最近点丸めによる方向付き丸めのエミュレート

柏木 雅英 [1)]
1) 早稲田大学 理工学術院 基幹理工学部 応用数理学科

概要

本稿では、最近点丸めのみを用いて、上方及び下方へ丸めた加算、減算、乗算、除算、平方根をエミュレートする方法を示す。これにより、ハード的な丸めモードの変更を行うことなく区間演算が実現できる。

Emulation of Rounded Arithmetic in Rounding to Nearest

Masahide Kashiwagi[1)]
1) Waseda University

Abstract

In this report, we give a method to emulate rounded arithmetic (addition, subtraction, multiplication, division, square root) using only floating point operations in rounding to nearest. Using this method, we can implement interval arithmetic without hardware change of rounding mode.

1 はじめに

区間演算においては、区間の両端の値を計算する際に丸めの向きを「外向き」にしておくことによって丸め誤差の影響分を区間内に収め、丸め誤差の把握を行う。例えば、区間 $X = [\underline{X}, \overline{X}]$、$Y = [\underline{Y}, \overline{Y}]$ の和 $Z = [\underline{Z}, \overline{Z}]$ は、丸めを $-\infty$ 方向に変更してから $\underline{X} + \underline{Y}$ を計算したものを $\underline{Z}$ 、丸めを $+\infty$ 方向に変更してから $\overline{X} + \overline{Y}$ を計算したものを $\overline{Z}$ とする。IEEE 754 Std. の規格では、加減乗除と平方根についてこのような丸めモードの変更が可能である。

しかし、GPU やスーパーコンピュータなどの特殊な環境の場合、丸めモードを変えにくい場合がある。また、丸めモードの変更を行う機能が用意されていない言語も多数存在する。よって、プログラムのポータビリティを考えると、丸めモードの変更を行わずに区間演算を行えることが望ましい。

本稿では、最近点丸め (デフォルトの丸めモード) のみを用いて、丸めの向きを変更した加算、減算、乗算、除算、平方根 をエミュレートする方法を示す。原理は簡単だが、全ての倍精度浮動小数点数に対してこれを実現するのは案外難しい。

2 準備

本手法に必要な、twosum, twoproduct, succ, pred アルゴリズムについて説明する。

twosum[1] は、python 風に書くと以下の通り。

```
def twosum(a, b) :
    x = a + b
    tmp = x - a
    y = (a - (x - tmp)) + (b - tmp)
    return x, y
```

a と b から $x, y = \mathrm{twosum}(a, b)$ を計算すると、計算の前後で $x + y = a + b$ が成立する。また、x は $a + b$ を浮動小数点数で計算したものに等しい。すなわち、$a + b$ を数学的に厳密に計算したものを上位 x と下位 y に分解したものと見ることが出来る。

twoproduct[2] は、以下の通り。

```
def split(a) :
    tmp = a * (2.**27 + 1)
    x = tmp - (tmp - a)
    y = a - x
    return x, y

def twoproduct(a, b) :
    x = a * b
    a1, a2 = split(a)
    b1, b2 = split(b)
    y = a2 * b2 - (((x - a1 * b1) - a2 * b1)
         - a1 * b2)
    return x, y
```

a と b から $x, y = \mathrm{twoproduct}(a, b)$ を計算すると、計算の前後で $x + y = a \times b$ が成立する。split は倍精度浮動小数点数を上位 bit と下位 bit に分けるための補助的な関数である。

succ, pred は、IEEE 754 Std. の浮動小数点数 x に対してその隣 (x より大きな最小の浮動小数点数または x より小さな最大の浮動小数点数) を計算する。実現方法はたくさんあるが、その一例として Rump による方法 [3] を示しておく。

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

3 エミュレートの基本原理

方向付き丸めのエミュレートは、次のような簡単な原理に基づいている。例えば、加算は以下の通り。

```
def add_up(a, b) :
    x, y = twosum(a, b)
    if y > 0.:
        x = succ(x)
    return x

def add_down(a, b) :
    x, y = twosum(a, b)
    if y < 0.:
        x = pred(x)
    return x
```

twosum によって $a+b$ の計算に発生した誤差を y として拾うことが出来るので、y が正なら x は真値より小さめに、y が負なら x は真値より大きめに、$y=0$ なら x が無誤差で計算されたことが分かる。

減算は b を $-b$ とすることによって、乗算は twosum を twoproduct に置き換えることによって加算とほぼ同様に実現できる。

除算、平方根は、twoproduct を応用して次のように実現できる。(up のみ示す)

```
def div_up(a, b) :
    if b < 0 :
        an, bn = -a, -b
    else:
        an, bn = a, b
    d = an / bn
    x, y = twoproduct(d, bn)
    if x < an or (x == an and y < 0.):
        return succ(d)
    return d
```

```
def sqrt_up(a) :
    d = math.sqrt(a)
    x, y = twoproduct(d, d)
    if x < a or (x == a and y < 0.):
        d = succ(d)
    return d
```

