2011 年 4 月 15 日 (2018 年 4 月 24 日修正)

IEEE754 浮動小数点数の隣の数を計算する方法のまとめ

柏木 雅英

1 はじめに

精度保証付き数値計算において、与えられた浮動小数点数の「隣の数」(それより大きな最小の浮動小数点数、またはそれより小さな最大の浮動小数点数) の計算が必要になることがよくある。例えば、区間演算において、丸めの向きの変更が難しい場合は、普通に計算した後、下限を下向きに、上限を上向きに「ずらす」場合などに使われる。`Intlab` では、`succ`、`pred` という名前の関数でこの機能が提供されている。successor、predecessor の略。

いろいろな方法が考えられるが、CPU や OS、言語環境の違いにより一長一短がありこれが決定版である、と言いづらいので、とりあえず思い付く限りの方法を列挙してみる。

以下、浮動小数点数は、IEEE 754 Std. の倍精度とする。プログラムは python 風に記述した。

2 丸めの向きを変える

丸めの向きの変更が許されるなら、

```
def succ(x) :
    上向き丸めに変更
    r = x + 2**(-1074)
    (丸めモードを元に戻す)
    return r

def pred(x) :
    下向き丸めに変更
    r = x - 2**(-1074)
    (丸めモードを元に戻す)
    return r
```

のような方法が考えられる。堅実でシンプルだが、

- 丸めモードの変更は言語によっては困難なことがある。実行コストが高い場合がある。
- 2^{-1074} は非正規化数であり、非正規化数を使った計算は実行コストが高い場合がある。

のような欠点がある。

3 除算を用いる方法

以下のような方法である。

```
def abssucc(x) :
    return x / (1 - 2**(-53))

def abspred(x) :
    return x * (1 - 2**(-53))

def succ(x) :
    if abs(x) < 2**(-1022) :
        return x + 2**(-1074)
    if x >= 0 :
        return abssucc(x)
    else :
        return abspred(x)

function pred(x) {
    if abs(x) < 2**(-1022)
        return x - 2**(-1074)
    if x >= 0 :
        return abspred(x)
    else :
        return abssucc(x)
```

非正規化数を別扱いにしている点を除けば、非常にシンプルな方法である。欠点はとしては、除算の計算コストが高い点が挙げられる。

また、コンパイラによっては「定数による除算」を「その逆数の乗算」に置き換える最適化を行う場合があり、この最適化をされてしまうと正常に動作しなくなる。

$$\frac{1}{1-2^{-53}} = 1 + 2^{-53} + 2^{-106} + 2^{-159} + \cdots$$

であり、この数に最も近い浮動小数点数は $1 + 2^{-52}$ である。これでは、2ulp 上がってしまうことがある。

4 柏木法?

次のような方法である。$x \pm |x| * c$ の型の補正をなるべく使いたいという方針。

```
def succ(x) :
    a = abs(x)
    if a > 2**(-1020) :
```

```
        return x + a * (2**(-53)+2**(-55))
    if -2**(-1021) <= x and x < 2**(-1021) :
        return x + 2**(-1074)
    if x < 0 or x < 2**(-1020) :
        return x + 2**(-1073)
    return x + 2**(-1072)

def pred(x) :
    a = abs(x)
    if a > 2**(-1020):
        return x - a * (2**(-53)+2**(-55))
    if -2**(-1021) < x and x <= 2**(-1021) :
        return x - 2**(-1074)
    if x > 0 or x > - 2**(-1020) :
        return x - 2**(-1073)
    return x - 2**(-1072)
```

欠点としては、

- 複雑である。
- 非正規化数を多用するため、非正規化数の実行コストが高い環境では不利。

が挙げられる。

5 Rumpによる方法

S. M. Rump, P. Zimmermann, S. Boldo and G. Melquiond: "Computing predecessor and successor in rounding to nearest", BIT Vol. 49, No. 2, pp.419–431, 2009 (http://www.ti3.tu-harburg.de/paper/rump/RuZiBoMe08.pdf) による方法。

```
def succ(x) :
    a = abs(x)
    if a >= 2**(-969) :
        return x + a * (2**(-53)+2**(-105))
    if a < -2**(-1021) :
        return x + 2**(-1074)
    c = 2**53 * x
    e = (2**(-53)+2**(-105)) * abs(c)
    return (c + e) * 2**(-53)

def pred(x) :
    a = abs(x)
```

```
    if a >= 2**(-969) :
        return x - a * (2**(-53)+2**(-105))
    if a < -2**(-1021) :
        return x - 2**(-1074)
    c = 2**53 * x
    e = (2**(-53)+2**(-105)) * abs(c)
    return (c - e) * 2**(-53)
```

複雑だが、なるべく非正規化数の演算を避けるように設計されている。

6 bit 操作

C 言語の union 型を用いて、直接 bit 操作をする方法。

```
double abssucc(double x) {
    union {
        double d;
        unsigned long long int i;
    } u;
    u.d = x;
    u.i++;
    return u.d;
}

double abspred(double x) {
    union {
        double d;
        unsigned long long int i;
    } u;
    u.d = x;
    u.i--;
    return u.d;
}

double succ(double x) {
    if (x > 0.) return abssucc(x);
    else if (x < 0.) return abspred(x);
    else return abssucc(+0.);
}

double pred(double x) {
    if (x > 0.) return abspred(x);
```

```
    else if (x < 0.) return abssucc(x);
    else return abssucc(-0.);
}
```

非常に高速だが、C 言語以外で実装するのは困難である。また、無限大や NaN の値も変えてしまうので、それらもきちんと扱おうとすると更に if が増え、速度が低下する。

なお、上の実装だと非正規化数が入力されたときに遅くなるが、次のように全て int の世界で済ませれば更に速くなる。

```
double succ(double x) {
    union {
        double d;
        unsigned long long int i;
    } u;
    u.d = x;
    if (u.i<<1 == 0) u.i = 1;
    else u.i += 1 - 2*(u.i>>63);
    return u.d;
}

double pred(double x) {
    union {
        double d;
        unsigned long long int i;
    } u;
    u.d = x;
    if (u.i<<1 == 0) u.i = ((unsigned long long)1<<63) + 1;
    else u.i -= 1 - 2*(u.i>>63);
    return u.d;
}
```

7 nextafter

C99 では、nextafter という関数が提供される。nextafter(x, y) は、x から見て y に近付く方向の、x の隣の浮動小数点数を返す。すなわち、

```
#include <math.h>
double succ(double x) {
    return nextafter(x, INFINITY)
}
double pred(double x) {
    return nextafter(x, -INFINITY)
}
```

でよい。明確に C99 準拠を謳っていないコンパイラでも、この関数を持っていることが多い。しかし、C 言語以外で使うのは困難である。

内部の実装は、調査したところ前述の bit 操作を用いたものが多いようである。

8 速度比較

これらの手法の速度に関心があるところだが、これは一筋縄では行かない。単にステップ数を数えればいいというものでは無く、

- 丸めの向きの変更によるペナルティ
- 除算のペナルティ
- `if` 文 (条件分岐) によるペナルティ
- 非正規化数の演算によるペナルティ

などを考慮する必要がある。コンパイラや CPU の種類によってこれらのペナルティの大きさは大きく異なるので、究極的には計算する環境で実測するしか無い。