単なるビット演算とfloatへの変換のみなので、ある種のハードウェアでは高速に実行されることが期待される。
  - とはいっても、今時のCPUでは、そこまでパフォーマンスは有利ではない。
  - 1億回の実行に、0.20 [sec]程度。

- 得られる、0より大きい最小値は1.19209290e-07となる。
  - 表現可能な値に比べると、かなり大きめ。
- 大きな問題として、最終結果の乱数のmantissaにおける最下位ビットが常にゼロになるという事実がある。
  - 表現可能な値が少ない。

mantissa(仮数部)法

- 最終結果の乱数の最下位ビットが常にゼロになる。
  - 0.348863: 00111110101100101001111000110100
  - 0.971680: 00111111011110001100000000000110
  - 0.060423: 00111101011101110111111001100000
  - 0.093260: 00111101101111101111111011100000
  - 0.683346: 00111111001011101110111111000110
  - 0.770189: 00111111010001010010101100100000
  - 0.236677: 00111110011100100101101101101000
  - 0.465761: 00111110111011100111100000111000
  - 0.824375: 00111111010100110000101000111010
  - 0.243600: 00111110011110010111001001101000
  - 0.395521: 00111110110010101000000111000100
  - 0.182249: 00111110001110101001111110010000
  - 0.198098: 00111110010010101101101000100000
  - 0.712343: 00111111001101100101110000010100
  - 0.576150: 00111111000100110111111010001100
  - 0.880710: 00111111011000010111011000111000

mantissa(仮数部)法まとめ

- ○ 実装が自明、かつ単純。
- ○ [0.0, 1.0) が保証される。
- ○ doubleにも自然に拡張可能。
- × [0.0, 1.0) の範囲における、表現可能な全てのfloat値を得ることが出来ない。

  しかも、mantissaの最下位ビットが常にゼロになる。
- × 除算法と比べても、出力される最小値が大きい。

  除算法:2.32830644e-10

  mantissa法:1.19209290e-07

余談：ビットパターンをfloatに変換するには

- この手法のような操作をする際、任意のビットパターンをそのままfloatとして解釈したいことがよくある。（逆変換も）
  - 一般に、type punning と呼ばれる。
- C++においては以下のような、 reinterpret_castやunionによるテクニックは undefined behaviorとされるので注意が必要。
  - Strict Aliasing 規則に抵触する。
  - https://blog.regehr.org/archives/959

```
float bitToFloat(uint32_t u) {
    return *reinterpret_cast<float*>(&u);
}

float bitToFloat(uint32_t u) {
    union {
        uint32_t u;
        float f;
    } t;
    t.u = u;
    return t.f;
}
```

余談：ビットパターンをfloatに変換するには

- Strict Aliasing規則に抵触する具体例。
  - 以下コードは gcc 13.2 で出力結果が変わる。
    - 最適化OFFだと 6
    - 最適化ONだと 5
  - https://godbolt.org/z/sjPjWEzo6

```
void check(int* h, long* k) {
    *h = 5;
    *k = 6;
    printf("%d\n", *h);
}

int main(void) {
    long k;
    check((int*)&k, &k);
    return 0;
}
```

余談：ビットパターンをfloatに変換するには

- 以下のようにするのが安全とされている。
  - memcpyを用いるやり方か、C++20から導入されるbit_castを使う。
  - memcpyは現代のコンパイラは高度に最適化してくれる。

```
float bitToFloat(uint32_t u) {
    float f;
    std::memcpy(&f, &u, sizeof(u));
    return f;
}

float bitToFloat(uint32_t u) {
    return std::bit_cast<float>(u);
}
```

除算法2

除算法2

- 除算法を修正する
  - `float r = (float)(random() >> 8) / 16777216.0f;`

- 改善点
  - `[0.0, 1.0)` が保証される！
    - 乱数として`0xffffffff`が出た時、`9.9999940e-01`となる。
  - `mantissa`法より分解能が高い！
    - `2^24`の分解能となる。`mantissa`法は`2^23`。

- 分解能について
  - 表現可能な`float`値全てを得ることはできない。
  - 出力される`0`より大きい最小の値は、`5.9604644775e-08`
    - 除算法:`2.32830644e-10`
    - `mantissa`法:`1.19209290e-07`

