(Version: 2014/6/9)

最近点丸めのみによる方向付き丸めのエミュレート

柏木 雅英

1 はじめに

数値計算において、計算を行うと同時にその結果の誤差評価をも同時に計算するような方法を総称して精度保証付き数値計算と呼び、近年急速な進歩を遂げている。

精度保証付き数値計算の実現において最も基本的かつ重要な技法に、区間演算が挙げられる。区間演算とは、実数値を [下限, 上限] という 2 つの浮動小数点数で挟まれた区間で表現し、その区間同士の加減乗除等の演算を「演算結果として有り得る集合を包含するように」定義することにより行われるものである。そのとき、区間の両端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ことが出来る。例えば、区間 $X = [\underline{X}, \overline{X}]$、$Y = [\underline{Y}, \overline{Y}]$ の和 $Z = [\underline{Z}, \overline{Z}]$ は、丸めモードを $-\infty$ 方向に変更してから $\underline{X} + \underline{Y}$ を計算したものを $\underline{Z}$ 、丸めモードを $+\infty$ 方向に変更してから $\overline{X} + \overline{Y}$ を計算したものを $\overline{Z}$ とすることにより得られる。

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を行えること要請している。従って、現在出回っているほとんど全ての CPU で丸めモードの変更を行うことが出来る。C99 準拠の C コンパイラが使えれば、標準的な丸めモード変更の手段 (`fenv.h` & `fesetround`) が提供されている。しかし、GPU やスーパーコンピュータなどの特殊な環境においては、丸めモードを変えにくい場合がある。更に、Java や C#などの仮想マシンベースの言語、python や javascript などのスクリプト言語では、多くの場合丸めモードの変更を行う機能が用意されておらず、また外部プログラム呼び出しでの丸めモードの変更も許されていないことがある。また、コンパイラの最適化が丸めモードの変更を考慮していないため、注意してプログラムを作成しないと目的の演算の丸めの向きがうまく変わらないこともある。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 (default 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、twosum や twoproduct の制限により、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。本稿では、全ての例外に対応した (つもりの) 方法を示す。

2 twosum, twoproduct, succ, pred

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 fasttwosum(a, b) :
    x = a + b
    tmp = x - a
    y = b - tmp
    return x, y

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 $x + y = a + b$ が数学的に厳密に成立する。また、x は $a + b$ を浮動小数点演算で計算したものに等しく、$|y|$ は $|x|$ に対して「$x + y$ を浮動小数点演算で計算すると x になる」程度に小さい。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。また、x と y を、$a + b$ の計算結果とその誤差、と見ることも出来る。

なお、fasttwosum は、twosum より計算量が小さいが、$|a| \geq |b|$ の場合にしか正しく動作しない。

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 $x + y = a \times b$ が数学的に厳密に成立する。また、x は $a \times b$ を浮動小数点数で計算したものに等しく、$|y|$ は $|x|$ に対して「$x + y$ を浮動小数点演算で計算すると x になる」程度に小さい。

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。なお、もし Fused Multiply Add(FMA) ($ab - c$ を計算し、最近点に丸める) が使えるなら、twoproduct は以下のように簡単に書ける。

```
def twoproductfma(a, b) :
    x = a * b
    y = fma(a, b, -x)
    return x, y
```

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アイデア

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。よって、上向き丸めの加算は、$y \leq 0$ ならそのまま x を、$y > 0$ なら $\mathrm{succ}(x)$ を返すことによって得られる。

減算、加算とほぼ同様。

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

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

乗算も加算とほぼ同様。twoproduct を使うことにより誤差を知ることが出来る。

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

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

除算は少し異なる。$a \div b$ を計算するのに、まず b が負なら a と b の符号を反転して b を正にしておく。次に、$a \div b$ を計算した結果を d とし、これと b を twoproduct を用いて乗ずる。これは除算が無誤差なら a に戻るはずの量。よって、x と a が異なるならそれで d が大きめか小さめか分かり、$x = a$ ならば y の正負で d が大きめか小さめか分かる。

```
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 div_down(a, b) :
    if b < 0 :
        a, b = -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 pred(d)
    return d
```

