Double-Double(DD) および Quad-Double(QD) 演算ライブラリ *

Yozo Hida† Xiaoye S. Li‡ David H. Bailey‡

Original documentation: May 8, 2008
日本語訳: 2016 年 2 月 9 日 §

概 要

Double-Double(DD) 型は，二つの IEEE 倍精度 (double) 型データから構成されており，106 ビットの有効精度を保持することができる。同様に，Quad-double(QD) 型は四つの倍精度型データから構成されており，212 ビットの有効精度を持つ。この二つのデータ型に対しては，様々な演算 (四則演算や平方根，初等関数など) が可能であり，そのためのアルゴリズムも本稿で解説する。C, Fortran 用のインターフェースと共に，これらのアルゴリズムの C++実装についても触れる。本ライブラリの性能についても最後に議論する。

*この研究は the Director, Office of Science, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy の契約番号 DE-AC03-76SF00098 の援助の元で行われた。

†Computer Science Division, University of California, Berkeley, CA 94720 (yozo@cs.berkeley.edu).

‡NERSC, Lawrence Berkeley National Laboratory, 1 Cycloton Rd, Berkeley, CA 94720 (xiaoye@nersc.gov, dhbailey@lbl.gov).

§訳注）日本語訳は幸谷智紀 (tkouya@cs.sist.ac.jp) による。

目 次

1 初めに

多倍長計算は，純粋数学，数学定数の研究，暗号化，計算幾何学等，様々な分野で活用されている。このため，固定精度桁数のデータ型を用いて，多くの任意精度演算アルゴリズムやライブラリが開発されてきた。これらのライブラリは，多倍長数の表現方法によって二つに分類される。一つは多数桁方式 (multiple-digit) で，一つの指数部と，多数の桁数がセットになったものである。例えば数式処理ソフトウェアの Mathematica，Bailey の MPFUN [2]，Brent の MP [3]，GNU MP [7] がこれにあたる。対して，多数のコンポーネント (multiple-component) から構成される形式もあり，これは通常の浮動小数点型を複数組み合わせて表現するもので，それぞれのコンポーネントごとに指数部と仮数部を持つ。この実装例としては [6, 10, 11] がある。多数桁方式は広範囲の数を表現することができる一方，多数コンポーネント方式は演算の高速性に利がある。

多くのアプリケーションでは，普通に使用される精度の数倍 (せいぜい 2 倍か 4 倍) 程度の利用で十分であり，任意精度を必要としていない。この種の「固定化」した精度の計算は，任意精度の計算よりも確実に高速化できる。Bailey [1] や Briggs [4] は，倍精度を二つ使うことによって,「double-double(DD, 倍々)」精度を実現するアルゴリズムとソフトウェアを作り上げている。この多数コンポーネント方式では，二つの倍精度データを上位桁と下位桁に割り当て，加算することなくそのままの形で利用する。

本稿では `qd` ライブラリで使われているアルゴリズムを解説する。このライブラリでは double-double(DD) 演算と quad-double(QD) 演算の両方をサポートしている。QD データ型の変数 a は (a_0, a_1, a_2, a_3) という形式で与えられる。これは $a = a_0 + a_1 + a_2 + a_3$ を正確に誤差なく表現しており，a_0 は最上位部の桁を表現するコンポーネントを意味する。本ライブラリでは，四則演算だけでなく，平方根などの代数演算や初等関数もアルゴリズムを構築し実装している。これらの機能は，任意精度演算パッケージ MPFUN を利用して広範囲にわたる比較検証を行ってある。この qd ライブラリは `http://www.nersc.gov/~dhbailey/mpdist/mpdist.html`[1] にて配布している。このライブラリは，並列流体シミュレーションのプログラム [2] にも組み込まれており，詳細は [8] で見ることができる。

本稿は下記のような構成になっている。まず，2 節では IEEE 浮動小数点演算の特徴と，QD 演算のアルゴリズム構成について述べる。次に，3 節では QD の四則演算アルゴリズム，即ち，再正規化 (renormalization)，加算，乗算，除算について触れる。4 節と 5 節では実装してある代数演算と初等関数のアルゴリズムについて触れる。6 節ではそれ以外のルーチンについての解説を行う。7 節ではこれらのアルゴリズムを実装した C++ライブラリについて簡単に解説する。8 節では，実装した主要な演算が異なる計算機環境でどのぐらいの計算時間を要するのかを示す。最後の 9 節では今後の課題について議論する。

2 準備

この節では，QD 演算で使用している IEEE 浮動小数点演算の性質とアルゴリズムを示す。ここで述べられている成果は Dekker [6], Knuth [9], Priest [10], Shewchuk [11] 等の研究によるものである。

[1] 訳注) 現在は `http://crd-legacy.lbl.gov/~dhbailey/mpdist/` で配布している。

[2] 訳注) parallel vortex roll-up simulation code の訳。もっといい訳語を知っている方はお知らせ下さい。

実際，以下で示すアルゴリズムや解説のための図は Shewchuk の論文からの引用（多少改変あり）である。

全ての基本演算は IEEE 倍精度浮動小数点型を用いて行われ，偶数丸めを利用するものとする。2 項演算子 $\cdot \in \{+, -, \times, /\}$ に対しては，$\mathrm{fl}(a \cdot b) = a \odot b$ という記法を使用し，演算 $a \cdot b$ の浮動小数点数による結果と，その誤差 $\mathrm{err}(a \cdot b)$ を表記する。この時，$a \cdot b = \mathrm{fl}(a \cdot b) + \mathrm{err}(a \cdot b)$ という等式が成立する。本稿では IEEE754 倍精度のマシンイプシロンを $\varepsilon = 2^{-53}$ と表記し，$\varepsilon_{\mathrm{qd}} = 2^{-211}$ は QD 数のマシンイプシロンとして使用する。

補題 **1.** [11, p. 310] a, b は二つの p ビット長の浮動小数点数で，$|a| \geq |b|$ とする。この時，$|\mathrm{err}(a+b)| \leq |b| \leq |a|$ となる。

補題 **2.** [11, p. 311] a, b は二つの p ビット長の浮動小数点数とする。この時，$|\mathrm{err}(a+b)| = (a+b) - \mathrm{fl}(a+b)$ は p ビット長の浮動小数点数として表現できる。

アルゴリズム **3.** [11, p. 312] 下記のアルゴリズムは，$|a| \geq |b|$ である時に，$s = \mathrm{fl}(a+b)$ と $e = \mathrm{err}(a+b)$ を計算するものである，

QUICK-TWO-SUM(a, b)
1. $s \leftarrow a \oplus b$
2. $e \leftarrow b \ominus (s \ominus a)$
3. **return** (s, e)

アルゴリズム **4.** [11, p. 314] 下記は $s = \mathrm{fl}(a+b)$ と $e = \mathrm{err}(a+b)$ を計算するアルゴリズムである。if 文を回避する代わりに 3 回の浮動小数点演算を使用している。

