疑似 4 倍精度数による区間演算とその実装

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

早稲田大学 基幹理工学部 応用数理学科

2021 年 11 月 28 日

概要

kv ライブラリで可能になっている、疑似 4 倍精度数 (通称 double-double, dd) を両端点に持つような区間演算 (`kv::interval<kv::dd>`) の内部アルゴリズムについて説明する。

- kv ライブラリの簡単な紹介、特に区間演算部分について
- twosum と twoproduct
- 疑似 4 倍精度演算 (double-double)
- twosum, twoproduct の問題と改良
- double-double の方向付き丸め実装

kv ライブラリ

- http://verifiedby.me/kv/ で公開中。
- 作成開始は 2007 年秋頃。公開開始は 2013 年 9 月 18 日。最新版は version 0.4.53。
- 言語は C++。boost C++ Libraries (http://www.boost.org/) も必要。
- 全てヘッダファイルで記述されており、インストールはヘッダファイルをどこかに配置するだけ。
- オープンソースである。精度保証付き数値計算の結果が「証明」であると主張するならば、計算に使われたプログラムは必ず公開されているべき。
- 計算に使う数値の型は double に制限されていない。C++のテンプレート機能を用いて容易に変更することが出来る。

kv ライブラリの主な機能

数値型

- double
- 区間演算 (多数の数学関数含む)
- 4 倍精度 (double-double) 演算
- MPFR ラッパー
- 複素数演算
- 自動微分
- affine arithmeric
- ベキ級数演算
- それらの型の組み合わせ

アプリケーション

- 非線形方程式の精度保証
- 非線形方程式の全解探索
- 常微分方程式の初期値問題
- 常微分方程式の境界値問題
- 数値積分 (1, 2 次元)
- 端点特異関数の数値積分
- 特殊関数
- その他

区間演算 (interval)

- 上端下端型の区間演算を行う。
- exp, log, sin, cos, tan, sinh, cosh, tanh, asin, acos, atan, asinh, acosh, atanh, expm1, log1p, abs, pow などの精度保証付きの数学関数を持つ。
- 10 進文字列と double との丸め方向指定付き相互変換を持ち、正しく入出力が出来る。
- 上端と下端に用いる数値型はテンプレートになっており、double 以外の型も使える。例えば double-double 型や MPFR を使える。ただし、上向き下向き双方の丸めに対応した加減乗除、平方根、文字列との相互変換の方法を定義する必要がある。
- サポートする環境は、C99 準拠の fesetround が使えること。
  - X86 CPU の SSE2
  - 最近点丸めのみを用いた方向付き丸めのエミュレーション
  - 最新の Intel CPU の AVX-512
  - FMA 命令

  を使うオプションもある。

区間演算の実装の方針

- IEEE754 標準で信頼できて丸めの向きの変更ができるのは加減乗除と平方根のみ。
- 上向きまたは下向き丸めの加減乗除と平方根のみを用いてアルゴリズムを構成する。
- 精度保証付き数学関数も同様に上向きまたは下向き丸めの加減乗除と平方根のみを用いて実装する。
- 上向きと下向き丸めの加減乗除と平方根が実装されているならば、double でなくとも区間の両端の数として使用できる。dd や mpfr を内部に持つ区間演算ができるのはそのため。

区間演算プログラム (double-double)

$$s = \sum_{k=1}^{1000} \frac{1}{k}$$ を double-double で計算する。

```
#include <kv/interval.hpp> // 区間演算
#include <kv/dd.hpp> // double-double
#include <kv/rdd.hpp> // dd の方向付き丸めを定義

int main()
{
    kv::interval<kv::dd> s, x;

    std::cout.precision(34);

    s = 0;
    for (int i=1; i<=1000; i++) {
        x = i;
        s += 1/x;
    }

    std::cout << s << "\n";
}
```

```
[7.4854708605503449126565182043 08257,7.4854708605503449126565182043 60964]
```

double-double (dd)

- twosum: 2 つの数の和を 2 つの仮数部の重なりのない数の和に変換するアルゴリズム。(例: $1234 + 5.432 \to 1239 + 0.432$)
- twoproduct: 2 つの数の積を 2 つの仮数部の重なりのない数の和に変換するアルゴリズム。(例: $1234 \times 5.432 \to 6703 + 0.088$)
- twosum と twoproduct を組み合わせると、2 つの倍精度数を用いた擬似的な 4 倍精度計算を実現できる。一般的な多倍長演算よりかなり高速。
- 単体 (`dd.hpp`) で使った時は近似計算。
- `dd.hpp` と `rdd.hpp`(方向付き丸めでの `dd` 型の演算を定義) を併用し、`interval` 型の内部型として `dd` 型を使うと、端点に `dd` 型を持つ 4 倍精度区間演算が可能。