平方根は除算とほぼ同様。$\sqrt{a}$ の計算値を d とし、d と d の積を twoproduct で計算する。後はこれと a を除算と同様に比較すればよい。

```
import math

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

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

しかし、後述する twosum, twoproduct の適用限界などで、これで全ての値についてうまく行くわけでは無い。

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、すなわち仮数部の長さは限られているが、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 を以下に再掲する。

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

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

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。しかし、オーバーフローからは逃れられない。オーバーフローを起こすと、

```
>>> twosum(1e308, 8e307)
(inf, nan)
```

のように y が NaN になってしまう点にも注意が必要。

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には、オーバーフロー寸前の場合に限って適当な 2 のベキでスケーリングする方法が考えられるが、ここではいわゆる 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
```

これなら中間変数がオーバーフローすることはない。先の例だと、

```
>>> twosum(3.5630624444874539e+307, -1.7976931348623157e+308)
(-1.4413868904135704e+308, 9.9792015476735991e+291)
```

のように正しく計算できる。

4.2 twoproduct の限界

twoproduct を以下に再掲する。

```
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
```

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。オーバーフローは特殊な数 (inf) になるので検出しやすいが、アンダーフローは黙って誤差が発生するので特に要注意。$x, y = \text{twoproduct}(a, b)$ を計算したとき、x がアンダーフローを起こしていなくても y がアンダーフローを起こして結果として無誤差が保たれないことがある。具体的には、a の最下位 bit を 2^{la}, b の最下位 bit を 2^{lb} としたとき、$la + lb \leq -1075$ ならば、この bit の情報が抜け落ちてしまう。このような数のうち積の絶対値が最大となるものの一つを示す。($la + lb = -1075$ で全 bit1 にした。)

$$
\begin{aligned}
a &= \sum_{i=-538}^{-486} 2^i = 2^{-485} - 2^{-538} \\
b &= \sum_{i=-537}^{-485} 2^i = 2^{-484} - 2^{-537}
\end{aligned}
$$

この積は

$$2^{-969} - 2^{-1021} + 2^{-1075}$$

であり、twoproduct の結果は、

$$
\begin{aligned}
x &= 2^{-969} - 2^{-1021} \\
y &= 2^{-1075}
\end{aligned}
$$

となるべきであるがアンダーフローを起こして

$$
\begin{aligned}
x &= 2^{-969} - 2^{-1021} \\
y &= 0
\end{aligned}
$$

となってしまう。逆に言えば、

$$|x| > 2^{-969} - 2^{-1021}$$

であれば twoproduct でアンダーフローは起きなかったと言える。

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。これは、例えば $|a|$ と $|b|$ の片方が 2^{996} より大きい場合、大きい方に 2^{-28} を、小さい方に 2^{28} を乗ずることにより容易に回避できる。これが出来ない場合はそもそも結果がオーバーフローするはず。

更に、非常に特殊な場合ではあるが、計算結果 $a \times b$ はオーバーフローしないが中間結果 $a1 \times b1$ 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)
```

これに関しては、$|x| \geq 2^{1023}$ と非常に大きい場合に限って $x - a1 \times b1$ を $x/2 - (a1/2) \times b1$ と計算することで対策した。

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

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
```

これにより、先の例は、

```
>>> twoproduct(6.929001713869936e+236, 2.5944475251952003e+71)
(1.7976931348623157e+308, -1.0027614963959625e+291)
```

ときちんと計算できるようになる。

twoproduct については、アンダーフローが起きたときの誤差の問題は解決していないことに注意。というか、結果が double で書けないのだから解決のしようがない。

5 オーバーフロー対策

下向き丸めを計算しているときに計算結果が $+\infty$ になった場合、または上向き丸めを計算しているときに計算結果が $-\infty$ になった場合に注意が必要である。例えば下向き丸めの加算を考えると、$\infty + 1$ のように、初めから ∞ 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。これらをきちんと処理して、丸めモードを上向きまたは下向きにして CPU に計算させた場合と動作を一致させる必要がある。

6 対策後のプログラム

今までに述べてきた対策を全て施して出来たプログラムを以下に示す。

加減算の場合は、対策された twosum を使えば、後はオーバーフロー対策のみである。なお、プログラム中の `sys.float_info.max` は、正の浮動小数点数の最大の数 (C 言語での `DBL_MAX`、$2^{1024} - 2^{971}$) を表し、`float("inf")` は無限大を表す python の表記である。

```
import sys

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
```

```
import sys

def sub_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 sub_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
```