TWO-SUM(a, b)
1. $s \leftarrow a \oplus b$
2. $v \leftarrow s \ominus a$
3. $e \leftarrow (a \ominus (s \ominus v)) \oplus (b \ominus v)$
4. **return** (s, e)

アルゴリズム **5.** [11, p. 325] 以下は，53 ビット長の IEEE 倍精度浮動小数点数 a を，26 ビット長の仮数部を持つ a_{hi} と a_{lo} に分割するアルゴリズムである。a_{hi} には先頭桁から 26 ビット分を，a_{lo} には後半桁 26 ビット分を保持する [3]。

SPLIT(a)
1. $t \leftarrow (2^{27} + 1) \otimes a$
2. $a_{\mathrm{hi}} \leftarrow t \ominus (t \ominus a)$
3. $a_{\mathrm{lo}} \leftarrow a \ominus a_{\mathrm{hi}}$
4. **return** $(a_{\mathrm{hi}}, a_{\mathrm{lo}})$

アルゴリズム **6.** [11, p. 326] 下記のアルゴリズムは $p = \mathrm{fl}(a \times b)$ と $e = \mathrm{err}(a \times b)$ を計算するものである。

[3] 訳注) ケチ表現 1 ビット分は a_{hi} に入る？

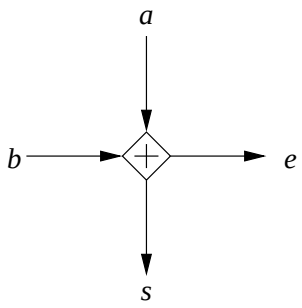

図 1: Quick-Two-Sum

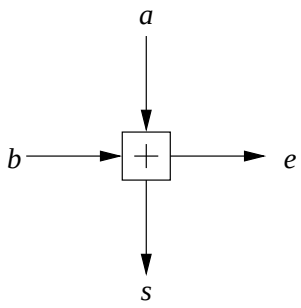

図 2: Two-Sum

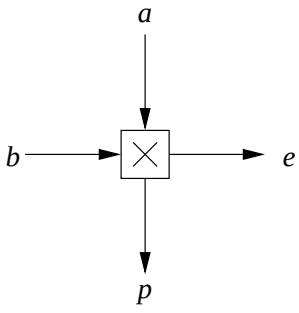

図 3: Two-Prod

Two-Prod(a, b)

1. $p \leftarrow a \otimes b$
2. $(a_{\mathrm{hi}}, a_{\mathrm{lo}}) \leftarrow$ Split(a)
3. $(b_{\mathrm{hi}}, b_{\mathrm{lo}}) \leftarrow$ Split(b)
4. $e \leftarrow ((a_{\mathrm{hi}} \otimes b_{\mathrm{hi}} \ominus p) \oplus a_{\mathrm{hi}} \otimes b_{\mathrm{lo}} \oplus a_{\mathrm{lo}} \otimes b_{\mathrm{hi}}) \oplus a_{\mathrm{lo}} \otimes b_{\mathrm{lo}}$
5. **return** (p, e)

現在のマシンでは複合積和演算 (FMA, fused multiply-add)，即ち $a \times b \pm c$ が一命令で実行できるので，丸め誤差は 1 回出るだけで済む。この利点を生かすと乗算が高速化できる。例えば，IBM Power シリーズ (Power PC 等) ではこの高速化が可能である [4]。

アルゴリズム **7.** 下記は，FMA 命令が使えるマシン上で $p = \mathrm{fl}(a \times b)$ と $e = \mathrm{err}(a \times b)$ を計算するアルゴリズムである。コンパイラによっては $a \times b - c$ ではなく $a \times b + c$ としか計算できないものもあるため，その際には符号を変えて実行する必要がある。

Two-Prod-FMA(a, b)

1. $p \leftarrow a \otimes b$
2. $e \leftarrow \mathrm{fl}(a \times b - p)$
3. **return** (p, e)

以上，提示してきたアルゴリズムは QD 演算のための基本単位となるもので，図 1, 図 2, 図 3 のように表現される。通常の倍精度の加算や乗算は図 4 のように表現される。

3 基本演算

3.1 再正規化

QD 数は加算処理していない 4 つの IEEE 倍精度数の和として表現されている。ある QD 数が (a_0, a_1, a_2, a_3) であれば，これは誤差なしの $a = a_0 + a_1 + a_2 + a_3$ という和を意味している。表現可

[4]訳注) 最近の Intel CPU でも fma 関数が使えるようになっている。

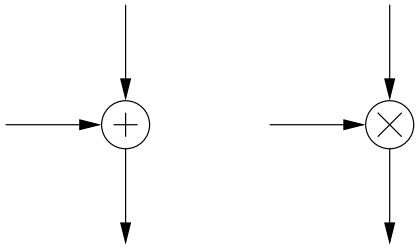

図 4: 通常の IEEE 倍精度の加算と乗算

能な数 x に対しては例外なく，加算処理していない和の表現が多数存在する。従って，(a_0, a_1, a_2, a_3) という倍精度の 4 連は

$$|a_{i+1}| \leq \frac{1}{2}\mathrm{ulp}(a_i) \text{ (for } i = 0, 1, 2)$$

という不等式も満足していなければならない。等号が成立するのは $a_i = 0$ であるか， a_i の最下位ビットがゼロの時だけである (即ち，偶数丸めが使われている時のみ)。最初の a_0 は QD 数 a の倍精度近似値であり，ulp の半分程度の誤差を持つ。

補題 **8.** 任意の QD 数 $a = (a_0, a_1, a_2, a_3)$ の正規化表現は一つに決まる。

ここで述べられている多くのアルゴリズムは標準形式ではない展開式を生成する，つまり，その数を表現するためのビットに重なりが生じうることになる。従って，5 項から成る展開形式を作っておき，これに基づいて 4 項で済むように再正規化する。

アルゴリズム **9.** この再正規化生成法は，Priest の方法 [10, p. 116] を改変したものである。入力値は 5 項の展開形式で，重複ビットは限定してあり，a_0 は最上位桁のコンポーネントとなっている。

RENORMALIZE$(a_0, a_1, a_2, a_3, a_4)$

1. $(s, t_4) \leftarrow$ QUICK-TWO-SUM(a_3, a_4)
2. $(s, t_3) \leftarrow$ QUICK-TWO-SUM(a_2, s)
3. $(s, t_2) \leftarrow$ QUICK-TWO-SUM(a_1, s)
4. $(t_0, t_1) \leftarrow$ QUICK-TWO-SUM(a_0, s)
5. $s \leftarrow t_0$
6. $k \leftarrow 0$
7. **for**[1] $i \leftarrow 1, 2, 3, 4$
8. $\quad (s, e) \leftarrow$ QUICK-TWO-SUM(s, t_i)
9. $\quad$ **if** $e \neq 0$
10. $\quad\quad b_k \leftarrow s$
11. $\quad\quad s \leftarrow e$
12. $\quad\quad k \leftarrow k + 1$
13. $\quad$ **end if**
14. **end for**
15. **return** (b_0, b_1, b_2, b_3)