除算法2まとめ

- ○ 実装が自明、かつ単純。
- ○ [0.0, 1.0) が保証される。
- ○ doubleにも自然に拡張可能。
- ○ mantissa法より分解能が高い(2^24)
- × [0.0, 1.0) の範囲における、表現可能な全てのfloat値を得ることが出来ない。
- × 除算法と比べても、出力される最小値が大きい。

    mantissa法よりは小さい

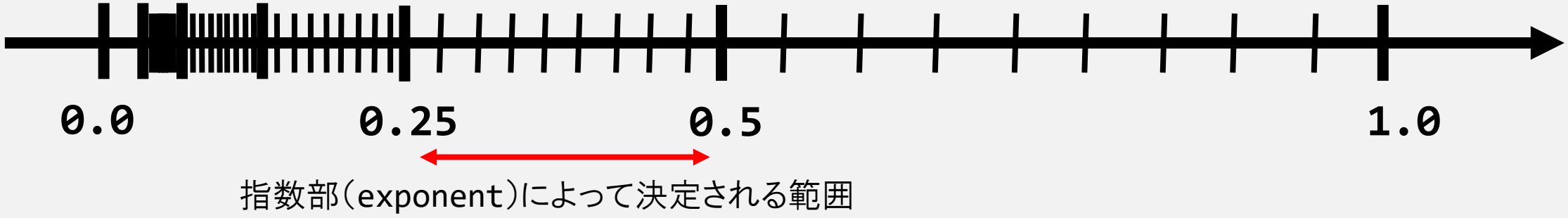
exponent+mantissa法

exponent+mantissa法

- [0.0, 1.0)区間の乱数を出力可能、かつ、表現可能な全てのfloat値を得ることが出来る。

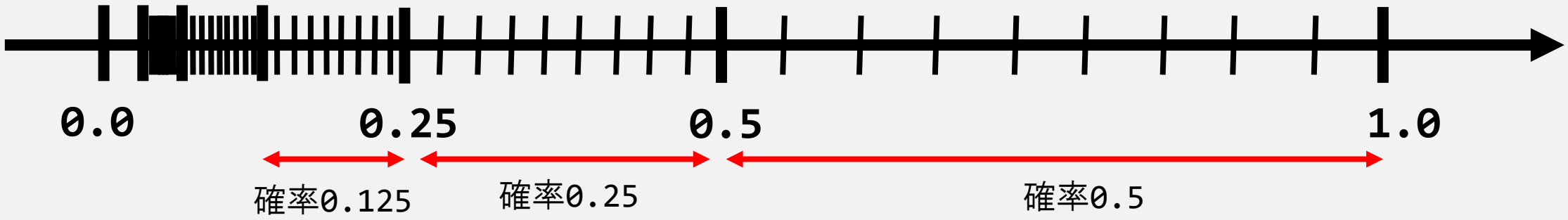
exponent+mantissa法

- アルゴリズムの概要
  - まず、乱数に基づいてexponentを決定する。この時、各区間の選択確率が幅に比例するようにする。



exponent+mantissa法

- アルゴリズムの概要
  - まず、乱数に基づいてexponentを決定する。この時、各区間の選択確率が幅に比例するようにする。



exponent+mantissa法

- アルゴリズムの概要
  - 1bit乱数を得る。
  - 1なら、[0.5, 1.0)区間のexponentに決定。
  - 0なら、再び1bit乱数を得る。
  - 1なら、[0.25, 0.5)区間のexponentに決定。
  - 0なら、再び1bit乱数を得る。以上を繰り返す。

```
 RandomBitProvider provider(rng);

 constexpr float lowV = 0.0f;
 constexpr float highV = 1.0f;
 const uint32_t lowExp = ((bit(lowV) >> 23) & 0xff);
 const uint32_t highExp = ((bit(highV) >> 23) & 0xff);

 uint32_t exponent;
 for (exponent = highExp - 1; exponent > lowExp; --exponent) {
     if (provider.getBit()) {
         break;
     }
 }
```

exponent+mantissa法

- アルゴリズムの概要
  - exponentが決定されれば、あとはmantissa法のように、mantissa 23bitを乱数で埋める。

```
··uint32_t·mantissa;
··mantissa·=·rng()·&·0x7fffff;
```

- これにより、[0.0, 1.0)区間の一様乱数を得ることができる。かつ、表現可能なfloat値を全て得ることができる。
  - 全てのexponentおよび全てのmantissaが出力されるため。

exponent+mantissa法まとめ

- ○ [0.0, 1.0) が保証される。
- ○ doubleにも自然に拡張可能。
- ○ [0.0, 1.0) の範囲における、表現可能な全てのfloat値を得ることが出来る！
- × 実装がやや複雑。（実行速度も不利になりえる）
- × 乱数を何度も引く可能性がある。

