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
kashi@waseda.jp
http://verifiedby.me/

早稲田大学

第 43 回数値解析シンポジウム (2014 年 6 月 13 日)

発表の概要

- 目的 (方向付き丸めを最近点丸めモードのみでエミュレート)
- 基本原理
- twosum, twoproduct の限界による問題点
- 問題点の解決
- 数値実験

区間演算

区間演算

- 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
- 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。
- 例えば $[\underline{Z}, \overline{Z}] := [\underline{X}, \overline{X}] + [\underline{Y}, \overline{Y}]$ を計算するには、
  - 丸めモードを $-\infty$ 方向に変更
  - $\underline{Z} := \underline{X} + \underline{Y}$
  - 丸めモードを $+\infty$ 方向に変更
  - $\overline{Z} := \overline{X} + \overline{Y}$
  - (丸めモードを 最近点 に戻す)

丸めモード

丸めモードの変更

- 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。

丸めモードを変更しづらい場合も

- 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いない。
- 丸めモードの変更を行わずに区間演算を行えると嬉しい。

本発表の目的

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
- 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
- 全ての浮動小数点数に対応した (つもり)

基本原理

浮動小数点演算における丸めは自動的に行われ、計算してしまった後にどちら側に丸まったかを知るのは難しい。

$$\Downarrow$$

twosum, twoproduct を用いてどちらに丸まったかを知り、succ, pred を用いて必要ならば補正する。

twosum アルゴリズム

```
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
```

Donald E. Knuth: "The Art of Computer Programming Volume 2: Seminumerical Algorithms", Addison-Wesley, 1969

- 計算の前後で $x + y = a + b$ が数学的に厳密に成立する。
- 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「$x + y$ を浮動小数点演算で計算すると x になる」程度に小さい。
- 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 アルゴリズム

```
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
```

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

- 計算の前後で $x + y = a \times b$ が数学的に厳密に成立する。
- 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「$x + y$ 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 に分けている

succ, pred アルゴリズム

- succ, pred は、IEEE 754 Std. の浮動小数点数 x に対してその隣 (x より大きな最小の数または x より小さな最大の数) を計算する。
- Rump による実現方法の例:
  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

```
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):
    (omitted)
```

エミュレートの基本原理 (加算)

```
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
- 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。

エミュレートの基本原理 (減算)

```
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
```

- 加算とほぼ同様

エミュレートの基本原理(乗算)

```
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
```

- 加算とほぼ同様
- twoproduct により誤差を把握する

エミュレートの基本原理 (除算)

```
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) :
    (omitted)
```

- 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。
- $a \div b$ 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。

エミュレートの基本原理(平方根)

```
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
```

- 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。

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。

twosum の限界 (1)

```
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。
- オーバーフローからは逃れられない。

twosum の限界 (2)

- 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。

- 計算結果はオーバーフローしないが中間変数がオーバーフローする例:

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

修正版 twosum

```
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
```

- fasttwosum を絶対値の大きさで切り替えて使用する
- 中間変数のオーバーフローを防げる。

```
>>> twosum(3.5630624444874539e+307, -1.7976931348623157e+308)
(-1.4413868904135704e+308, 9.979201547673599 1e+291)
```

twoproduct の限界 (1)

```
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
- $x, y = \mathrm{twoproduct}(a, b)$ 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。

twoproduct の限界 (2)

- $|x| > 2^{-969} - 2^{-1021}$ であればアンダーフローは起きていないが、そうでなければアンダーフローが起きた可能性がある。
- split でオーバーフローが起きる可能性がある。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。
- 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, 2.594447525195200 3e+71)
(1.797693134862315 7e+308, inf)
```

修正版 twoproduct(1)

```
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
```

修正版 twoproduct(2)

- split のオーバーフローと $a1 \times b1$ のオーバーフローに対処した。
- 先の例は、

```
>>> twoproduct(6.929001713869936e+236, 2.594447525195200 3e+71)
(1.797693134862315 7e+308, -1.0027614963959625e+291)
```