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

- すべての変数は IEEE754 倍精度とする。
- 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い。また、y はその誤差 $(a + b) - x$ 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。
- `fasttwosum` (3step) は、`twosum` (6step) 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 = (((a1 * b1 - x) + a2 * b1) + a1 * b2) + a2 * b2
    return x, y
```

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

- すべての変数は IEEE754 倍精度とする。
- 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 $a \times b$ を浮動小数点演算で計算したものに等しい。また、y はその誤差 $(a \times b) - x$ に等しい。すなわち、$a \times b$ 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。
- `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

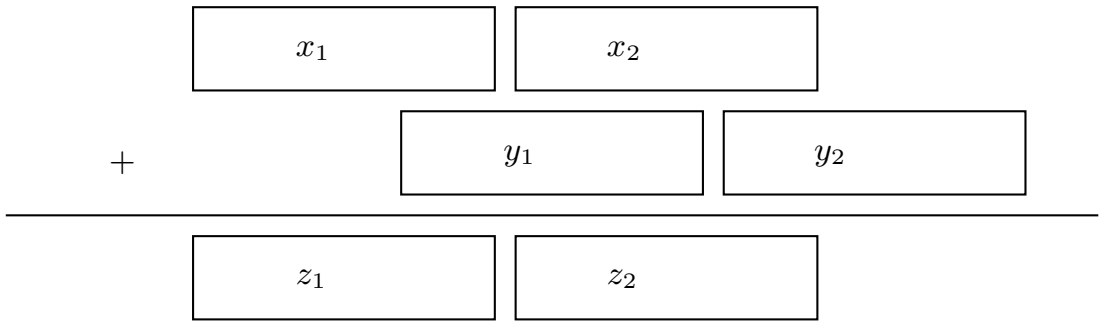
twoproduct アルゴリズム (FMA あり)

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

- もし FMA (Fused Multiply Add) 命令 ($a \times b + c$ を無誤差で計算したものを最近点に丸める) が使えるなら、`twoproduct` はわずか 2step で実装できる。

double-double の加算

```
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)
    if abs(z1) == float("inf") :
        return z1, 0
    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。
- `twosum` 後に z_1 が `inf` になってしまうような場合、オーバーフローとみなして $(\pm\texttt{inf}, 0)$ を返している。

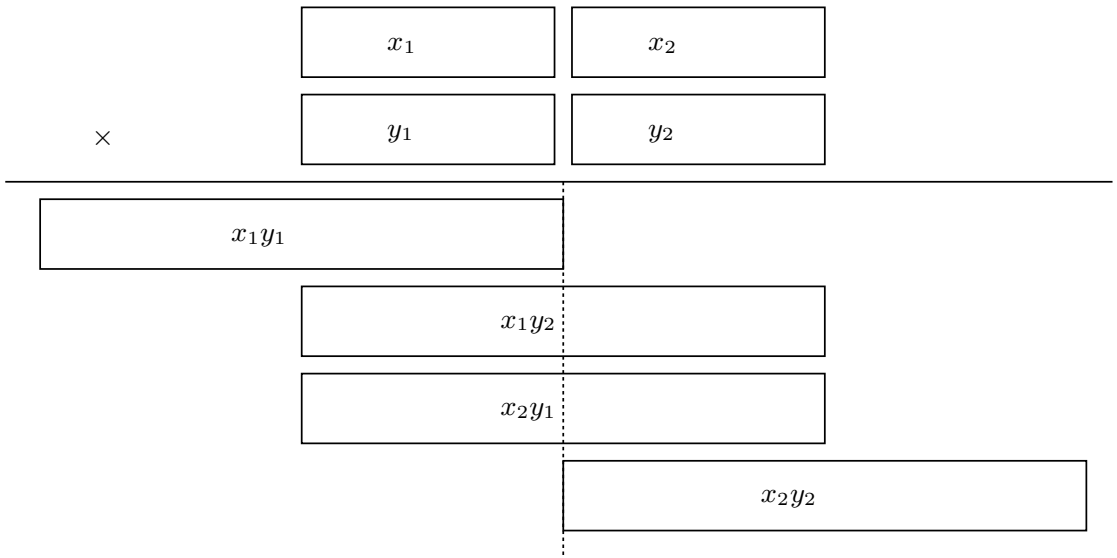
double-double の減算

```
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)
    if abs(z1) == float("inf") :
        return z1, 0
    return z1, z2
```