乗算では、まずオーバーフロー処理を行う。これは加算と同様。

次に twoproduct を行い、

$$|x| \geq 2^{-969}$$

を満たす場合はアンダーフローの心配が無いので、通常の処理を行う。

アンダーフローの心配がある場合は、a と b に $2^{537} = \sqrt{2^{1074}}$ を乗じてから再度 twoproduct を行う (s1,s2 とする)。これにより、a と b の考え得る最小の bit (2^{-1074}) の積 (2^{-2148}) が 2^{-1074} まで持ち上げられるので、アンダーフローは起きなくなる。また、$|x| < 2^{-969}$ よりオーバーフローの心配もない。s1 の値と $x \times 2^{1074}$ を比較し、同じなら更に s2 の正負を見れば、x が大きめだったか小さめだったか判定できる。

```
import sys

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
```

除算はやや複雑である。まず、a と b のどちらかが $0, \pm\infty$, NaN の場合を除外する (黙って $a \div b$ の計算結果を返す)。

次に b が正になるように正規化する。

以下、twoproduct のアンダーフロー対策を行う。$|a| < 2^{-969}$ のときアンダーフローの危険があるが、もし $|b| < 2^{918}$ なら、a と b それぞれに 2^{105} を乗ずることで回避できる。回避できない場合は、$|a \div b| < 2^{-1887}$ なので、a の符号と丸めの向きによって $0, \pm 2^{-1074}$ のうちから適切なものを返す。

これ以降は、オーバーフロー判定が加わる以外は同じ。

```
import sys

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
```

平方根も、除算と同様、アンダーフロー対策の場合分けを行う。$|a| < 2^{-969}$ の場合、a に 2^{106} を、d ($\sqrt{a}$ の計算結果) に 2^{53} を乗じて比較する。

```
import math

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
```

7 数値実験

本節の実験環境は、以下の通り。

- CPU: core i7 2640M (2.8GHz)
- OS: ubuntu 10.04 LTS (64bit)
- compiler: gcc 4.4
- compile option: -O3

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。

この実験に用いたプログラムを付録に示した。

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。

次に、区間演算を多用した精度保証付き計算の実アプリケーションでの速度を示す。区間演算では、丸めモード変更のオーバーヘッドを極力減らすために、一度丸めモードを変更したらなるべくそのモードのままで計算するようにコードを書くので、一般的にはこの例のように一演算毎に 2 回の丸め変更を行うことはない。よって、実アプリケーションで比較する必要がある。

ここでは、[4] で示された、2 つのトランジスタを含む回路で 5 つの解を持つ例について、4 つの未知数それぞれについて $[-10, 10]$ の範囲で全解探索を行った例を示す。解いた方程式は以下のようなもの。

$$\begin{pmatrix} 1 & -0.5 & 0 & 0 \\ -0.99 & 1 & 0 & 0 \\ 0 & 0 & 1 & -0.5 \\ 0 & 0 & -0.99 & 1 \end{pmatrix} \begin{pmatrix} 10^{-9}/0.99 \times (\exp(x_1/0.053) - 1) \\ 10^{-9}/0.5 \times (\exp(x_2/0.053) - 1) \\ 10^{-9}/0.99 \times (\exp(x_3/0.053) - 1) \\ 10^{-9}/0.5 \times (\exp(x_4/0.053) - 1) \end{pmatrix}$$
$$+ \quad 10^{-4} \begin{pmatrix} 4 & -3 & -2 & 1 \\ -3 & 3 & 1 & 0 \\ -2 & 1 & 4 & -3 \\ 1 & 0 & -3 & 3 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} + \begin{pmatrix} -0.001 \\ 0.000936 \\ -0.001 \\ 0.000936 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

自作の全解探索プログラムで、非存在テストを 263922 回、存在テストを 125957 回行った結果、5 つの解の存在証明に成功した。その実行時間は、

丸めモード変更	12.84s
エミュレート	19.18s

であった。やや遅くなっているが、丸めモードの変更の必要がなく幅広い環境で動くことを考えれば、十分実用的であると言えるのではないか。

8 おわりに

まだ次の問題点が残っていることが分かっている。

- $+0$ が -0 になる等の、0 の符号が異なることがある。

- NaN の種類が異なることがある。

前節のテストプログラムではこれらの差異は検出できないが、bit pattern を比較すれば検出可能である。区間演算を実現する目的にはこれらの挙動まで一致させる必要はないので放置したが、速度を損なわずに直せるものならば直したい。

