

(Version: 2017/5/12)

## double-double 演算、double-double 区間演算に関するまとめ

柏木 雅英

### 1 はじめに

double-double 演算 (dd 演算と省略されることもある) は、double 型のデータを 2 つ用いてそれぞれに上位 bit と下位 bit を格納し、擬似的に 4 倍精度計算を行なう技術である。古くから知られており多くの実装があるが、<http://crd-legacy.lbl.gov/~dhbailey/mpdist/> で公開されている Bailey の一連のソフトウェアがよく使われている。

本文書は、double-double 演算について知識を自分なりに整理したものである。

また、ちょっとした工夫で double-double 演算に丸めの向きを指定する機能を付加することができ、これを使うと double-double 型を両端に持つ区間演算を実現することができる。それについてもまとめている。

### 2 twosum, twoproduct

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

twosum[1] は、python 風にかくと以下の通り。

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

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

$a$  と  $b$  から  $x, y = \text{twosum}(a, b)$  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_org(a, b) :
    x = a * b
    a1, a2 = split(a)
    b1, b2 = split(b)
    y = (((a1 * b1 - x) + a2 * b1) + a1 * b2) + a2 * b2
    return x, y

```

$a$  と  $b$  から  $x, y = \text{twoproduct}(a, b)$  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 twoproduct_fma(a, b) :
    x = a * b
    y = fma(a, b, -x)
    return x, y

```

なお、`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。

### 3 改良版 `twosum`, `twoproduct`

前述したように、`twosum`、`twoproduct` はアンダーフローやオーバーフローを考えると無誤差というわけではない。ここでは、特にオーバーフロー近辺での挙動を改善した `twosum`、`twoproduct` を示す。

#### 3.1 `twosum` の改良

`twosum` を以下に再掲する。

```

def twosum_org(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りて適当な 2 のべきでスケールリングする方法が考えられるが、ここではいわゆる `fasttwosum` を絶対値の大きさに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。

## 3.2 twoproduct の改良

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

`twoproduct` は、 $a_1 \times b_1$ ,  $a_2 \times b_1$ ,  $a_1 \times b_2$ ,  $a_2 \times b_2$  の 4 つの積の部分でアンダーフローを起こす可能性があり、その場合は無誤差ではない。

オーバーフローに関して述べる。`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。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  $a_1 \times b_1$  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  $x - a_1 \times b_1$  を  $x/2 - (a_1/2) \times b_1$  と計算することで対策した。

split と  $a_1 \times b_1$  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 = (((a1 * 0.5) * b1 - (x * 0.5)) * 2. + a2 * b1) + a1 * b2) + a2 * b2
    else:
        y = (((a1 * b1 - x) + a2 * b1) + a1 * b2) + a2 * 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。

この節で示した方法は無限大付近での動作の改良であり、「そんなに大きい数ちゃんと扱えなくてもいいよ。それより少しでも速い方がいいよ」という場合は、必ずしも改良版を使う必要はない。

## 4 dd 演算

twosum と twoproduct は、加算や乗算の結果を上位 bit と下位 bit に分けて出力してくれる。これを用いて、データ構造として double を 2 つ用いてそれぞれに上位 bit と下位 bit を格納し、擬似的に 4 倍精度計算を行なうことが出来る。double-double 演算と呼ばれることもある。

## 4.1 加算

加算は以下のように行なう。

```
def dd_add(x1, x2, y1, y2) :
    z1, z2 = twosum(x1, y1)
    if abs(z1) == float("inf") :
        return z1, 0
    z2 = z2 + x2 + y2
    z1, z2 = twosum(z1, z2)
    return z1, z2
```

上位桁の  $x_1, y_1$  を `twosum` で加えたものを  $z_1, z_2$  とし、結果の上位はそのまま  $z_1$  に、結果の下位は  $z_2 + x_2 + y_2$  とする。 $z_1$  と  $z_2$  は仮数部の重なりが無い程度に離れているはずであるが、 $z_2$  に  $x_2$  と  $y_2$  を加えたことによりわずかに  $z_1$  と重なる可能性があるため、最後に `twosum` で重なりが無いように整えている。

また、 $z_1$  が無限大になってしまうような場合、次の `twosum` で NaN が発生してしまうので、特殊処理をしている。

## 4.2 減算

減算は、 $y_1, y_2$  の符号を反転させる以外は加算と同じ。