この再正規化アルゴリズムが正しく実行できるための必要条件は残念ながら知られていない。Priest は，入力値に 51 ビット以上の重複がない場合は正しく再正規化できることを証明している。しかし，この条件は不要である。というのも，再正規化アルゴリズム (アルゴリズム 9) は，以下で示すアルゴリズムが作り出す値すべてに対して正しく動作しているからである。

3.2 加算

3.2.1 Quad-Double + Double

QD 数に倍精度の値を加算する方法は，Shewchuk の GROW-EXPANSION [11, p. 316] と類似のものである。倍精度値 b は QD 値 a に，最下位桁からではなく，最上位桁から加算される。これによって 5 項の展開式が正しい演算結果として生成され，その後で再正規化される。詳細は下記の図 5 を参照のこと。

正確な演算結果が出て再正規化によって 4 項にまとめられるので，最低でも 212 ビット (=53 ビット × 4) は正しい値が出てくることになる。

3.2.2 Quad-Double + Quad-Double

QD+QD の計算用には 2 つのアルゴリズムが提供されている。一つは高速だが (Cray 形式の) 誤差限界 が弱くなるアルゴリズムで， $a \oplus b = (1+\delta_1)a + (1+\delta_2)b$ となる。ここで δ_1 と δ_2 は $\varepsilon_{\mathrm{qd}} = 2^{-211}$ 以下で抑えられる定数である。

[1] この実装では，ループ部分を複数の `if` 文として展開してある。

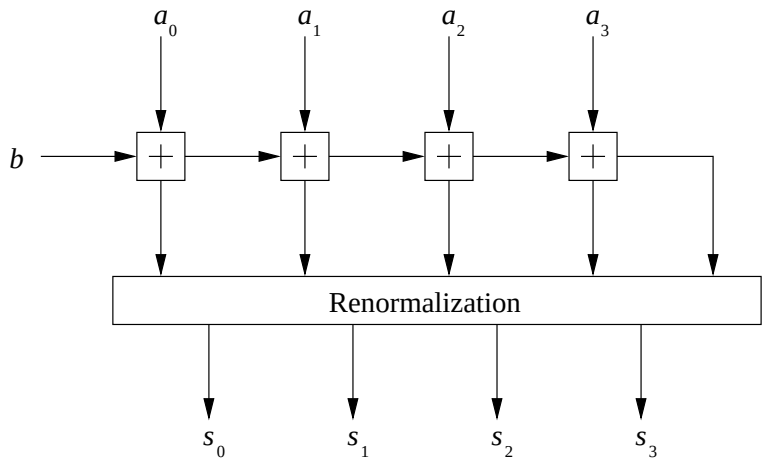

図 5: Quad-Double + Double

図 6 はこの一つ目の高速アルゴリズムを表現したもので，3 つの入力値を取って出力するボックスがある。このような Three-Sum ボックスについては図 7 に示してある。

他にも幾つか補題を示しておく。

補題 **10.** a と b を二つの倍精度浮動小数点数とし，$M = \max(|a|, |b|)$ とする。この時 $|\text{fl}(a+b)| \leq 2M$ となり，結果として $|\text{err}(a+b)| \leq \frac{1}{2}\text{ulp}(2M) \leq 2\varepsilon M$ という不等式が成立する。

補題 **11.** x, y, z を Three-Sum の入力値とし，図 7 の一番左の図中の u, v, w, r_0, r_1, r_2 があるものとする。更に $M = \max(|x|, |y|, |z|)$ とする。この時，$|r_0| \leq 4M$, $|r_1| \leq 8\varepsilon M$, $|r_2| \leq 8\varepsilon^2 M$ という不等式が成立する。

Proof. 補題 10 を Two-Sum 部分にそれぞれ適用することでこの不等式が得られる。まず Two-Sum で $|u| \leq 2M$ となり，次に $|v| \leq 2\varepsilon M$，そして Two-Sum (u と z を加算) で，$|r_0| \leq 4M$ と $|w| \leq 4\varepsilon M$ が得られる。最終的に，最後の Two-Sum でこの補題の不等式を得る。 □

図 7 の中央と右の Three-Sum 演算は最初の Three-Sum の単純化であることに留意すること。ここでこの二つの Three-Sum は 3 つのコンポーネントを出力するものではなく，1 もしくは 2 コンポーネントの出力しかしていない。

上記の誤差限界はそれほどタイトなものではない。$|r_0|$ は $3M$ (もしくは $|x| + |y| + |z|$) に近接したものであるし，r_1 と r_2 の誤差限界はもっと小さくなる。しかしながら，この誤差限界は次の補題を満足するものになっている。

補題 **12.** 図 6 で示した QD 加算アルゴリズムにおいて，再正規化前の 5 項展開値の誤差は $\varepsilon_{\text{qd}} M$ 以下である。ここで $M = \max(|a|, |b|)$ である。

Proof. この補題は，図 6 にある全ての Two-Sum と Three-Sum に，補題 10 と補題 11 と適用することでうまく証明することができる。詳細は付録 A を参照のこと。 □