- 加算とほぼ同様。y_1, y_2 を $-y_1, -y_2$ に置き換えただけ。

double-double の乗算

```
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)
    if abs(z1) == float("inf") :
        return z1, 0
    return z1, z2
```



- x_1y_1 のみ twoproduct を用いてきちんと計算し、それ以外の x_1y_2, x_2y_1, x_2y_2 の項は普通に計算する。
- 近似計算と割り切るなら x_2y_2 は省略可能。($(1+2^{-54})(1-2^{-54})$ のような計算では差が。)

double-double の除算

```
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)
    if abs(z1) == float("inf") :
        return z1, 0
    return z1, z2
```

- $z_1 \simeq x_1/y_1$ を近似計算でまず求め、次に $\dfrac{x_1+x_2}{y_1+y_2} = z_1 + z_2$ となる z_2 の計算を考える。

$$z_2 = \frac{x_1+x_2}{y_1+y_2} - z_1 = \frac{x_1 - y_1 z_1 + x_2 - y_2 z_1}{y_1+y_2}$$

  更に、$-y_1 z_1$ の部分を $z_3, z_4 = \texttt{twoproduct}(y_1, -z_1)$ のように変換し、

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

  とする。(x_1+z_3) の部分はキャンセルするので先に計算する。
- 分母の y_1+y_2 の部分の y_2 は (どうせ消えるので) 省略。
- z_3 がオーバーフローした場合は、z_2 の計算式の分母分子を半分に。

double-double の平方根

```
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
```

$z_1 \simeq \sqrt{x_1}$ 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、$z_3, z_4 = \mathrm{twoproduct}(z_1, -z_1)$ 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。$(x_1 + z_3)$ の部分は先に計算する。分母の $x_1 + x_2$ の部分の x_2 は省略している。

twosum, twoproduct の誤差、異常動作

- 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、指数部には制限がない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く、異常な動作をする場合もある。

twosum の異常動作

```
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。
- オーバーフローは発生する。
- オーバーフローで y が NaN になってしまうケース:

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

- 計算結果はオーバーフローしないが中間変数 tmp がオーバーフローするケース:

```
>>> 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 の誤差、異常動作

```
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 = (((a1 * b1 - x) + a2 * b1) + a1 * b2) + a2 * 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
- $x, y =$ 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。
- $|x| > 2^{-969} - 2^{-1021}$ 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。
- 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, 2.594447525195200 3e+71)
(1.7976931348623157e+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 = ((((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
```

修正版 twoproduct (2)

- `split` のオーバーフローと $a_1 \times b_1$ のオーバーフローに対処した。先の例は、

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

  のようにきちんと計算できる。
- FMA を使った twoproduct の場合は、このような対策は必要ない。
- 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)。FMA 版であっても。`twoproduct` は誤差のある計算と考える必要がある。

double-double を端点に持つ区間演算

- dd_add, dd_sub, dd_mul, dd_div, dd_sqrt の 5 演算について、上向き丸め、下向き丸めのバージョンを作ればよい。
- 上記 5 演算の内部に含まれる、
  - twoproduct
  - 誤差の入る可能性のある箇所

  について、丸めの向きを適切に制御する。
- 特に、twoproduct については上向き丸めで twoproduct を行う twoproduct_up、下向き丸めで twoproduct を行う twoproduct_down を用意する。
- 結果が ±inf になった場合の処理が必要。例えば下向き丸めの double の加算の場合、正の大きい数同士を加えてオーバーフローしても決して inf にはならず、double で表現可能な正の最大数 $(2^{1024}(1-2^{-53}))$ になる。double-double 演算の場合も同様に振る舞うべきで、オーバーフローした場合は double-double で表現可能な正の最大数 $((2^{1024}(1-2^{-53}), 2^{970}(1-2^{-53}))$ のペア) を返すべきである。
- ただし、入力のどちらかが最初から ±inf だった場合は ±inf を返してよい。このため、最初に引数をチェックしどちらかが ±inf だった場合は (double の演算結果, 0) を返す。

twoproduct_up

```
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
```

- アンダーフローの可能性がある部分を上向き丸めで計算する

twoproduct_down

```
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
```

- アンダーフローの可能性がある部分を下向き丸めで計算する

twoproduct_up, twoproduct_down (FMA あり)

```
def twoproductfma_up(a, b) :
    x = a * b
    up()
    y = fma(a, b, -x)
    near()
    return x, y
```

```
def twoproductfma_down(a, b) :
    x = a * b
    down()
    y = fma(a, b, -x)
    near()
    return x, y
```