```
def dd_sub(x1, x2, y1, y2) :
    z1, z2 = twosum(x1, -y1)
    if abs(z1) == float("inf") :
        return z1, 0
    z2 = z2 + x2 - y2
    z1, z2 = twosum(z1, z2)
    return z1, z2
```

## 4.3 乗算

乗算は、次のように行なう。

```
def dd_mul(x1, x2, y1, y2) :
    z1, z2 = twoproduct(x1, y1)
    if abs(z1) == float("inf") :
        return z1, 0
    z2 = z2 + x1 * y2 + x2 * y1 + x2 * y2
    z1, z2 = twosum(z1, z2)
    return z1, z2
```

$x_1 y_1$  のみ `twoproduct` を用いてきちんと計算し、それ以外の  $x_1 y_2, x_2 y_1, x_2 y_2$  の項は普通に計算する。

なお、 $x_2 y_2$  は  $|x_1 y_1| \times 2^{-106}$  程度の大きさであり、通常は結果に影響を及ぼさないので省略してしまう場合が多いが、精度を気にするのならちゃんと計算することをお勧めする<sup>1</sup>。

<sup>1</sup> 例えば  $(1 + 2^{-54}) \times (1 - 2^{-54})$  みたいな計算だと、省略しなければきちんと  $1 - 2^{-108}$  になるが、省略すると 1 になってしまう。

## 4.4 除算

$z_1 \simeq x_1/y_1$  を近似計算でまず求め、次にそれを利用して

$$\begin{aligned}\frac{x_1 + x_2}{y_1 + y_2} &= \frac{x_1 + x_2 - (y_1 + y_2)z_1 + (y_1 + y_2)z_1}{y_1 + y_2} \\ &= z_1 + \frac{x_1 + x_2 - (y_1 + y_2)z_1}{y_1 + y_2} \\ &= z_1 + \frac{x_1 - y_1z_1 + x_2 - y_2z_1}{y_1 + y_2}\end{aligned}$$

のように変形する。更に、

$$z_3, z_4 = \text{twoproduct}(y_1, -z_1)$$

のように変換し、

$$\frac{x_1 + x_2}{y_1 + y_2} = z_1 + \frac{(x_1 + z_3) + z_4 + x_2 - y_2z_1}{y_1 + y_2}$$

とする。 $x_1$  と  $z_3 \simeq -y_1z_1$  はキャンセルする筈なので、 $(x_1 + z_3)$  の部分は先に計算する。残りの部分  $(z_4, x_2, -y_2z_1)$  の大きさはいずれも  $2^{-53}|x_1|$  程度の大きさで予想されるので、加えるのはどの順序でも良さそう。また、分母の  $y_1 + y_2$  の部分の  $y_2$  の加算は影響を及ぼさないはずなので、結局

$$z_2 = \frac{(x_1 + z_3) + z_4 + x_2 - y_2z_1}{y_1}$$

と計算する。

```
def dd_div(x1, x2, y1, y2) :
    z1 = x1 / y1

    if abs(z1) == float("inf") :
        return z1, 0
    if abs(y1) == float("inf") :
        return z1, 0

    z3, z4 = twoproduct(-z1, y1)

    if abs(z3) == float("inf") :
        z3, z4 = twoproduct(-z1, y1 * 0.5)
        z2 = (((z3 + (x1 * 0.5)) - z1 * (y2 * 0.5)) + (x2 * 0.5)) + z4) / (y1 * 0.5)
    else :
        z2 = (((z3 + x1) - z1 * y2) + x2) + z4) / y1