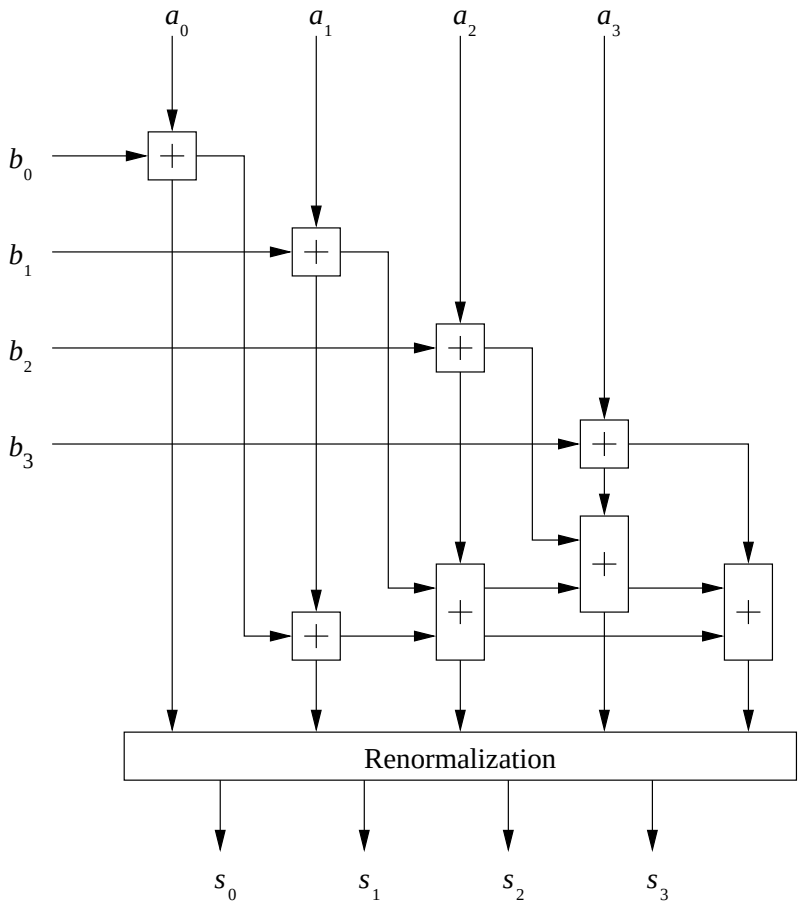

図 6: Quad-Double + Quad-Double

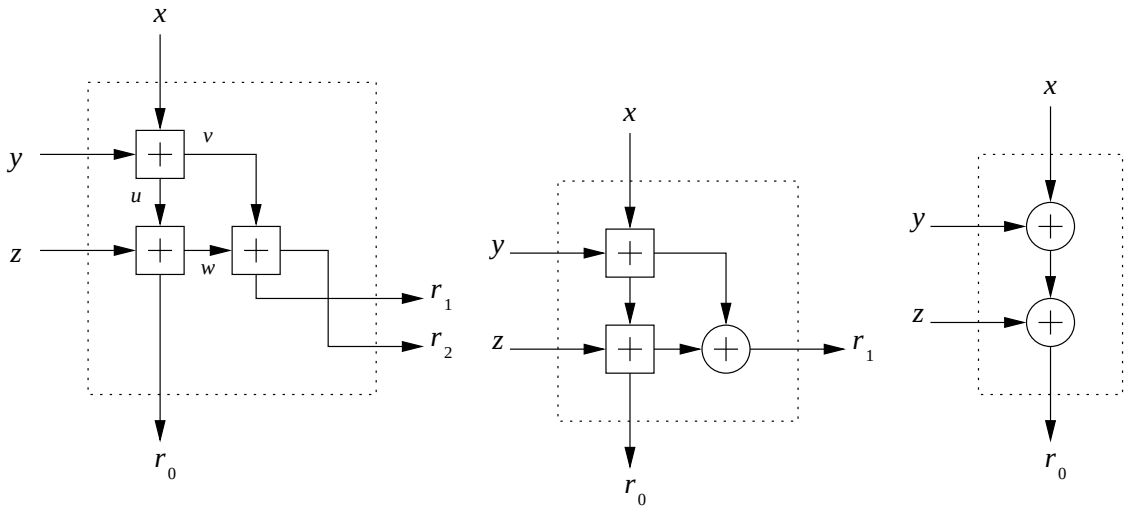

図 7: THREE-SUMS

再正規化によって（証明すべき点は残るものの），下記の誤差限界を得ることができる。

$$\mathrm{fl}(a+b) = (1+\delta_1)a + (1+\delta_2)b \qquad \text{ここで} \quad |\delta_1|, |\delta_2| \le \varepsilon_{\mathrm{qd}}.$$

上記の加算アルゴリズムは，特に命令レベルの並列性に優れた現代のプロセッサに適していることに留意されたい。ことに，最初の 4 つの TWO-SUM は並列に実行できる。再正規化前の部分に条件分岐がないことも，プロセッサのパイプラインをフルに活用できることを示している。

上記のアルゴリズムは，下記の IEEE スタイルの誤差限界を満足していないことに注意しておく必要がある。

$$\mathrm{fl}(a+b) = (1+\delta)(a+b) \qquad \text{ここで} \quad |\delta| \le 2\varepsilon_{\mathrm{qd}}$$

これを確認するために，$a = (u, v, w, x)$ と $b = (-u, -v, y, z)$ を考える。ここで，w, x, y, z には重複ビットはなく，かつ $|w| > |x| > |y| > |z|$ であるとする。この時，上記のアルゴリズムは，厳格な誤差限界を得るために必要となる $c = (w, x, y, z)$ という値ではなく，$c = (w, x, y, 0)$ を計算していることになる。

2 番目のアルゴリズムは，J. Shewchuk と S. Boldo が提案したもので，演算結果の最初の 4 つのコンポーネントを正確に計算する。従って誤差限界

$$\mathrm{fl}(a+b) = (1+\delta)(a+b) \qquad \text{ここで} \quad |\delta| \le 2\varepsilon_{\mathrm{qd}}$$

を厳格に満足している。その分，確実に低速になってしまう (2〜3.5 倍遅くなる)。

このアルゴリズムは Shewchuk の FAST-EXPANSION-SUM[11, p. 320] と同様のものであり，二つの QD 値のマージソートを行っている。有効桁数の欠落を防ぐため，倍の長さのアキュムレータ（積算器）を使い，入力値が小さすぎて演算に影響しないものであっても，1 コンポーネント分は出力するにようにしてある。

アルゴリズム **13.** u, v を 2 項展開値とする。下記のアルゴリズムは $(u, v) + x$ という和を計算し，2 つ以上の有効倍精度コンポーネントを持つ時には，1 つの倍精度コンポーネント s のみを出力する。u と v は，和を構成する残りの二つのコンポーネントに置き換えられる。

DOUBLE-ACCUMULATE(u, v, x)
1. $(s, v) \leftarrow$ TWO-SUM(v, x)
2. $(s, u) \leftarrow$ TWO-SUM(u, s)
3. **if** $u = 0$
4. $\quad u \leftarrow s$
5. $\quad s \leftarrow 0$
6. **end if**
7. **if** $v = 0$
8. $\quad v \leftarrow u$
9. $\quad u \leftarrow s$
10. $\quad s \leftarrow 0$
11. **end if**
12. **return** (s, u, v)

精度の良い加算は下記のアルゴリズムで実行できる。

アルゴリズム **14.** このアルゴリズムは $a = (a_0, a_1, a_2, a_3)$ と $b = (b_0, b_1, b_2, b_3)$ の和を計算する。基本的には 8 つの倍精度値のマージソートを実行し，4 つの倍精度コンポーネントが得られるまで DOUBLE-ACCUMULATE 演算を繰り返す。