9 重要な制限事項

本アルゴリズムは IEEE 754 Std. に完全に準拠している環境でのみ正しく動作する。従って、Intel の CPU で SSE2 でなく旧来の FPU を用いた場合、FPU は内部的に 80bit の (IEEE 754 Std. の double より長い) レジスタを使っているため IEEE 754 Std. に準拠しておらず、本アルゴリズムは正しく動作しない。内部で 80bit (仮数部 64bit) で計算した後 64bit (仮数部 53bit) に丸めるという動作をするため、ごく稀に計算結果が nearest にならないことがあるためである。

例の一つを示す。

$$\begin{aligned} a &= 8033714.4154930002987384796142578125 \\ b &= 736959594.84309303760528564453125 \end{aligned}$$

これら double で表現可能な 2 数 a, b の積を考える。$a \times b$ は正確には

$$5920522920726837.49981155413076772120462010207120329141616821289062 5$$

となり、これより大きな最小の、小さな最大の double はそれぞれ

$$5920522920726837$$
$$5920522920726838$$

であり、nearest モードなら明らかに末尾 7 になるべき。ところが、FPU を使ってこれを計算させると (例えば 32bit linux で計算すると)、末尾が 8 になってしまう。この理由は、$a \times b$ の 2 進数表現を考えると分かる。

$$1.0101000010001010111100101001100100110001000100110101|01111111111|1001$$
$$1101001100110011100100111100100101 01 \times 2^{52}$$

(53 桁目と 54 桁目、64 桁目と 65 桁めの間に | を入れた。) これを普通に double (仮数部 53 桁) に丸めると、

$$1.0101000010001010111100101001100100110001000100110101 \times 2^{52}$$

となるが、一回 64bit にすると、繰り上がって

$$1.0101000010001010111100101001100100110001000100110101|10000000000 \times 2^{52}$$

となり、再度これを double(53bit) にすると、偶数丸めルールも手伝って、

$$1.0101000010001010111100101001100100110001000100110110 \times 2^{52}$$

となってしまう。

32bit OS であっても SSE2 を持つ CPU ならばコンパイルオプションでそれを使わせるとか、あるいは FPU の有効数字を 53bit に制限するなどして本アルゴリズムをうまく動かすことは可能と思われるが、お勧めしない。

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

[4] Yusuke Nakaya, Tetsuo Nishi, Shin’ichi Oishi, and Martin Claus: “Numerical Existence Proof of Five Solutions for Certain Two-Transistor Circuit Equations”, Japan J. Indust. Appl. Math. Volume 26, Number 2-3, pp.327–336, 2009