4 twosum, twoproduct の限界

twosum や twoproduct は無誤差変換を謳っているが、それはオーバーフローやアンダーフローが発生しない、指数部は $-\infty \sim \infty$ を取れるという仮定の下で証明されている。実際の IEEE 754 Std. の倍精度数は当然指数部に制限があるので、常に無誤差というわけではない。

4.1 twosum の限界

twosum は、IEEE 754 Std. に備わったいわゆる非正規化数のおかげで、アンダーフローは発生しないが、オーバーフローからは逃れられない。 また、ごく稀ではあるが次のような例もある。

```
>>> twosum(3.5630624444874539e+307,
    -1.7976931348623157e+308)
(-1.4413868904135704e+308, nan)
```

この例だと、$a+b$ は (ギリギリで) オーバーフローしないが、中間変数 `tmp` がオーバーフローしてしまう。

中間変数のオーバーフローを避けるため、fasttwosum を絶対値の大きさで切り替えて使う方法を採用する。

```
def twosum(a, b) :
    x = a + b
    if (abs(a) > abs(b)):
        tmp = x - a
        y = b - tmp
    else:
        tmp = x - b
        y = a - tmp
    return x, y
```

4.2 twoproduct の限界

twoproduct は、アンダーフロー、オーバーフローともに注意が必要である。 詳細は省略するが、$x, y = \mathrm{twoproduct}(a, b)$ を計算したとき、$|x| > 2^{-969} - 2^{-1021}$ であれば twoproduct は無誤差であるが、そうでなければ一般に無誤差であるとは言えない。

また、split で $2^{27}+1$ を乗じている部分でオーバーフローを起こす可能性もある。更に、非常に特殊な場合ではあるが、$a \times b$ はオーバーフローしないが $a1 \times b1$ がオーバーフローする場合がある。

```
>>> twoproduct(6.929001713869936e+236,
    2.5944475251952003e+71)
(1.7976931348623157e+308, inf)
```

split と $a1 \times b1$ の問題に対して対策を行った結果、twoproduct は次のように改変することにした。

```
def twoproduct(a, b) :
    x = a * b
    if abs(a) > 2.**996:
        an = a * 2.**(-28)
        bn = b * 2.**28
    elif abs(b) > 2.*996:
        an = a * 2.**28
        bn = b * 2.**(-28)
    else:
        an = a
        bn = b
    a1, a2 = split(an)
    b1, b2 = split(bn)
    if abs(x) > 2.**1023:
        y = a2 * b2 - ((((x * 0.5) - (a1 *
            0.5)  * b1) * 2. - a2 * b1) - a1
```

```
            * b2)
    else:
        y = a2 * b2 - (((x - a1 * b1) - a2 *
            b1) - a1 * b2)
    return x, y
```

5 対策後のプログラム

無限大に関して、次の点に注意が必要である。下向き丸めで計算結果が ∞ になった場合、例えば $\infty+1$ のように、初めから入力に ∞ が含まれているならば、そのままでよい。しかし、$10^{308}+10^{308}$ のようにオーバーフローした場合は、計算結果は ∞ ではなく正の浮動小数点数の最大値 $(2^{1024}-2^{971})$ でなければならない。

ここまでに述べてきた問題点全て対策を施したプログラムを以下に示す。

```
def add_up(a, b) :
    x, y = twosum(a, b)
    if x == float("inf"):
        return x
    elif x == -float("inf"):
        if a == -float("inf") or b == -float
            ("inf"):
            return x
        else:
            return -sys.float_info.max
    if y > 0.:
        x = succ(x)
    return x

def add_down(a, b) :
    x, y = twosum(a, b)
    if x == float("inf"):
        if a == float("inf") or b == float("
            inf"):
            return x
        else:
            return sys.float_info.max
    elif x == -float("inf"):
        return x
    if y < 0.:
        x = pred(x)
    return x
```