Abseilの最適化

Abseilの最適化

- Abseilは、exponent+mantissa法の簡略版。
  - 単一の64bit乱数で処理を完結させている。

- exponent決定時、反復回数を63回に制限（＝乱数のbit数に制限）。
  - 何度も乱数を引かない、ということ。exponentの範囲が狭くなり、出力される最小値が大きくなる。
  - といっても、出力される最小値は5.42101086e-20なので、そこまで悪くない。
- mantissa決定時、exponent決定時に余ったbitを流用。
  - exponent次第で、mantissaが常に23bit使い切らない。
  - といっても、非常に小さい値の範囲での話なので、そこまで問題にならない。

Abseilの最適化

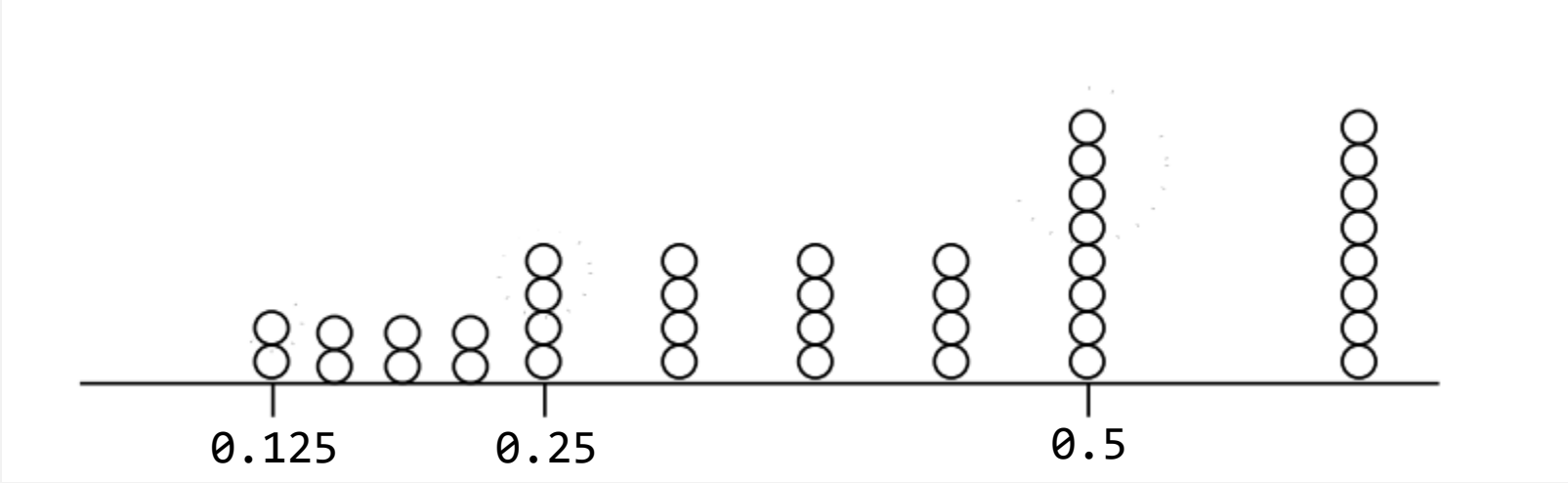
- ○ [0.0, 1.0) が保証される。
- ○ doubleにも自然に拡張可能。
- ○ 乱数を何度も引かなくて済む。
- × floatを得るのにも64bit乱数が必要。
- × [0.0, 1.0) の範囲における、表現可能な全てのfloat値を得ることが出来ない。（除算法やmantissa法に比べれば、マシ）
- × 実装がやや複雑。（実行速度も不利になりえる）