10 付録

数値実験に使った C++プログラム。

```
#include <fenv.h>
#include <boost/random.hpp>

#ifndef NT
#define NT 1000000000
#endif

struct hwround {
    public:

    static void roundnear() {
        fesetround(FE_TONEAREST);
    }

    static void rounddown() {
        fesetround(FE_DOWNWARD);
    }

    static void roundup() {
        fesetround(FE_UPWARD);
    }

    static void roundchop() {
        fesetround(FE_TOWARDZERO);
    }

    static double add_up(const double& x,
         const double& y) {
        volatile double r, x1 = x, y1 = y
            ;
        roundup();
        r = x1 + y1;
        roundnear();
        return r;
    }

    static double add_down(const double&
        x, const double& y) {
        volatile double r, x1 = x, y1 = y
            ;
        rounddown();
        r = x1 + y1;
        roundnear();
        return r;
    }

    static double sub_up(const double& x,
         const double& y) {
        volatile double r, x1 = x, y1 = y
            ;
        roundup();
        r = x1 - y1;
        roundnear();
        return r;
    }

    static double sub_down(const double&
        x, const double& y) {
        volatile double r, x1 = x, y1 = y
            ;
        rounddown();
        r = x1 - y1;
        roundnear();
        return r;
    }

    static double mul_up(const double& x,
         const double& y) {
        volatile double r, x1 = x, y1 = y
            ;
        roundup();
        r = x1 * y1;
        roundnear();
        return r;
    }

    static double mul_down(const double&
        x, const double& y) {
        volatile double r, x1 = x, y1 = y
            ;
        rounddown();
        r = x1 * y1;
        roundnear();
        return r;
    }

    static double div_up(const double& x,
         const double& y) {
        volatile double r, x1 = x, y1 = y
            ;
        roundup();
        r = x1 / y1;
        roundnear();
        return r;
    }

    static double div_down(const double&
        x, const double& y) {
        volatile double r, x1 = x, y1 = y
            ;
        rounddown();
        r = x1 / y1;
        roundnear();
        return r;
    }

    static double sqrt_up(const double& x
        ) {
        volatile double r, x1 = x;
        roundup();
        r = sqrt(x1);
        roundnear();
        return r;
    }

    static double sqrt_down(const double&
         x) {
        volatile double r, x1 = x;
        rounddown();
        r = sqrt(x1);
        roundnear();
        return r;
    }
};

struct nohwround {

    static void twosum(const double& a,
        const double& b, double& x,
        double& y) {
        double tmp;

        x = a + b;
        if (std::fabs(a) > std::fabs(b))
            {
            tmp = x - a;
            y = b - tmp;
        } else {
            tmp = x - b;
            y = a - tmp;
        }
    }

    static void split(const double& a,
        double& x, double& y) {
        static const double sigma = ldexp
            (1., 27) + 1.;
        double tmp;

        tmp = a * sigma;
        x = tmp - (tmp - a);
        y = a - x;
    }
```

```
static void twoproduct(const double&
    a, const double& b, double& x,
    double& y) {
    static const double th = ldexp
        (1., 996);
    static const double c1 = ldexp
        (1., -28);
    static const double c2 = ldexp
        (1., 28);
    static const double th2 = ldexp
        (1., 1023);

    double na, nb, a1, a2, b1, b2;

    x = a * b;
    if (std::fabs(a) > th) {
        na = a * c1;
        nb = b * c2;
    } else if (std::fabs(b) > th) {
        na = a * c2;
        nb = b * c1;
    } else {
        na = a;
        nb = b;
    }
    split(na, a1, a2);
    split(nb, b1, b2);
    if (std::fabs(x) > th2) {
        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);
    }
}

static double succ(const double& x) {
    static const double th1 = ldexp
        (1., -969);
    static const double th2 = ldexp
        (1., -1021);
    static const double c1 = ldexp
        (1., -53) + ldexp(1., -105);
    static const double c2 = ldexp
        (1., -1074);
    static const double c3 = ldexp
        (1., 53);
    static const double c4 = ldexp
        (1., -53);

    double a, c, e;

    a = std::fabs(x);
    if (a >= th1) return x + a * c1;
    if (a < th2) return x + c2;
    c = c3 * x;
    e = c1 * std::fabs(c);
    return (c + e) * c4;
}

static double pred(const double& x) {
    static const double th1 = ldexp
        (1., -969);
    static const double th2 = ldexp
        (1., -1021);