    z1, z2 = twosum(z1, z2)
    return z1, z2
```

なお、入力値  $x_1$  が無限大に近い場合、それを  $y_1$  で割って再度  $y_1$  を乗じたときにその結果 ( $z_3$ ) がオーバーフローすることがある。そのときは

$$z_3, z_4 = \text{twoproduct}(y_1/2, -z_1)$$

$$z_2 = \frac{(x_1/2 + z_3) + z_4 + x_2/2 - y_2 z_1/2}{y_1/2}$$

のように分母分子を半分にすることで対応する。

$x \div y = x \times (1/y)$  と逆数と乗算に分解し、逆数を Newton 法で計算するやり方も考えられる。

## 4.5 平方根

$z_1 \simeq \sqrt{x_1}$  でまず近似平方根を求める。次に、

$$z_1 + z_2 = \sqrt{x_1 + x_2}$$

となるような  $z_2$  を計算することを考える。

$$\begin{aligned} z_2 &= \sqrt{x_1 + x_2} - z_1 \\ &= \frac{x_1 + x_2 - z_1^2}{\sqrt{x_1 + x_2} + z_1} \end{aligned}$$

と変形する。ここで、

$$z_3, z_4 = \text{twoproduct}(z_1, -z_1)$$

と変換し、

$$z_2 = \frac{(x_1 + z_3) + x_2 + z_4}{\sqrt{x_1 + x_2} + z_1}$$

とする。 $x_1$  と  $z_3 \simeq -z_1^2$  はキャンセルする筈なので、 $(x_1 + z_3)$  の部分は先に計算する。また、分母の  $x_1 + x_2$  の部分の  $x_2$  の加算は影響を及ぼさないはずであり、 $\sqrt{x_1}$  はすでに  $z_1$  に計算されているので、結局

$$z_2 = \frac{(x_1 + z_3) + x_2 + z_4}{2z_1}$$

と計算する。

```
def dd_sqrt(x1, x2) :
    if x1 == 0 and x2 == 0:
        return 0, 0
    if x1 == float("inf"):
        return x1, 0

    z1 = math.sqrt(x1)
    z3, z4 = twoproduct(-z1, z1)
    z2 = ((z3 + x1) + x2 + z4) / (2 * z1)
    z1, z2 = twosum(z1, z2)
    return z1, z2
```

Newton 法と既に定義した dd の除算、加算を使って、

```

def dd_sqrt_old(x1, x2) :
    if x1 == 0 and x2 == 0:
        return 0, 0
    if x1 == float("inf"):
        return x1, 0

    z1 = math.sqrt(x1)
    z2 = 0
    z3, z4 = dd_div(x1, x2, z1, z2)
    z1, z2 = dd_add(z1, z2, z3, z4)
    z1 = z1 * 0.5
    z2 = z2 * 0.5

    return z1, z2

```

のように計算する方法もある。

## 5 dd 区間演算

double 型を両端に持つ区間演算は、加減乗除と平方根に対する上向き丸め、下向き丸めの演算を用いて実現されている。それと同様に、「誤差が入るとすれば必ず上向き丸めになる」ような dd の加減乗除、平方根と、「誤差が入るとすれば必ず下向き丸めになる」ような dd の加減乗除、平方根を準備すれば、両端に dd 数を持つような区間演算が実現できる。

本章のサンプルではこれ以降、

```
near(), up(), down()
```

などの丸めモードを変更する命令が出てくるが、これらは python の組み込み命令ではない。これらを実現する方法は、文末の付録を見て欲しい。

まず、オーバーフロー対策を施した twoproduct を再掲する。

```

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
    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 = (((a1 * 0.5) * b1 - (x * 0.5)) * 2. + a2 * b1) + a1 * b2) + a2 * b2
    else:
        y = (((a1 * b1 - x) + a2 * b1) + a1 * b2) + a2 * b2
    return x, y

```



赤字の部分でアンダーフローが起きる可能性があるため、twoproductは無誤差ではない。そこで、その部分を方向付き丸めで計算する twoproduct\_up, twoproduct\_down を作成する。