Downeyの方法

Downeyの方法

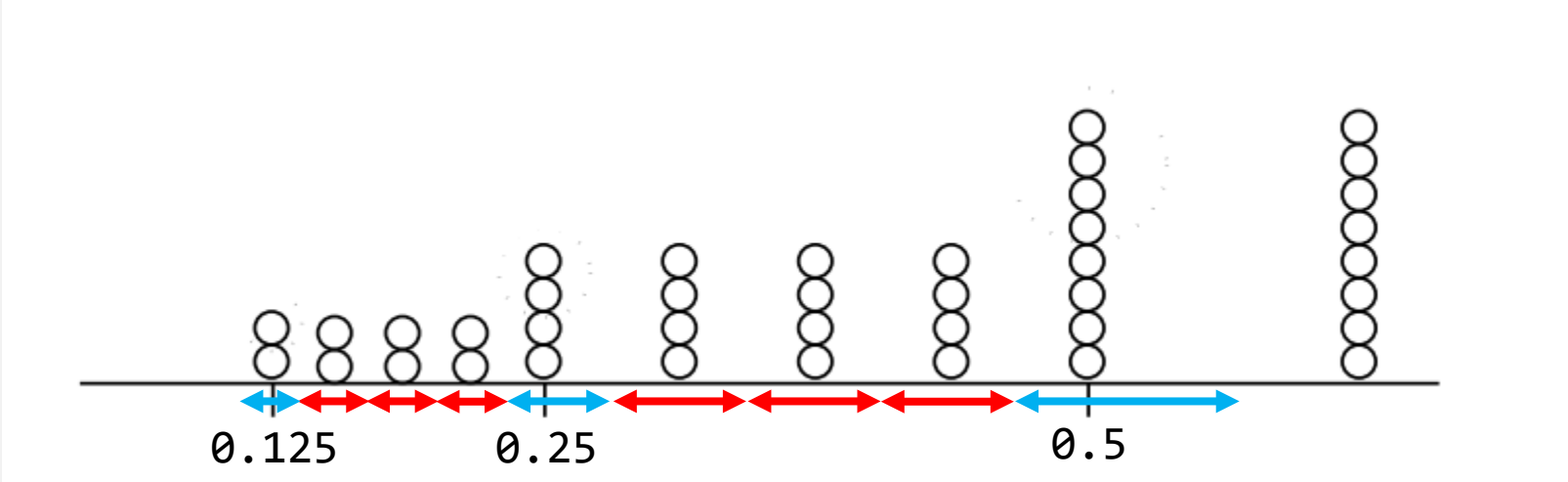
- exponent+mantissa法には確率に関する問題がある。
  - 実は、境界の値の生成確率が正しくない。