    static const double c1 = ldexp
        (1., -53) + ldexp(1., -105);
    static const double c2 = ldexp
        (1., -1074);
    static const double c3 = ldexp
        (1., 53);
    static const double c4 = ldexp
        (1., -53);

    double a, c, e;

    a = std::fabs(x);
    if (a >= th1) return x - a * c1;
    if (a < th2) return x - c2;
    c = c3 * x;
    e = c1 * std::fabs(c);
    return (c - e) * c4;
}

static double add_up(const double& x,
     const double& y) {
    double r, r2;

    twosum(x, y, r, r2);
    if (r == std::numeric_limits<
        double>::infinity()) {
        return r;
    } else if (r == -std::
        numeric_limits<double>::
        infinity()) {
        if (x == -std::numeric_limits
            <double>::infinity() || y
             == -std::numeric_limits<
            double>::infinity()) {
            return r;
        } else {
            return -(std::
                numeric_limits<double
                >::max)();
        }
    }

    if (r2 > 0.) {
        return succ(r);
    }

    return r;
}

static double add_down(const double&
    x, const double& y) {
    double r, r2;

    twosum(x, y, r, r2);
    if (r == std::numeric_limits<
        double>::infinity()) {
        if (x == std::numeric_limits<
            double>::infinity() || y
            == std::numeric_limits<
            double>::infinity()) {
            return r;
        } else {
            return (std::
                numeric_limits<double
                >::max)();
        }
    } else if (r == -std::
        numeric_limits<double>::
        infinity()) {
        return r;
    }

    if (r2 < 0.) {
        return pred(r);
    }

    return r;
}

static double sub_up(const double& x,
     const double& y) {
    double r, r2;

    twosum(x, -y, r, r2);
    if (r == std::numeric_limits<
        double>::infinity()) {
        return r;
    } else if (r == -std::
        numeric_limits<double>::
        infinity()) {
        if (x == -std::numeric_limits
            <double>::infinity() || y
             == std::numeric_limits<
            double>::infinity()) {
```

```
            return r;
        } else {
            return -(std::
                numeric_limits<double
                >::max)();
        }
    }

    if (r2 > 0.) {
        return succ(r);
    }

    return r;
}

static double sub_down(const double&
    x, const double& y) {
    double r, r2;

    twosum(x, -y, r, r2);
    if (r == std::numeric_limits<
        double>::infinity()) {
        if (x == std::numeric_limits<
            double>::infinity() || y
            == -std::numeric_limits<
            double>::infinity()) {
            return r;
        } else {
            return (std::
                numeric_limits<double
                >::max)();
        }
    } else if (r == -std::
        numeric_limits<double>::
        infinity()) {
        return r;
    }

    if (r2 < 0.) {
        return pred(r);
    }

    return r;
}

static double mul_up(const double& x,
     const double& y) {
    double r, r2;
    double x1, y1;
    double s, s2, t;
    static const double th = ldexp
        (1., -969); // -1074 + 106 -
        1
    static const double c = ldexp(1.,
         537); // 1074 / 2

    twoproduct(x, y, r, r2);
    if (r == std::numeric_limits<
        double>::infinity()) {
        return r;
    } else if (r == -std::
        numeric_limits<double>::
        infinity()) {
        if (std::fabs(x) == std::
            numeric_limits<double>::
            infinity() || std::fabs(y
            ) == std::numeric_limits<
            double>::infinity()) {
            return r;
        } else {
            return -(std::
                numeric_limits<double
                >::max)();
        }
    }

    if (fabs(r) >= th) {
        if (r2 > 0.) return succ(r);
        return r;
    } else {
        twoproduct(x * c, y * c, s,
            s2);
        t = (r * c) * c;
        if ( t < s || (t == s && s2 >
             0.)) {
            return succ(r);
        }
        return r;
    }
}

static double mul_down(const double&
    x, const double& y) {
    double r, r2;
    double x1, y1;
    double s, s2, t;
    static const double th = ldexp
        (1., -969); // -1074 + 106 -
        1
    static const double c = ldexp(1.,
         537); // 1074 / 2

    twoproduct(x, y, r, r2);
    if (r == std::numeric_limits<
        double>::infinity()) {
        if (std::fabs(x) == std::
            numeric_limits<double>::
            infinity() || std::fabs(y