```
def twoproduct_up(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)
    up()
    if abs(x) > 2.**1023:
        y = (((a1 * 0.5) * b1 - (x * 0.5)) * 2. + a2 * b1) + a1 * b2) + a2 * b2
    else:
        y = (((a1 * b1 - x) + a2 * b1) + a1 * b2) + a2 * b2
    near()
    return x, y
```

```
def twoproduct_down(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)
    down()
    if abs(x) > 2.**1023:
        y = (((a1 * 0.5) * b1 - (x * 0.5)) * 2. + a2 * b1) + a1 * b2) + a2 * b2
    else:
        y = (((a1 * b1 - x) + a2 * b1) + a1 * b2) + a2 * b2
    near()
    return x, y
```

これを使うと、次のように twoproduct に誤差が入る場合にその誤差の向きを制御できる。

```
>>> twoproduct(1e-150,1e-150)
(1e-300, -1.246838e-317)
>>> twoproduct_up(1e-150,1e-150)
(1e-300, -1.246837e-317)
>>> twoproduct_down(1e-150,1e-150)
(1e-300, -1.2468384e-317)
```

## 5.1 加算

double-double の加算アルゴリズム

```
def dd_add(x1, x2, y1, y2) :
    z1, z2 = twosum(x1, y1)
    if abs(z1) == float("inf") :
        return z1, 0
    z2 = z2 + x2 + y2
    z1, z2 = twosum(z1, z2)
    return z1, z2
```

では、赤字の部分で誤差が入る可能性がある。そこで、

```
def dd_add_up(x1, x2, y1, y2) :
    z1, z2 = twosum(x1, y1)
    if abs(z1) == float("inf") :
        return z1, 0
    up()
    z2 = z2 + x2 + y2
    near()
    z1, z2 = twosum(z1, z2)
    return z1, z2
```

```
def dd_add_down(x1, x2, y1, y2) :
    z1, z2 = twosum(x1, y1)
    if abs(z1) == float("inf") :
        return z1, 0
    down()
    z2 = z2 + x2 + y2
    near()
    z1, z2 = twosum(z1, z2)
    return z1, z2
```

のようにすれば、上向き丸め、下向き丸めの加算が実現できる。

## 5.2 減算

加算とほぼ同様。

```
def dd_sub_up(x1, x2, y1, y2) :
    z1, z2 = twosum(x1, -y1)
    if abs(z1) == float("inf") :
        return z1, 0
    up()
    z2 = z2 + x2 - y2
    near()
    z1, z2 = twosum(z1, z2)
    return z1, z2
```

```
def dd_sub_down(x1, x2, y1, y2) :
    z1, z2 = twosum(x1, -y1)
    if abs(z1) == float("inf") :
        return z1, 0
    down()
    z2 = z2 + x2 - y2
    near()
    z1, z2 = twosum(z1, z2)
    return z1, z2
```

### 5.3 乗算

乗算で誤差が発生する可能性があるのは、以下の赤字の部分。

```
def dd_mul(x1, x2, y1, y2) :
    z1, z2 = twoproduct(x1, y1)
    if abs(z1) == float("inf") :
        return z1, 0
    z2 = z2 + x1 * y2 + x2 * y1 + x2 * y2
    z1, z2 = twosum(z1, z2)
    return z1, z2
```

そこで、赤字の部分の丸めを制御する。もちろん、精度保証のためには小さいからといって最後の  $x_2 \times y_2$  を省略することは出来ない。

```
def dd_mul_up(x1, x2, y1, y2) :
    z1, z2 = twoproduct_up(x1, y1)
    if abs(z1) == float("inf") :
        return z1, 0
    up()
    z2 = z2 + x1 * y2 + x2 * y1 + x2 * y2
    near()
    z1, z2 = twosum(z1, z2)
    return z1, z2
```