QD-ADD-ACCURATE(a, b)
1. $(x_0, x_1, \ldots, x_7) \leftarrow$ MERGE-SORT$(a_0, a_1, a_2, a_3, b_0, b_1, b_2, b_3)$
2. $u \leftarrow 0$
3. $v \leftarrow 0$
4. $k \leftarrow 0$
5. $i \leftarrow 0$
6. **while** $k < 4$ **and** $i < 8$ **do**
7. $\quad (s, u, v) \leftarrow$ DOUBLE-ACCUMULATE(u, v, x_i)
8. $\quad$ **if** $s \neq 0$
9. $\quad\quad c_k \leftarrow s$
10. $\quad\quad k \leftarrow k + 1$
11. $\quad$ **end if**
12. $\quad i \leftarrow i + 1$
13. **end while**
14. **if** $k < 2$ **then** $c_{k+1} \leftarrow v$
15. **if** $k < 3$ **then** $c_k \leftarrow u$
16. **return** RENORMALIZE(c_0, c_1, c_2, c_3)

3.3 減算

減算 $a - b$ は，$a + (-b)$ という加算として実行する。従ってアルゴリズムも特性も加算と同じである。QD 数の正負の変更は単純に 4 つの倍精度コンポーネントごとに行う。C++コンパイラではインライン関数として実行するので，オーバーヘッドはあるがそれほど大きなものではない (5%程度らしい)。

3.4 乗算

乗算は基本的に筆算と同じ，つまり，各項ごとに掛算を行って足し込んでいく方式である。加算と違って桁落ち生じることはないので，乗算に伴う誤差は IEEE スタイルの誤差限界，即ち $a \otimes b = (1+\delta)(a \times b)$ である。ここで δ は $\varepsilon_{\mathrm{qd}}$ で抑えられる値である。

3.4.1 Quad-Double × Double

$a = (a_0, a_1, a_2, a_3)$ を QD 数とし，b を倍精度数とする。この時，この積は 4 項の和，つまり $a_0 b + a_1 b + a_2 b + a_3 b$ となる。$|a_3| \leq \varepsilon^3 |a_0|$, つまり，$|a_3 b| \leq \varepsilon^3 |a_0 b|$, であるから，積 $a_3 b$ の頭から 53 ビット分だけが計算に必要なものとなる。この積和の他の 3 項分は Two-Prod (もしくは Two-Prod-FMA) を使って正確に求められる。この時，全ての項は同様のやり方で加算されていく。詳細は図 8 参照。

3.4.2 Quad-Double × Quad-Double

二つの QD 数の乗算は少し複雑にはなるが，考え方は同じである。 $a = (a_0, a_1, a_2, a_3)$，$b = (b_0, b_1, b_2, b_3)$ が共に QD 数であるとする。一般性を失うことなく，a と b は $O(1)$ であると仮定する。

乗算後に，$O(\varepsilon^4)$ より大きいオーダーの 13 項分の加算が必要になる。

$$
\begin{aligned}
a \times b \;\approx\; & a_0 b_0 && O(1) \text{ の項} \\
& + a_0 b_1 + a_1 b_0 && O(\varepsilon) \text{ の項} \\
& + a_0 b_2 + a_1 b_1 + a_2 b_0 && O(\varepsilon^2) \text{ の項} \\
& + a_0 b_3 + a_1 b_2 + a_2 b_1 + a_3 b_0 && O(\varepsilon^3) \text{ の項} \\
& + a_1 b_3 + a_2 b_2 + a_3 b_1 && O(\varepsilon^4) \text{ の項}
\end{aligned}
$$

より小さいオーダーの項（例えば $a_2 b_3$ は $O(\varepsilon^5)$ の項) は加算することができなくなる。というのも，先頭の 212 ビット分に届かないからである。$O(\varepsilon^4)$ の項は最初の数ビット分のみ必要になる程度で済むため，通常の倍精度演算で計算する。

$i + j \leq 3$ の時は，(p_{ij}, q_{ij}) =Two-Prod(a_i, b_j) とする。さすれば，$p_{ij} = O(\varepsilon^{i+j})$ かつ $q_{ij} = O(\varepsilon^{i+j+1})$ となり，結果として $O(1)$ となる 1 項 (p_{00})，$O(\varepsilon)$ となる 3 項 (p_{01}, p_{10}, q_{00})，$O(\varepsilon^2)$，$O(\varepsilon^2)$ となる 5 項 $(p_{02}, p_{11}, p_{20}, q_{01}, q_{10})$，そして $O(\varepsilon^4)$ となる 7 項が発生する。