- 以下は、各値が得られる確率のヒストグラム（概念図）。
  - 0.125、0.25、0.5といった境界の値は本来もっと確率が低い。



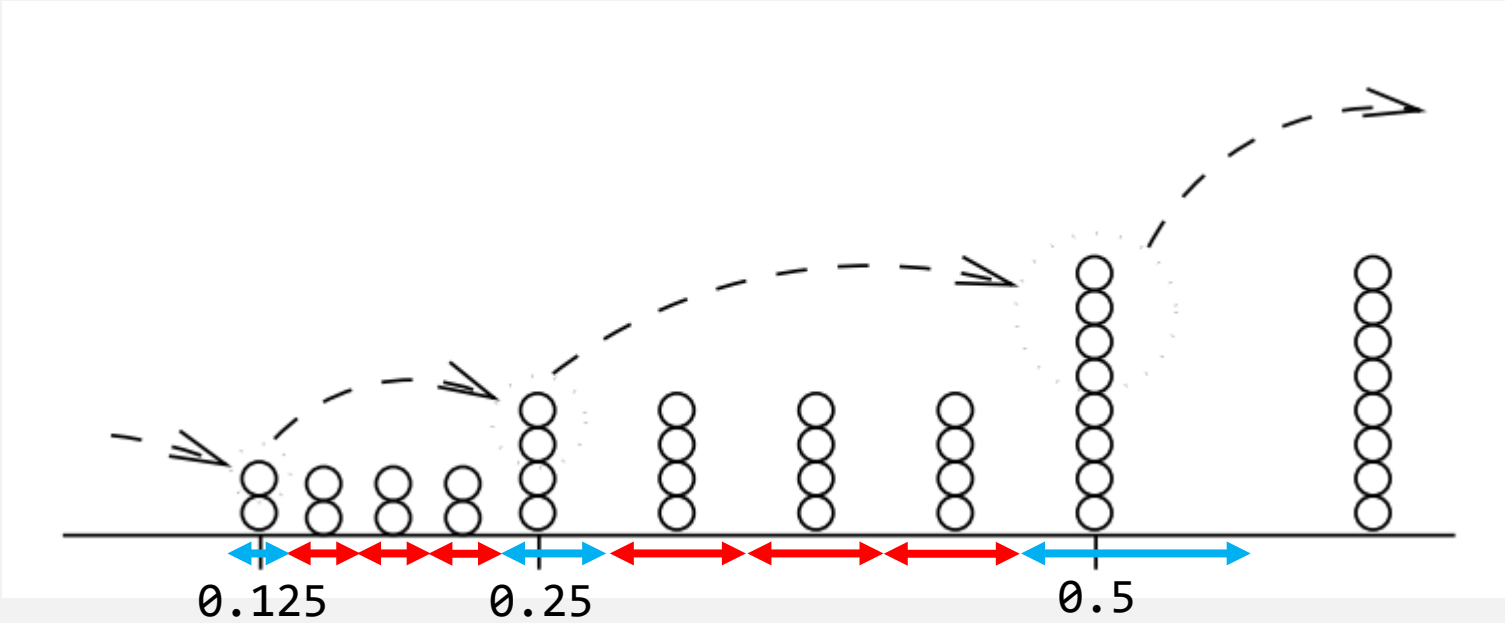
Downeyの方法

- 以下のように、境界の値はカバーする範囲が若干狭い。
- [0.25, 0.5)の範囲において、0.25は他の値の3/4倍の範囲しかカバーしておらず、得られる確率もそれに比例すべき。
  - 赤：通常の値のカバー範囲
  - 青：境界の値のカバー範囲



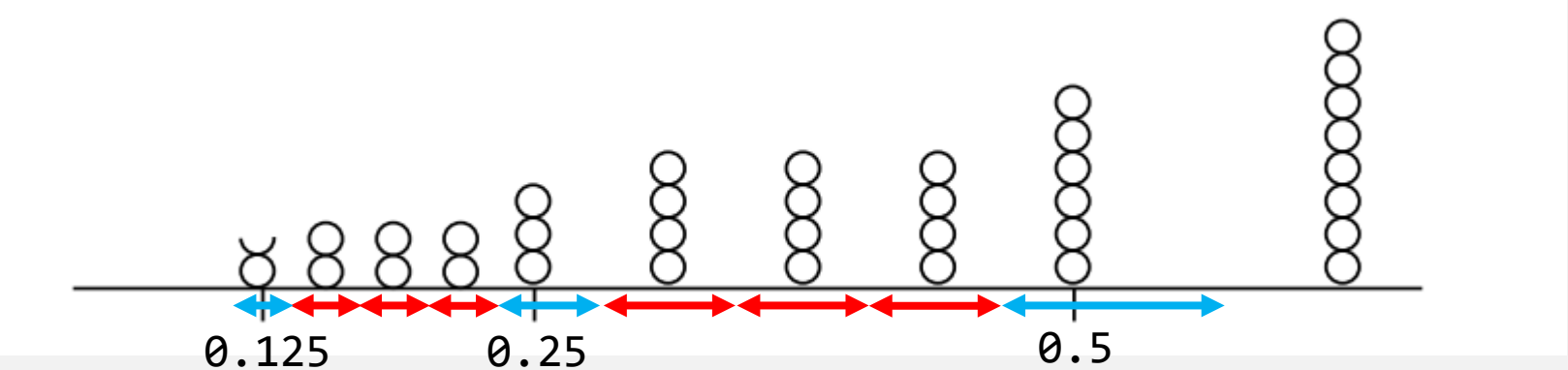
Downeyの方法

- そこで、境界の値が得られた場合、1/2の確率でexponentをインクリメントする。
  - つまり、次の区間の境界に移動する。0.25の場合、0.5に移動する。



Downeyの方法

- すると、ヒストグラムは以下のようになり、期待通りの確率分布になる。



Downeyの方法

- exponent+mantissa法に、以下のexponentインクリメント処理を追加すればOK。
  - mantissaがゼロの時が、境界の値。

```
uint32_t mantissa;
mantissa = rng() & 0x7fffff;
if (mantissa == 0 && provider.getBit()) {
    ++exponent;
}
```

- Generating Pseudo-random Floating-Point Values, Allen B. Downey 2007 においてこの方法が提案された。

Downeyの方法

- 全てをまとめるとこのようになる。
  - 1億回の実行に 1.21 [sec]

```
template·<typename·rng_t>·float·rand01_Downey(rng_t·&rng)·{
··RandomBitProvider·provider(rng);

··constexpr·float·lowV·=·0.0f;
··constexpr·float·highV·=·1.0f;
··const·uint32_t·lowExp·=·((bit(lowV)·>>·23)·&·0xff);
··const·uint32_t·highExp·=·((bit(highV)·>>·23)·&·0xff);

··uint32_t·exponent;
··for·(exponent·=·highExp·-·1;·exponent·>·lowExp;·--exponent)·{
····if·(provider.getBit())·{
····  break;
····}
··}

··uint32_t·mantissa;
··mantissa·=·rng()·&·0x7fffff;
··if·(mantissa·==·0·&&·provider.getBit())·{
····++exponent;
··}

··return·bitToFloat((exponent·<<·23)·|·mantissa);
}
```