```
def dd_mul_down(x1, x2, y1, y2) :
    z1, z2 = twoproduct_down(x1, y1)
    if abs(z1) == float("inf") :
        return z1, 0
    down()
    z2 = z2 + x1 * y2 + x2 * y1 + x2 * y2
    near()
    z1, z2 = twosum(z1, z2)
    return z1, z2
```

### 5.4 除算

除算は少し複雑である。先に示した除算と同様に、 $z_1 \simeq x_1/y_1$  を近似計算でまず求め、次にそれを利用して

$$\frac{x_1 + x_2}{y_1 + y_2} = z_1 + \frac{x_1 + \text{twoproduct}(y_1, -z_1) + x_2 - y_2 z_1}{y_1 + y_2}$$

のようにする。ここで、

$$z_2 = \frac{x_1 + \text{twoproduct}(y_1, -z_1) + x_2 - y_2 z_1}{y_1 + y_2}$$

の部分の計算は、丸めの向きを慎重に考える必要がある。区間演算して上端または下端を取れば確実だが、計算量を意識してきちんと場合分けすると次のようになる。

除算の丸めの向きは、次の表の通り。まず上向き丸めの結果が欲しい場合。

分子の符号	分母の符号	分子の計算の丸め	分母の計算の丸め	除算の計算の丸め
+	+	up	down	up
+	-	down	down	up
-	+	up	up	up
-	-	down	up	up

下向き丸めの結果が欲しい場合は以下の通り (全て逆)。

分子の符号	分母の符号	分子の計算の丸め	分母の計算の丸め	除算の計算の丸め
+	+	down	up	down
+	-	up	up	down
-	+	down	down	down
-	-	up	down	down

これをじつと見ると、 $z_2$  を計算するには、まず分母の符号を見て、それで決まる丸めの向きで分子を計算し、その分子の符号を見てそれで決まる丸めの向きで分母を計算し、最後に除算、という順序でうまく行くことが分かる。無論  $y_1 + y_2$  を  $y_1$  に省略するようなことをしてはいけない。 $-y_2z_1$  の部分の丸めに気を付けながらアルゴリズムを作ると、次のようになる。

```
def dd_div_up(x1, x2, y1, y2) :
    z1 = x1 / y1

    if abs(z1) == float("inf") :
        return z1, 0
    if abs(y1) == float("inf") :
        return z1, 0

    if y1 > 0:
        z3, z4 = twoproduct_up(-z1, y1)
        if abs(z3) == float("inf") :
            z3, z4 = twoproduct_up(-z1, y1 * 0.5)
            up()
            z2 = (((z3 + (x1 * 0.5)) + (-z1) * (y2 * 0.5)) + (x2 * 0.5)) + z4
            if z2 > 0.:
                down()
                tmp = (y1 + y2) * 0.5
                up()
            else :
                tmp = (y1 + y2) * 0.5
            z2 = z2 / tmp
        else:
            up()
            z2 = (((z3 + x1) + (-z1) * y2) + x2) + z4
            if z2 > 0.:
                down()
                tmp = y1 + y2
                up()
            else :
                tmp = y1 + y2
            z2 = z2 / tmp
    else:
        z3, z4 = twoproduct_down(-z1, y1)
        if abs(z3) == float("inf") :
            z3, z4 = twoproduct_down(-z1, y1 * 0.5)
```

```

down()
z2 = (((z3 + (x1 * 0.5)) + (-z1) * (y2 * 0.5)) + (x2 * 0.5)) + z4
if z2 > 0.:
    tmp = (y1 + y2) * 0.5
    up()
else:
    up()
    tmp = (y1 + y2) * 0.5
z2 = z2 / tmp
else:
down()
z2 = (((z3 + x1) + (-z1) * y2) + x2) + z4
if z2 > 0.:
    tmp = y1 + y2
    up()
else:
    up()
    tmp = y1 + y2
z2 = z2 / tmp

near()
z1, z2 = twosum(z1, z2)
return z1, z2

```

```

def dd_div_down(x1, x2, y1, y2) :
z1 = x1 / y1

if abs(z1) == float("inf") :
    return z1, 0
if abs(y1) == float("inf") :
    return z1, 0