(減算はスペースの都合上省略。)

```
def mul_up(a, b) :
    x, y = twoproduct(a, b)
    if x == float("inf"):
        return x
    elif x == -float("inf"):
        if abs(a) == float("inf") or abs(b)
            == float("inf"):
            return x
        else:
            return -sys.float_info.max

    if (abs(x) >= 2.**(-969)):
        if y > 0.:
            return succ(x)
        return x
    else:
        s, s2 = twoproduct(a * 2.**537, b *
            2.**537)
        t = (x * 2.**537) * 2.**537
        if t < s or (t == s and s2 > 0):
            return succ(x)
        return x

def mul_down(a, b) :
    x, y = twoproduct(a, b)
    if x == float("inf"):
        if abs(a) == float("inf") or abs(b)
            == float("inf"):
            return x
        else:
            return sys.float_info.max
    elif x == -float("inf"):
        return x

    if (abs(x) >= 2.**(-969)):
        if y < 0.:
            return pred(x)
        return x
    else:
        s, s2 = twoproduct(a * 2.**537, b *
            2.**537)
        t = (x * 2.**537) * 2.**537
        if t > s or (t == s and s2 < 0):
            return pred(x)
        return x
```

```
def div_up(a, b) :
    if a == 0 or b == 0 or abs(a) == float("
        inf") or abs(b) == float("inf") or a
         != a or b != b:
        return a / b

    if b < 0 :
        an, bn = -a, -b
    else:
        an, bn = a, b

    if abs(an) < 2.**(-969):
        if abs(bn) < 2.**918:
            an *= 2.**105
            bn *= 2.**105
        else:
            if an < 0.:
                return 0.
            else:
                return 2.**(-1074)

    d = an / bn

    if d == float("inf"):
        return d
    elif d == -float("inf"):
        return -sys.float_info.max

    x, y = twoproduct(d, bn)
    if x < an or (x == an and y < 0.):
        return succ(d)
    return d

def div_down(a, b) :
```

```
    if a == 0 or b == 0 or abs(a) == float("
        inf") or abs(b) == float("inf") or a
         != a or b != b:
        return a / b

    if b < 0 :
        an, bn = -a, -b
    else:
        an, bn = a, b

    if abs(an) < 2.**(-969):
        if abs(bn) < 2.**918:
            an *= 2.**105
            bn *= 2.**105
        else:
            if an < 0.:
                return -2.**(-1074)
            else:
                return 0.

    d = an / bn

    if d == float("inf"):
        return sys.float_info.max
    elif d == -float("inf"):
        return d

    x, y = twoproduct(d, bn)
    if x > an or (x == an and y > 0.):
        return pred(d)
    return d
```

```
def sqrt_up(a) :
    d = math.sqrt(a)

    if a < 2.**(-969):
        a2 = a * 2.**106
        d2 = d * 2.**53
        x, y = twoproduct(d2, d2)
        if x < a2 or (x == a2 and y < 0.):
            d = succ(d)
        return d

    x, y = twoproduct(d, d)
    if x < a or (x == a and y < 0.):
        d = succ(d)
    return d

def sqrt_down(a) :
    d = math.sqrt(a)

    if a < 2.**(-969):
        a2 = a * 2.**106
        d2 = d * 2.**53
        x, y = twoproduct(d2, d2)
        if x > a2 or (x == a2 and y > 0.):
            d = pred(d)
        return d

    x, y = twoproduct(d, d)
    if x > a or (x == a and y > 0.):
        d = pred(d)
    return d
```

6 数値実験

実験環境は、CPU: core i7 2640M (2.8GHz)、OS: ubuntu 10.04 LTS (64bit)、software: gcc 4.4。

丸めの変更を行った場合と同じ結果になるかどうか数値実験してみた。全ての場合を尽くすには $2^{64} \times 2^{64}$ 通りのテストを行う必要があり、現実的には不可能である。そこで、

- 64bit 整数の乱数を発生し、その bit pattern を double に読み替えたものを 10^{10} 個。
- $\pm\infty$, ± 0, $\pm(2^{1024}-2^{971})$(正負の最大数), $\pm 2^{-1022}$, $\pm 2^{-1074}$, NaN などの特殊数

を用いて加減乗除と平方根を計算し、丸めの変更を行った場合と同じ結果になることを確認した。

次に、速度評価を行う。C99 準拠なコンパイラが備える `fe_setround` を用いた次のようなプログラムと本稿のプログラムとで速度を比較した。

```
#include <fenv.h>
double add_up(const double& x, const double&
     y) {
    volatile double r, x1 = x, y1 = y;
    fesetround(FE_UPWARD);
    r = x1 + y1;
    fesetround(FE_TONEAREST);
    return r;
}
```

先の実験と同様に乱数を発生し、10^{10} 回計算した場合の平均速度を示す。

	加算	減算	乗算	除算	平方根
丸めモード変更	13.2ns	13.2ns	14.1ns	14.5ns	13.3ns
エミュレート	1.2ns	1.2ns	13.4ns	21.7ns	7.6ns

これを見ると、除算では遅くなるものの、加減算では 10 倍程度、乗算は同程度、平方根で 2 倍程度速いことが分かる。実アプリケーションでの速度比較は発表時に示す。また、本発表で用いた全てのプログラムは `http://verifiedby.me/`で公開する。

参考文献

[1] Donald E. Knuth: “The Art of Computer Programming Volume 2: Seminumerical Algorithms”, Addison-Wesley, 1969

[2] T. J. Dekker: “A Floating-Point Technique for Extending the Available Precision”, Numerische Mathematik, 18, pp.224–242, 1971

[3] 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