こうして，各オーダーごとに全ての項を足し込むことができるようになるので，$O(\varepsilon)$ の項から加算をスタートしていく（図 9 参照)。

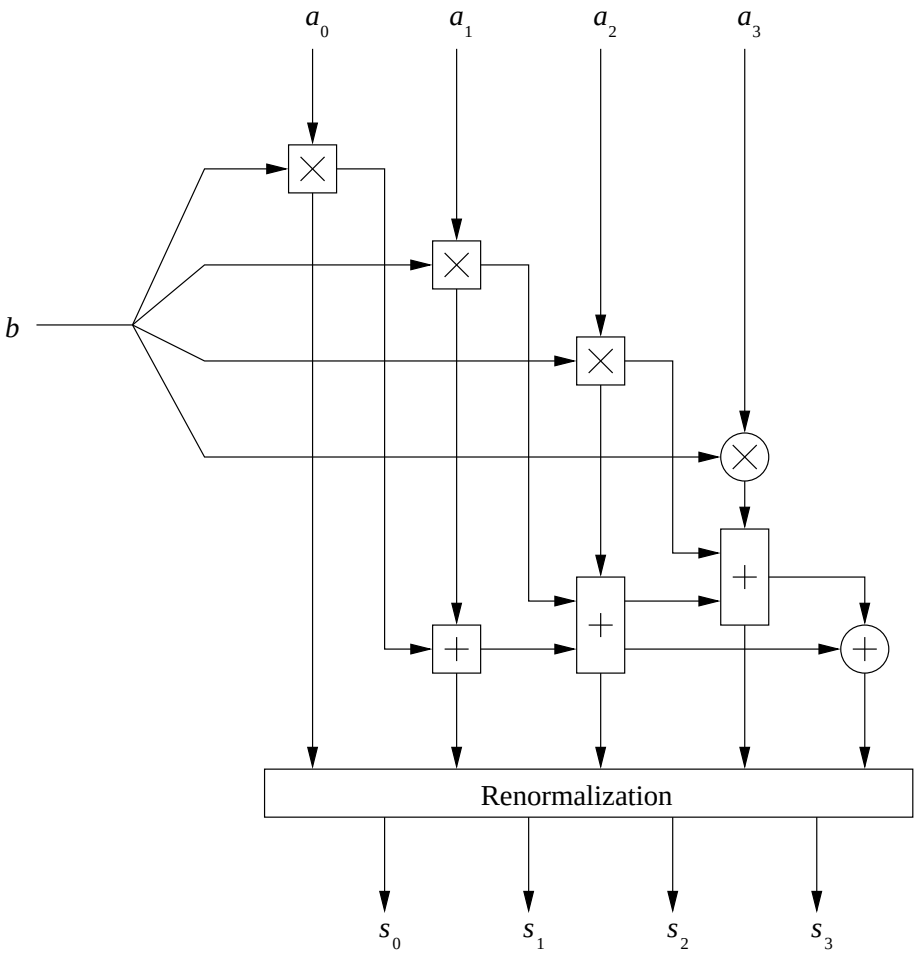

図 8: Quad-Double × Double

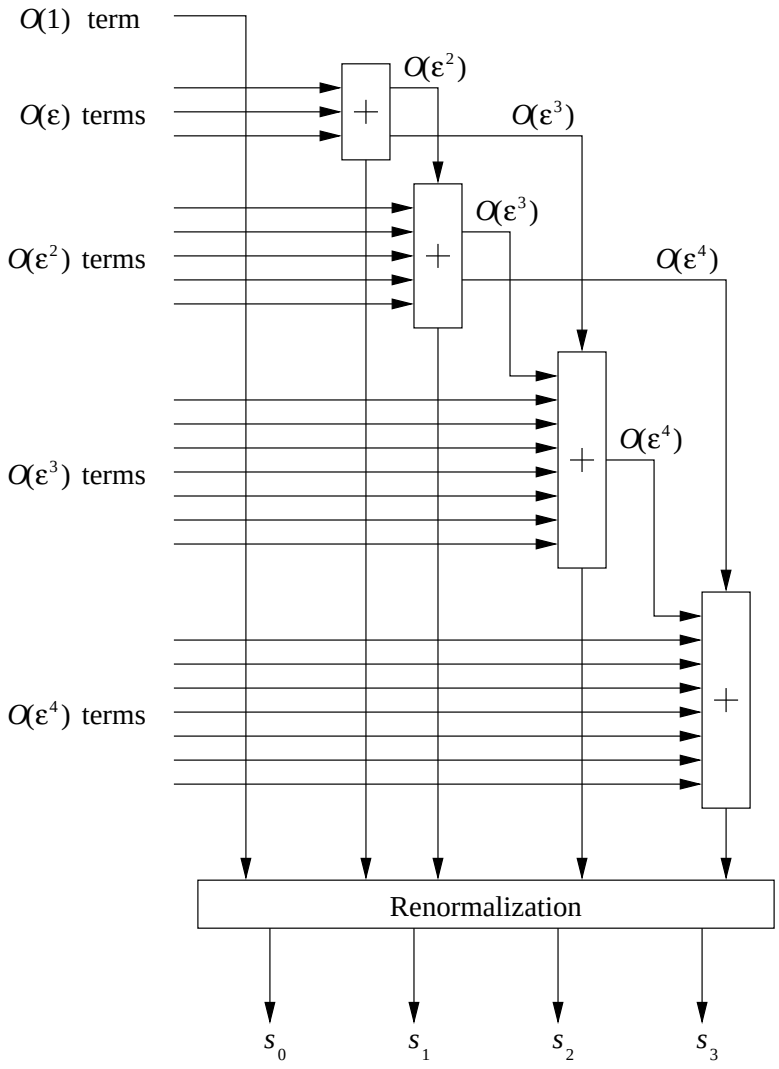

図 9: Quad-Double × Quad-Double の積算部分

このダイアグラムには 4 つの加算ボックスが入っている。一番上の加算ボックスは Three-Sum で，QD 加算と同じである。次の 3 つの加算ボックスはそれぞれ Six-Three-Sum(6 つの倍精度値の和を求め，先頭の 3 つの倍精度コンポーネントを出力)，Nine-Two-Sum(9 つの倍精度値の和を求め，先頭の 2 つの倍精度コンポーネントを出力)，Nine-One-Sum(9 つの倍精度値の和を倍精度計算で普通に求める) である。

Six-Three-Sum 演算は 6 つの倍精度値の和を計算し，精度を保っている先頭の 3 つの倍精度コンポーネントだけを求める（即ち，相対誤差を $O(\varepsilon^3)$ にする)。ここでは，6 つの入力値を倍精度 3 つ分のグループに 2 分割し，それぞれのグループ単位で Three-Sum 演算を行っている。その後，QD 加算と同様の方法でその和の加算を行う。詳細は図 10 を参照のこと。

Nine-Two-Sum は 9 つの倍精度コンポーネントを DD 精度で行う。これは，4 つの DD 数のペアと 1 つの倍精度数に分け，最終結果が DD 精度の出力結果になるまで二つの DD 数の和を求めていくことで実現される。DD 加算 (図中の一番大きいボックス）は Bailey のアルゴリズム [1] を使用する。詳細は図 11 を参照のこと。

高速化のために数ビットを犠牲にしても良いというのであれば，$O(\varepsilon^4)$ の項を計算する必要がなくなる。これによって，最初の 212 ビット分に対してのみ，足し込み計算中の桁上がりだけの影響で済む。

この場合，$O(\varepsilon^3)$ の項は通常の倍精度演算で計算を行うことができ，乗算の大幅な高速化ができる。

QD 数の 2 乗の計算は，対称性のおかげで項数を減らすことができるため，大幅な高速化が可能となる。

3.5 除算

除算は長い数に対する手計算の除算アルゴリズムで行われる。$a = (a_0, a_1, a_2, a_3)$ と $b = (b_0, b_1, b_2, b_3)$ をそれぞれ QD 数とする。最初に近似商 $q_0 = a_0/b_0$ を求め，次にその剰余 $r = a - q_0 \times b$ を求めて，次の修正項 $q_1 = r_0/b_0$ を求める。このプロセスを繰り返し，q_0, q_1, q_2, q_3, q_4 の 5 項を得る（末尾の数ビットが不要ということであれば 4 項で良い)。

それぞれの段階ではフルの QD 乗算と QD 減算が必要となることを忘れてはならない。なぜなら，q_3 と q_4 の計算で桁落ちが起きる可能性があるからである。この 4 項 (あるいは 5 項) 展開を再正規化して，QD 数の商を得ることになる。