            ) == std::numeric_limits<
            double>::infinity()) {
            return r;
        } else {
            return (std::
                numeric_limits<double
                >::max)();
        }
    } else if (r == -std::
        numeric_limits<double>::
        infinity()) {
        return r;
    }

    if (fabs(r) >= th) {
        if (r2 < 0.) return pred(r);
        return r;
    } else {
        twoproduct(x * c, y * c, s,
            s2);
        t = (r * c) * c;
        if ( t > s || (t == s && s2 <
             0.)) {
            return pred(r);
        }
        return r;
    }
}

static double div_up(const double& x,
     const double& y) {
    double r, r2;
    double xn, yn, d;
    static const double th1 = ldexp
        (1., -969); // -1074 + 106 -
        1
    static const double th2 = ldexp
        (1., 918); // 1023 - 105
    static const double c1 = ldexp
        (1., 105); // -969 - (-1074)
    static const double c2 = ldexp
        (1., -1074);

    if (x == 0. || y == 0. || std::
        fabs(x) == std::
        numeric_limits<double>::
        infinity() || std::fabs(y) ==
         std::numeric_limits<double
        >::infinity() || x != x  || y
         != y) {
        return x / y;
    }
```

```
    if (y < 0.) {
        xn = -x;
        yn = -y;
    } else {
        xn = x;
        yn = y;
    }

    if (fabs(xn) < th1) {
        if (fabs(yn) < th2) {
            xn *= c1;
            yn *= c1;
        } else {
            if (xn < 0.) return 0.;
            else return c2;
        }
    }

    d = xn / yn;

    if (d == std::numeric_limits<
        double>::infinity()) {
        return d;
    } else if (d == -std::
        numeric_limits<double>::
        infinity()) {
        return -(std::numeric_limits<
            double>::max)();
    }

    twoproduct(d, yn, r, r2);
    if ( r < xn || ((r == xn) && r2 <
         0.)) {
        return succ(d);
    }
    return d;
}

static double div_down(const double&
    x, const double& y) {
    double r, r2;
    double xn, yn, d;
    static const double th1 = ldexp
        (1., -969); // -1074 + 106 -
        1
    static const double th2 = ldexp
        (1., 918); // 1023 - 105
    static const double c1 = ldexp
        (1., 105); // -969 - (-1074)
    static const double c2 = ldexp
        (1., -1074);

    if (x == 0. || y == 0. || std::
        fabs(x) == std::
        numeric_limits<double>::
        infinity() || std::fabs(y) ==
         std::numeric_limits<double
        >::infinity() || x != x  || y
         != y) {
        return x / y;
    }

    if (y < 0.) {
        xn = -x;
        yn = -y;
    } else {
        xn = x;
        yn = y;
    }

    if (fabs(xn) < th1) {
        if (fabs(yn) < th2) {
            xn *= c1;
            yn *= c1;
        } else {
            if (xn < 0.) return -c2;
            else return 0.;
        }
    }

    d = xn / yn;

    if (d == std::numeric_limits<
        double>::infinity()) {
        return (std::numeric_limits<
            double>::max)();
    } else if (d == -std::
        numeric_limits<double>::
        infinity()) {
        return d;
    }

    twoproduct(d, yn, r, r2);
    if ( r > xn || ((r == xn) && r2 >
         0.)) {
        return pred(d);
    }
    return d;
}

static double sqrt_up(const double& x
    ) {
    double r, r2, d;
    static const double th1 = ldexp
        (1., -969); // -1074 + 106 -
        1
    static const double c1 = ldexp
        (1., 106); // -969 - (-1074)
        + 1
    static const double c2 = ldexp
        (1., 53); // sqrt(c1)

    d = sqrt(x);

    if (x < th1) {
        double d2, x2;
        x2 = x * c1;
        d2 = d * c2;
        twoproduct(d2, d2, r, r2);
        if ( r < x2 || (r == x2) &&
            r2 < 0.) {
            return succ(d);
        }
        return d;
    }

    twoproduct(d, d, r, r2);
    if ( r < x || (r == x) && r2 <
        0.) {
        return succ(d);
    }
    return d;
}

static double sqrt_down(const double&
     x) {
    double r, r2, d;
    static const double th1 = ldexp
        (1., -969); // -1074 + 106 -
        1
    static const double c1 = ldexp
        (1., 106); // -969 - (-1074)
        + 1
    static const double c2 = ldexp
        (1., 53); // sqrt(c1)

    d = sqrt(x);

    if (x < th1) {
        double d2, x2;
        x2 = x * c1;
        d2 = d * c2;
        twoproduct(d2, d2, r, r2);
        if ( r > x2 || (r == x2) &&
            r2 > 0.) {
            return pred(d);
        }
        return d;
    }
```

```