if y1 > 0:
z3, z4 = twoproduct_down(-z1, y1)
if abs(z3) == float("inf") :
z3, z4 = twoproduct_down(-z1, y1 * 0.5)
down()
z2 = (((z3 + (x1 * 0.5)) + (-z1) * (y2 * 0.5)) + (x2 * 0.5)) + z4
if z2 > 0.:
    up()
    tmp = (y1 + y2) * 0.5
    down()
else:
    tmp = (y1 + y2) * 0.5
z2 = z2 / tmp
else:
down()
z2 = (((z3 + x1) + (-z1) * y2) + x2) + z4
if z2 > 0.:
    up()
    tmp = y1 + y2
    down()
else:
    tmp = y1 + y2
z2 = z2 / tmp
else:
z3, z4 = twoproduct_up(-z1, y1)
if abs(z3) == float("inf") :
z3, z4 = twoproduct_up(-z1, y1 * 0.5)
up()
z2 = (((z3 + (x1 * 0.5)) + (-z1) * (y2 * 0.5)) + (x2 * 0.5)) + z4
if z2 > 0.:
    tmp = (y1 + y2) * 0.5
    down()
else:
    down()

```

```

        tmp = (y1 + y2) * 0.5
        z2 = z2 / tmp
    else:
        up()
        z2 = (((z3 + x1) + (-z1) * y2) + x2) + z4
        if z2 > 0.:
            tmp = y1 + y2
            down()
        else:
            down()
            tmp = y1 + y2
        z2 = z2 / tmp

    near()
    z1, z2 = twosum(z1, z2)
    return z1, z2

```

## 5.5 平方根

除算と同様、

$$z_2 = \frac{x_1 + x_2 + \text{twoproduct}(z_1, -z_1)}{\sqrt{x_1 + x_2} + z_1}$$

とし、除算と同様に場合分けを行なう。すなわち、

- 上向き丸めの  $z_2$  を得るには、`twoproduct` を含む分子の計算は上向き丸め、分母は分子が正なら下向き丸め、分子が負なら上向き丸め、除算は上向き丸め
- 下向き丸めの  $z_2$  を得るには、`twoproduct` を含む分子の計算は下向き丸め、分母は分子が正なら上向き丸め、分子が負なら下向き丸め、除算は下向き丸め

とすればよい。無論  $\sqrt{x_1 + x_2}$  を  $\sqrt{x_1}$  に省略するようなことをしてはいけない。以上により、次のようになる。

```

def dd_sqrt_up(x1, x2):
    if x1 == 0 and x2 == 0:
        return 0, 0
    if x1 == float("inf"):
        return x1, 0

    z1 = math.sqrt(x1)
    z3, z4 = twoproduct_up(-z1, z1)
    up()
    z2 = (z3 + x1) + x2 + z4
    if z2 > 0.:
        down()
        tmp = math.sqrt(x1 + x2) + z1
        up()
    else:
        tmp = math.sqrt(x1 + x2) + z1
    z2 = z2 / tmp
    near()
    z1, z2 = twosum(z1, z2)
    return z1, z2

```

```

def dd_sqrt_down(x1, x2) :
    if x1 == 0 and x2 == 0:
        return 0, 0
    if x1 == float("inf") :
        return x1, 0

    z1 = math.sqrt(x1)
    z3, z4 = twoproduct_down(-z1, z1)
    down()
    z2 = (z3 + x1) + x2 + z4
    if z2 > 0.:
        up()
        tmp = math.sqrt(x1 + x2) + z1
        down()
    else:
        tmp = math.sqrt(x1 + x2) + z1
    z2 = z2 / tmp
    near()
    z1, z2 = twosum(z1, z2)
    return z1, z2

```

## 6 付録: pythonでの丸めモード変更

本資料のサンプルを python で書いたのは単にアルゴリズムの説明のためであり、python で実装することを推奨しているわけではない。しかし、とりあえず本資料のサンプルをそのまま動かしてみたい人のため、python で丸めモード変更を行なう方法の一例を示しておく。

python 自身は丸めモード変更の機能を持たないため、C 言語でシェアードライブラリ (.so) を作成しておき、それを import する方針で行なう。

python2 と python3 で少し違うので、それぞれ示す。動作確認は ubuntu 14.04 64bit で行った。

### 6.1 python2 での方法

hwround.c