4 代数演算

4.1 N 乗の計算

N 乗の計算は，a が QD 数である時に a^n の値を算出する。これは，David Bailey[1] 同様，単純に 2 乗を繰り返していけばよい。

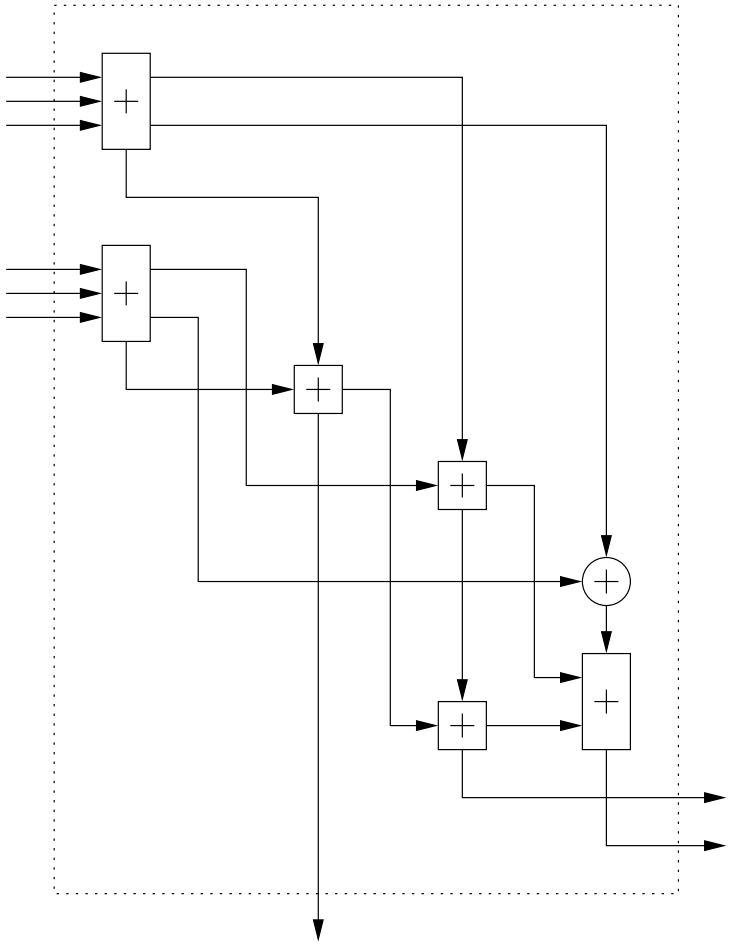

図 10: Six-Three-Sum

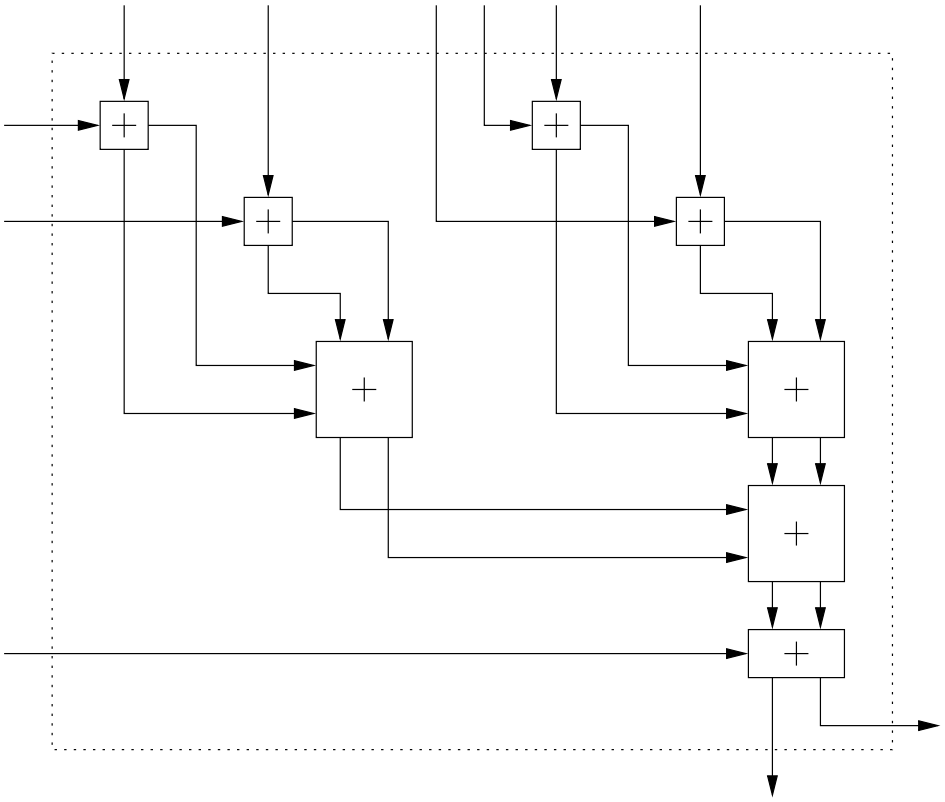

図 11: Nine-Two-Sum

4.2 平方根

平方根は与えられた QD 数 a に対して $\sqrt{a}$ を求める計算である。解が $\pm a^{-1/2}$ となる下記の方程式に対し

$$f(x) = \frac{1}{x^2} - a$$

下記の Newton 反復を用いて値を得る。

$$x_{i+1} = x_i + \frac{x_i(1 - ax_i^2)}{2}.$$

この反復は QD 数の除算を必要としない（1/2 を乗じる計算はコンポーネントごとに行えばよい)。

Newton 反復は局所的に 2 次収束するので，倍精度の初期値 $x_0 = \sqrt{a_0}$ を与えると大体 2 回の反復で済む。(実際には 3 回実行するように作ってある)。$x = a^{-1/2}$ が求められた後で，$\sqrt{a} = ax$ を得るための乗算を行う。

4.3 N 乗根

N 乗根は，与えられた QD 数 a と整数 n に対して $\sqrt[n]{a}$ を求める計算である。これも下記の方程式に対し

$$f(x) = \frac{1}{x^n} - a$$

Newton 反復を行って $a^{-1/n}$ を得る。

$$x_{i+1} = x_i + \frac{x_i(1 - ax_i^n)}{n}.$$

2 回の反復で十分であるところを 3 回反復するように作ってある。$x = a^{-1/n}$ を得た後で，逆数 $a^{1/n} = 1/x$ を求める。

5 初等関数の計算

5.1 指数関数

古典的な Taylor-Maclaurin 級数を使って e^x の計算を行っている。級数計算の前に，引数のリダクション処理を行って

$$e^{kr+m\log 2} = 2^m(e^r)^k,$$

としている。ここで，整数 m は，x に最も近い $m\log 2$ になるように決められる。こうすることで，$|kr| \le \frac{1}{2}\log 2 \approx 0.34657$ となるようにすることができる。$k = 256$ と指定すると，$|r| \le \frac{1}{512}\log 2 \approx 0.001354$ となる。ここで e^r の値を Taylor 級数を使って求める。この引数リダクションは級数の収束性を高めるためののもので，最大 18 項程度で収束するようになる。

5.2 対数関数

対数関数の Taylor 級数は，指数関数のそれに比べて収束が緩やかなので，関数 $f(x) = e^x - a$ のゼロ点を計算するための Newton 反復を利用する。反復式は下記のようになる。

$$x_{i+1} = x_i + ae^{-x_i} - 1,$$

この反復を 3 回行う。

5.3 三角関数

サイン (sine) 関数とコサイン (cosine) 関数は，引数のリダクション処理を行ってから，Taylor 級数を使って計算を行う。$\sin x$ と $\cos x$ の値を求めるには，まず引数 x に対して 2π の剰余計算を行って $|x| \le \pi$ となるようにする。$\sin(y + k\pi/2)$ と $\cos(y + k\pi/2)$ は，全ての整数 k に対して $\pm\sin y$ か $\pm\cos y$ となるので，$\pi/2$ の剰余計算を行って $|y| \le \pi/4$ となるようにし，しかるのちに $\sin y$ と $\cos y$ を計算する。