        twoproduct(d, d, r, r2);
        if ( r > x || (r == x) && r2 >
            0.) {
            return pred(d);
        }
        return d;
    }

};

bool samedouble(double x, double y)
{
    if (x != x && y != y) return true;
    return x == y;
}

void check(double x, double y)
{
    volatile double r1, r2;

    r1 = hwround::add_up(x, y);
    r2 = nohwround::add_up(x, y);
    if (!samedouble(r1, r2)) {
        std::cout << "add_up error\n";
            std::cout << x << "\n";
        std::cout << y << "\n";
        std::cout << r1 << "\n";
        std::cout << r2 << "\n";
    }

    r1 = hwround::add_down(x, y);
    r2 = nohwround::add_down(x, y);
    if (!samedouble(r1, r2)) {
        std::cout << "add_down error\n";
        std::cout << x << "\n";
        std::cout << y << "\n";
        std::cout << r1 << "\n";
        std::cout << r2 << "\n";
    }

    r1 = hwround::sub_up(x, y);
    r2 = nohwround::sub_up(x, y);
    if (!samedouble(r1, r2)) {
        std::cout << "sub_up error\n";
        std::cout << x << "\n";
        std::cout << y << "\n";
        std::cout << r1 << "\n";
        std::cout << r2 << "\n";
    }

    r1 = hwround::sub_down(x, y);
    r2 = nohwround::sub_down(x, y);
    if (!samedouble(r1, r2)) {
        std::cout << "sub_down error\n";
        std::cout << x << "\n";
        std::cout << y << "\n";
        std::cout << r1 << "\n";
        std::cout << r2 << "\n";
    }

    r1 = hwround::mul_up(x, y);
    r2 = nohwround::mul_up(x, y);
    if (!samedouble(r1, r2)) {
        std::cout << "mul_up error\n";
        std::cout << x << "\n";
        std::cout << y << "\n";
        std::cout << r1 << "\n";
        std::cout << r2 << "\n";
    }

    r1 = hwround::mul_down(x, y);
    r2 = nohwround::mul_down(x, y);
    if (!samedouble(r1, r2)) {
        std::cout << "mul_down error\n";
        std::cout << x << "\n";
        std::cout << y << "\n";
        std::cout << r1 << "\n";
        std::cout << r2 << "\n";
    }

    r1 = hwround::div_up(x, y);
    r2 = nohwround::div_up(x, y);
    if (!samedouble(r1, r2)) {
        std::cout << "div_up error\n";
        std::cout << x << "\n";
        std::cout << y << "\n";
        std::cout << r1 << "\n";
        std::cout << r2 << "\n";
    }

    r1 = hwround::div_down(x, y);
    r2 = nohwround::div_down(x, y);
    if (!samedouble(r1, r2)) {
        std::cout << "div_down error\n";
        std::cout << x << "\n";
        std::cout << y << "\n";
        std::cout << r1 << "\n";
        std::cout << r2 << "\n";
    }

    r1 = hwround::sqrt_up(x);
    r2 = nohwround::sqrt_up(x);
    if (!samedouble(r1, r2)) {
        std::cout << "sqrt_up error\n";
        std::cout << x << "\n";
        std::cout << r1 << "\n";
        std::cout << r2 << "\n";
    }

    r1 = hwround::sqrt_down(x);
    r2 = nohwround::sqrt_down(x);
    if (!samedouble(r1, r2)) {
        std::cout << "sqrt_down error\n";
        std::cout << x << "\n";
        std::cout << r1 << "\n";
        std::cout << r2 << "\n";
    }
}

int main() {
    double x, y;
    int i, j;
    unsigned long long t;

    double specials[11] = {
        0.,
        -0.,
        std::numeric_limits<double>::
            infinity(),
        -std::numeric_limits<double>::
            infinity(),
        (std::numeric_limits<double>::max
            )(),
        -(std::numeric_limits<double>::
            max)(),
        (std::numeric_limits<double>::min
            )(),
        -(std::numeric_limits<double>::
            min)(),
        std::numeric_limits<double>::
            denorm_min(),
        -std::numeric_limits<double>::
            denorm_min()
    };
    specials[10] = specials[2] + specials
        [3]; // making NaN

    boost::variate_generator<boost::
        mt19937, boost::uniform_int<
        unsigned long long> > rand(boost
        ::mt19937(time(0)), boost::
        uniform_int<unsigned long long
        >(0, -1));

    std::cout.precision(17);

    for (i=0; i<NT; i++) {
        t = rand();
        x = *((double*)(&t));
```

```
        t = rand();
        y = *((double*)(&t));
        check(x, y);
    }

    // check general-special case

    for (i=0; i<NT; i++) {
        t = rand();
        x = *((double*)(&t));
        for (j=0; j<11; j++) {
            y = specials[j];
            check(x, y);
            check(y, x);
        }
    }

    // check special-special case

    for (i=0; i<11; i++) {
        x = specials[i];
        for (j=0; j<11; j++) {
            y = specials[j];
            check(x, y);
        }
    }
}
```