- FMA があれば複雑な処理は要らない。

dd_add_up

```
def dd_add_up(x1, x2, y1, y2) :
    if abs(x1) == float("inf") or abs(y1) == float("inf") :
        return x1 + y1, 0
    z1, z2 = twosum(x1, y1)
    if z1 == float("inf") :
        return z1, 0
    if z1 == -float("inf") :
        z1 = -sys.float_info.max
        z2 = -sys.float_info.max * 2.**(-54)
    up()
    z2 = z2 + x2 + y2
    near()
    z1, z2 = twosum(z1, z2)
    if z1 == float("inf") :
        return z1, 0
    if z1 == -float("inf") :
        z1 = -sys.float_info.max
        z2 = -sys.float_info.max * 2.**(-54)
    return z1, z2
```

- 誤差の入るの可能性がある部分を上向き丸めで計算する
- 入出力の無限大処理

dd_add_down

```
def dd_add_down(x1, x2, y1, y2) :
    if abs(x1) == float("inf") or abs(y1) == float("inf") :
        return x1 + y1, 0
    z1, z2 = twosum(x1, y1)
    if z1 == -float("inf") :
        return z1, 0
    if z1 == float("inf") :
        z1 = sys.float_info.max
        z2 = sys.float_info.max * 2.**(-54)
    down()
    z2 = z2 + x2 + y2
    near()
    z1, z2 = twosum(z1, z2)
    if z1 == -float("inf") :
        return z1, 0
    if z1 == float("inf") :
        z1 = sys.float_info.max
        z2 = sys.float_info.max * 2.**(-54)
    return z1, z2
```

- 誤差の入るの可能性がある部分を下向き丸めで計算する
- 入出力の無限大処理

dd_sub_up

```
def dd_sub_up(x1, x2, y1, y2) :
    if abs(x1) == float("inf") or abs(y1) == float("inf") :
        return x1 - y1, 0
    z1, z2 = twosum(x1, -y1)
    if z1 == float("inf") :
        return z1, 0
    if z1 == -float("inf") :
        z1 = -sys.float_info.max
        z2 = -sys.float_info.max * 2.**(-54)
    up()
    z2 = z2 + x2 - y2
    near()
    z1, z2 = twosum(z1, z2)
    if z1 == float("inf") :
        return z1, 0
    if z1 == -float("inf") :
        z1 = -sys.float_info.max
        z2 = -sys.float_info.max * 2.**(-54)
    return z1, z2
```

- 誤差の入るの可能性がある部分を上向き丸めで計算する
- 入出力の無限大処理

dd_sub_down

```
def dd_sub_down(x1, x2, y1, y2) :
    if abs(x1) == float("inf") or abs(y1) == float("inf") :
        return x1 - y1, 0
    z1, z2 = twosum(x1, -y1)
    if z1 == -float("inf") :
        return z1, 0
    if z1 == float("inf") :
        z1 = sys.float_info.max
        z2 = sys.float_info.max * 2.**(-54)
    down()
    z2 = z2 + x2 - y2
    near()
    z1, z2 = twosum(z1, z2)
    if z1 == -float("inf") :
        return z1, 0
    if z1 == float("inf") :
        z1 = sys.float_info.max
        z2 = sys.float_info.max * 2.**(-54)
    return z1, z2
```

- 誤差の入るの可能性がある部分を下向き丸めで計算する
- 入出力の無限大処理

dd_mul_up

```
def dd_mul_up(x1, x2, y1, y2) :
    if abs(x1) == float("inf") or abs(y1) == float("inf") :
        return x1 * y1, 0
    z1, z2 = twoproduct_up(x1, y1)
    if z1 == float("inf") :
        return z1, 0
    if z1 == -float("inf") :
        z1 = -sys.float_info.max
        z2 = -sys.float_info.max * 2.**(-54)
    up()
    z2 = z2 + x1 * y2 + x2 * y1 + x2 * y2
    near()
    z1, z2 = twosum(z1, z2)
    if z1 == float("inf") :
        return z1, 0
    if z1 == -float("inf") :
        z1 = -sys.float_info.max
        z2 = -sys.float_info.max * 2.**(-54)
    return z1, z2
```