Downeyの方法

- 1bitずつ乱数を得る部分がボトルネックになるので、右のように最適化することができる。
  - 乱数を引く回数を減らせる
  - まだ最適化の余地あるかも？

- 1億回の実行に 0.34 [sec]

```
template·<typename·rng_t>·float·rand01_Downey_opt(rng_t·&rng)·{
··constexpr·int32_t·lowExp·=·0;
··constexpr·int32_t·highExp·=·127;

··const·uint32_t·u·=·rng();
··const·uint32_t·b·=·u·&·0xff;
··int32_t·exponent·=·highExp·-·1;
··if·(b·==·0)·{
····exponent·-=·8;
····while·(1)·{
·····const·uint32_t·bits·=·rng();
·····if·(bits·==·0)·{
·····exponent·-=·32;
·····if·(exponent·<·lowExp)·{
·····exponent·=·lowExp;
·····break;
·····}
·····}·else·{
·····exponent·-=·ctz(bits);
·····break;
·····}
····}
··}·else·{
····exponent·-=·ctz(b);
··}

··const·uint32_t·mantissa·=·(u·>>·8)·&·0x7fffff;
··if·(mantissa·==·0·&&·(u·>>·31))·{
····++exponent;
··}

··return·bitToFloat((exponent·<<·23)·|·mantissa);
}
```

Downeyの方法

- Downeyの方法で得られるのは[0.0, 1.0]の範囲の乱数になる。
- [0.0, 1.0)や(1.0, 0.0]を得るには以下のように再生成する。
  - 再生成が起こる確率は極めて低いので大きな問題にはならないとは思われるが、無限ループを避ける工夫も入れた方がいいかもしれない。
  - クランプでも良いが、確率がバイアスされる。
- 他の手法にも適用可能。

```
template·<typename·rng_t>·float·rand01_Downey_rightopen(rng_t&·rng)·{
····float·f;
····do·{
····    ····f·=·rand01_Downey(rng);
····}·while·(f·==·1.0f);
····return·f;
}

template·<typename·rng_t>·float·rand01_Downey_leftopen(rng_t&·rng)·{
····float·f;
····do·{
····    ····f·=·rand01_Downey(rng);
····}·while·(f·==·0.0f);
····return·f;
}
```

Downeyの方法まとめ

- ○ doubleにも自然に拡張可能。
- ○ [0.0, 1.0] の範囲における、表現可能な全てのfloat値を得ることが出来る！しかも生成確率が正しい。
- × [0.0, 1.0)の範囲が保証されない。
- × 実装がやや複雑。（最適化すればそこそこ高速）
- × 乱数を何度も引く可能性がある。（ただし、32bitより多く必要になる確率は1/256で、最大でも160bit）

結論

結論

- なんだかんだ普通に除算法するのでもあまり問題にならない。
  - パフォーマンスは良い。表現可能なfloat値の全ては得られない。
  - [0.0, 1.0)を保証するかどうかなどについても考えておく必要がある。

- [0.0, 1.0)を保証しつつ、表現可能なfloat値の全てを得たいなら、exponent+mantissa法やDowneyの方法もあり得る。
  - ただし、乱数を何度も引く可能性などについて考えておく必要がある。
  - Abseilの最適化が良いバランスになることもあり得る。

- 目的やパフォーマンス、得られる乱数のbit数（と乱数生成速度）などに応じて、最適な選択がきまる、かもしれない。

おまけ：std::uniform_real_distributionの問題

- std::uniform_real_distributionを使うことで、任意の範囲 [a, b) の一様分布を得られる。
- が、与える値次第で容易に区間に関する条件を満たさなくなる。
  - つまり、bが返ってしまうことがある。
  - [0.0, 1.0)の乱数を用いても、丸めによってbになりえる。

- 一般的に解決するにはおそらく難しく、再生成する、クランプする、などしかないかもしれない。
  - 世間の実装も概ねそのように対処しているか、そもそも対処していないか。

- ただし、得たい値がfloatならdouble除算法の結果を最後にキャストするようにすればある程度精度向上を図れる（今どきなら、パフォーマンスもあまり変わらない）