```

#include <Python.h>
#include <fenv.h>

PyObject *near(PyObject *self, PyObject *args)
{
    fesetround(FE_TONEAREST);
    Py_RETURN_NONE;
}

PyObject *down(PyObject *self, PyObject *args)
{
    fesetround(FE_DOWNWARD);
    Py_RETURN_NONE;
}

PyObject *up(PyObject *self, PyObject *args)
{
    fesetround(FE_UPWARD);
    Py_RETURN_NONE;
}

```

```

}

PyObject *chop(PyObject *self, PyObject *args)
{
    fesetround(FE_TOWARDZERO);
    Py_RETURN_NONE;
}

static PyMethodDef hwrroundmethods[] = {
    {"near", near, METH_NOARGS},
    {"down", down, METH_NOARGS},
    {"up", up, METH_NOARGS},
    {"chop", chop, METH_NOARGS},
    {NULL}
};

void inithwrround(void) {
    Py_InitModule("hwrround", hwrroundmethods);
}

```

のようなファイルを作成し、

```

cc -fpic -c hwrround.c -o hwrround.so -O3 -I/usr/include/python2.7
cc -shared hwrround.o -o hwrround.so

```

のようにコンパイルして hwrround.so を作成する。コンパイルは、

setup.py

```

from distutils.core import setup, Extension

module1 = Extension('hwrround', sources = ['hwrround.c'])

setup (name = 'hwrround',
       version = '1.0',
       description = 'This is a package for hardware rounding',
       ext_modules = [module1])

```

のようなファイルを作成して

```

python setup.py build

```

とすると build ディレクトリ以下に作られる、という方法もある。実行は、hwrround.so ファイルをカレントディレクトリに置いて python を起動し、

```

from hwrround import *

```

とすると near(), up() などが使えるようになる。

```

python setup.py install

```

とするとシステム標準の場所にインストールされていつでも使えるようになるが、root 権限がないとインストールできない場合もあろう。



## 6.2 python3 での方法

hwround.c

```
#include <Python.h>
#include <fenv.h>

PyObject *near(PyObject *self, PyObject *args)
{
    fesetround(FE_TONEAREST);
    Py_RETURN_NONE;
}

PyObject *down(PyObject *self, PyObject *args)
{
    fesetround(FE_DOWNWARD);
    Py_RETURN_NONE;
}

PyObject *up(PyObject *self, PyObject *args)
{
    fesetround(FE_UPWARD);
    Py_RETURN_NONE;
}

PyObject *chop(PyObject *self, PyObject *args)
{
    fesetround(FE_TOWARDZERO);
    Py_RETURN_NONE;
}

static PyMethodDef hwroundmethods[] = {
    {"near", near, METH_NOARGS},
    {"down", down, METH_NOARGS},
    {"up", up, METH_NOARGS},
    {"chop", chop, METH_NOARGS},
    {NULL}
};

static struct PyModuleDef hwroundmodule = {
    PyModuleDef_HEAD_INIT,
    "hwround",
    NULL,
    -1,
    hwroundmethods
};

PyMODINIT_FUNC PyInit_hwround(void) {
    return PyModule_Create(&hwroundmodule);
}
```

のようなファイルを作成し、

```
cc -fpic -c hwround.c -o hwround.so -O3 -I/usr/include/python3.4
cc -shared hwround.o -o hwround.so
```

のようにコンパイルして hwround.so を作成する。python3 と setup.py を使って

```
python3 setup.py build
```

としてもよい。

## 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