- twoproduct_up を呼び出す。
- 誤差の入るの可能性がある部分を上向き丸めで計算する
- 入出力の無限大処理

dd_mul_down

```
def dd_mul_down(x1, x2, y1, y2) :
    if abs(x1) == float("inf") or abs(y1) == float("inf") :
        return x1 * y1, 0
    z1, z2 = twoproduct_down(x1, y1)
    if z1 == -float("inf") :
        return z1, 0
    if z1 == float("inf") :
        z1 = sys.float_info.max
        z2 = sys.float_info.max * 2.**(-54)
    down()
    z2 = z2 + x1 * y2 + x2 * y1 + x2 * y2
    near()
    z1, z2 = twosum(z1, z2)
    if z1 == -float("inf") :
        return z1, 0
    if z1 == float("inf") :
        z1 = sys.float_info.max
        z2 = sys.float_info.max * 2.**(-54)
    return z1, z2
```

- twoproduct_down を呼び出す。
- 誤差の入るの可能性がある部分を下向き丸めで計算する
- 入出力の無限大処理

dd_div_up

```
def dd_div_up(x1, x2, y1, y2) :
    if abs(x1) == float("inf") or abs(y1)
        == float("inf") :
        return x1 / y1, 0
    z1 = x1 / y1

    if z1 == float("inf") :
        return z1, 0
    if z1 == -float("inf") :
        z1 = -sys.float_info.max
        z2 = -sys.float_info.max *
            2.**(-54)

    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)

    if z1 == float("inf") :
        return z1, 0
    if z1 == -float("inf") :
        z1 = -sys.float_info.max
        z2 = -sys.float_info.max *
            2.**(-54)

    return z1, z2
```

- $z_2 = \dfrac{(x_1 + z_3) + z_4 + x_2 - y_2 z_1}{y_1 + y_2}$ が上向き丸めになるように計算。分母の y_2 の省略は不可。

dd_div_down

```
def dd_div_down(x1, x2, y1, y2):
    if abs(x1) == float("inf") or abs(y1)
        == float("inf"):
        return x1 / y1, 0
    z1 = x1 / y1

    if z1 == -float("inf"):
        return z1, 0
    if z1 == float("inf"):
        z1 = sys.float_info.max
        z2 = sys.float_info.max * 2.**(-54)

    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)

    if z1 == -float("inf"):
        return z1, 0
    if z1 == float("inf"):
        z1 = sys.float_info.max
        z2 = sys.float_info.max * 2.**(-54)

    return z1, z2
```

- $z_2 = \dfrac{(x_1 + z_3) + z_4 + x_2 - y_2 z_1}{y_1 + y_2}$ が下向き丸めになるように計算。分母の y_2 の省略は不可。

dd_sqrt_up

```
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:
        if x1 == sys.float_info.max:
            tmp = math.sqrt(x1*0.25 + x2*0.25)*2 + z1
        else:
            tmp = math.sqrt(x1 + x2) + z1
    z2 = z2 / tmp
    near()
    z1, z2 = twosum(z1, z2)
    return z1, z2
```

- $z_2 = \dfrac{(x_1 + z_3) + x_2 + z_4}{\sqrt{x_1 + x_2} + z_1}$ が上向き丸めになるように計算。分母の x_2 の省略は不可。
- x_1 が正の最大値だった場合の $x_1 + x_2$ のオーバーフローを避けている。

dd_sqrt_down

```
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()
        if x1 == sys.float_info.max:
            tmp = math.sqrt(x1*0.25 + x2*0.25)*2 + z1
        else:
            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
```

- $z_2 = \dfrac{(x_1 + z_3) + x_2 + z_4}{\sqrt{x_1 + x_2} + z_1}$ が下向き丸めになるように計算。分母の x_2 の省略は不可。
- x_1 が正の最大値だった場合の $x_1 + x_2$ のオーバーフローを避けている。

最後に宣伝

kv ライブラリ

- http://verifiedby.me/kv/ で公開中。
- 言語は C++。boost C++ Libraries も必要。
- 全てヘッダファイルで記述されており、インストールはヘッダファイルをどこかに配置するだけ。
- オープンソースである。精度保証付き数値計算の結果が「証明」であると主張するならば、計算に使われたプログラムは必ず公開されているべき。
- 計算に使う数値の型は double に制限されていない。C++のテンプレート機能を用いて容易に変更することが出来る。
- (数値型) 区間演算 (多数の数学関数含む)、4 倍精度 (double-double) 演算、MPFR ラッパー、複素数演算、自動微分、affine arithmeric、ベキ級数演算、とそれらの組み合わせ。
- (アプリケーション) Krawczyk 法による非線形方程式の精度保証、非線形方程式の全解探索、常微分方程式の初期値問題、常微分方程式の境界値問題、数値積分、特殊関数、他

皆様のご利用をお待ちしております!