  のようにきちんと計算できる。
- アンダーフロー問題は解決していない (解決のしようがない)。

twosum, twoproduct の修正まとめ

	twosum	twoproduct
アンダーフロー	元々問題なし	未解決
中間変数のオーバーフロー	解決	解決
オーバーフロー	未解決	未解決

オーバーフロー対策

- オーバーフローについては、
  - 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
  - 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
- 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 $\Rightarrow$ そのままでよい。
  - $10^{308} + 10^{308}$ のようにオーバーフローした $\Rightarrow$ 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。

修正した加算のエミュレート (オーバーフロー処理)

```
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
```

修正した減算のエミュレート (オーバーフロー処理)

```
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
```

修正した乗算のエミュレート (1)

```
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
```

(原稿に typo あり。オーバーフロー処理 + アンダーフロー処理)

修正した乗算のエミュレート (2)

```
import sys

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
```

(原稿に typo あり。オーバーフロー処理 + アンダーフロー処理)

修正した乗算のエミュレート (3)

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。

修正した除算のエミュレート (1)

```
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
```

修正した除算のエミュレート (2)

```
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
```

修正した除算のエミュレート (3)

- 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。

修正した平方根のエミュレート (1)

```
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
```

修正した平方根のエミュレート (2)

```
import math

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
```

修正した平方根のエミュレート (3)

- オーバーフローは考えなくてよい。
- 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。

数値実験環境

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

正しさの検証

- 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で検証。
- 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
- 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。
- 実験に使ったプログラムを、http://verifiedby.me/ の「精度保証日記」にあげておく。

速度評価

- 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

- このプログラムも http://verifiedby.me/ にあげておく。

実アプリケーションでの速度評価(1)

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。

5つの解を持つ2-トランジスタ回路方程式

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

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

実アプリケーションでの速度評価 (2)

- 4 つの未知数それぞれについて $[-10, 10]$ の範囲で全解探索
- 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。
- 実行時間は、

丸めモード変更	12.84s
エミュレート	19.18s

- このプログラムも http://verifiedby.me/ にあげておく。
- 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。

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。
- Intel CPU の 32bit 環境 (FPU を使った環境) は 80bit 過剰精度を持つため IEEE 754 Std. に準拠しておらず、本手法は正常に動作しない可能性がある。

宣伝

本プログラムを含め、最近の研究で作成したプログラムのほぼ全てを `http://verifiedby.me/` で公開している。是非使ってみて欲しい。

おまけ: FMA を使ってみる (1)

- 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
```

- 中間変数のオーバーフローや split のオーバーフローの問題は無いので、修正の必要無し。
- Haswell が持つ AVX2 は FMA 命令を持っている。
- 最近 Haswell なマシンに乗り換えたので FMA で twoproduct を作って遊んでみた。

おまけ: FMA を使ってみる (2)

FMA を使う最小の?コード

```
#include <immintrin.h>

// need -mfma on gcc

double fma(double a, double b, double c)
{
    double d;
    __m128d va = _mm_set_sd(a);
    __m128d vb = _mm_set_sd(b);
    __m128d vc = _mm_set_sd(c);
    __m128d vd = _mm_fmadd_sd(va, vb, vc);
    _mm_store_sd(&d, vd);
    return d;
}
```

Haswell は 4 並列で FMA を計算できるが、このコードでは 1 つしか計算していない。

おまけ: FMA を使ってみる (3)

個々の演算のベンチマーク

	加算	減算	乗算	除算	平方根
丸めモード変更	14.0ns	14.0ns	15.4ns	14.9ns	13.8ns
エミュレート	1.0ns	1.0ns	13.6ns	14.0ns	5.7ns
エミュレート+ FMA	1.0ns	1.0ns	1.6ns	7.1ns	2.0ns

全解探索のベンチマーク

丸めモード変更	10.03s
エミュレート	13.39s
エミュレート+ FMA	9.51s

core i7 4600U (2.1GHz), ubuntu 14.04 LTS (64bit), gcc 4.8.2, -O3