最終的には $y = z + m(\pi/1024)$ となる。ここで整数 m は $|z| \le \pi/2048 \approx 0.001534$ を満足するような値である。$|y| \le \pi/4$ であるから，$|m| \le 256$ と考えてよい。$\sin(m\pi/1024)$ と $\cos(m\pi/1024)$ をあらかじめ計算して記録した数表を使うと，

$$\sin(z + m\pi/1024) = \sin z \cos(m\pi/1024) + \cos z \sin(m\pi/1024)$$

と計算でき，$\cos(z + m\pi/1024)$ に対しても同様の関係式が成立する。このような引数リダクションを使うことで，サイン関数の収束率が劇的に向上し，高々10 項の和の計算で済むようになる。

コサインとサイン両方の値が必要な場合は，下記の式を使うことでサインの値からコサインの値を導出することができる。

$$\cos x = \sqrt{1 - \sin^2 x}.$$

$\sin(m\pi/1024)$ と $\cos(m\pi/1024)$ の値はあらかじめ MPFUN [2] のような任意精度演算パッケージを用いて計算する。計算式は下記の通り。

$$\sin\left(\frac{\theta}{2}\right) = \frac{1}{2}\sqrt{2 - 2\cos\theta}$$

$$\cos\left(\frac{\theta}{2}\right) = \frac{1}{2}\sqrt{2 + 2\cos\theta}$$

$\cos\pi = -1$ から出発すると，上記の式を使って再帰的に $\sin(m\pi/1024)$ と $\cos(m\pi/1024)$ の値を得ることができる。

5.4 逆三角関数

$\arcsin a$[5] のような逆三角関数は，$f(x) = \sin x - a$ に対する Newton 反復を用いて値を求めている。

[5]訳注) 原文は arctan。

5.5 双曲線関数

双曲線関数は次の式を使って計算する。

$$\sinh x = \frac{e^x - e^{-x}}{2} \qquad \cosh x = \frac{e^x + e^{-x}}{2}$$

x が小さい (例えば $|x| \leq 0.01$) 場合，sinh に対する上式の計算は不安定になるので，Taylor 級数を用いて計算を行う。

6 その他のルーチン

6.1 入出力

QD 数 x の 2 進数から 10 進数への変換は，$1 \leq |x10^{-k}| < 10$ を満足する整数 k を決め，10 を乗じて一桁ずつ抽出していくことで得られる。誤差の蓄積を最小限にとどめるため，あらかじめ 10 のべき乗を計算した数表を作っておく。この数表は 10 進表現から 2 進表現への変換にも使用される。

6.2 比較

QD 数は一演算ごとに全体が再正規化されるため，2 つの QD 数が等しいかどうかの判定もコンポーネントごとに行えば良い。国語辞典における単語順同様に，最初に最大桁コンポーネントに対して大きさの比較を行う。ゼロとの比較は最大コンポーネントだけで良い [6]。

6.3 乱数の生成

QD 表現の乱数は，$[0, 1)$ 区間における一様分布に従うものが生成される。これは最初の 212 ビットをランダムに選ぶことで得られる。31 ビットの乱数生成をサポートしているシステムでは 31 ビットごとに生成を行うので，$\lceil 212/31 \rceil = 7$ 回繰り返して 212 ビット乱数を得る。

7 C++プログラミング

QD ライブラリは ANSI C++で記述されており，演算子や関数のオーバーロードやユーザ定義の構造体をフル活用している。コンパイルに当たっては ANSI 標準をサポートする C++をコンパイラを使用すること。テンプレートをフルサポートしていないコンパイラではテストプログラムがまともに動作しない可能性がある。実装やビルド方法の詳細については `README` や `INSTALL` を参照されたい。

DD 数を扱うライブラリも全て C++で記述しており，QD ライブラリの一部に含まれている。倍精度，DD 精度，QD 精度の 3 種類の混合精度計算を完全にサポートしている。このライブラリを使用するには，`qd.h` ヘッダファイルをインクルードし，`libqd.a` ライブラリファイルをリンクすること

[6] 訳注) ということは非正規化数 (denomal, unnormal) は存在しない？

が必要不可欠である。QD 変数は qd_real 型として定義されており，DD 変数は dd_real 型として定義されている。

C++のサンプルプログラムを下記に示す。

```
#include <iostream>
#include <qd/qd_real.h>

using std::cout;
using std::endl;

int main() {
  unsigned int oldcw;
  fpu_fix_start(&oldcw);      // 本文の x86 マシンに関する記述を参照

  qd_real a = "3.14159265358979323846264338327950288419716939937510582";
  dd_real b = "2.249775724709369995957";
  qd_real r;

  r = sqrt(a * b + 1.0);
  cout << " sqrt(a * b + 1) = " << r << endl;

  fpu_fix_end(&oldcw);        // 本文の x86 マシンに関する記述を参照
  return 0;
}
```

文字列は QD 数 (または DD 数) として解釈するように指定しなければならない。これがない場合は倍精度の値になる。例えば a = 0.1 とすると，これは QD 精度の 0.1 とはならず，倍精度の 0.1 となってしまう。そうしたくなければ，a = "0.1"と書くこと。

$\pi, \pi/2, \pi/4, e, \log 2$ という定数の類は，qd_real::_pi, qd_real::_pi2, qd_real::_pi4, qd_real::_e, qd_real::_log2 として提供してある。この定数値は任意精度パッケージ (MPFUN++ [5]) を使って計算したもので，最後の 1 ビットまで正確である。[7]

Intel x86 プロセッサ使用時の注意　この QD ライブラリのアルゴリズムは，IEEE 倍精度浮動小数点演算が使えることを仮定している。Intel x86 プロセッサは拡張倍精度 (80 ビット) 浮動小数点レジスタを持っているため，FPU の制御ワードに round-to-double フラグを立てておく必要がある。fpu_fix_start 関数はこのフラグを FPU の制御ワードに立てることができる。元の制御ワードに戻したい時には fpu_fix_end 関数を使用する。

8　性能評価

QD 数演算の性能を下記の環境で計測した結果を表 8 に示す。

[7]訳注) Correctly rounded という意味であろう。

Operation	Pentium II 400MHz Linux 2.2.16	UltraSparc 333 MHz SunOS 5.7	PowerPC 750 266 MHz Linux 2.2.15	Power3 200 MHz AIX 3.4
Quad-double				
add	0.583	0.580	0.868	0.710
accurate add	1.280	2.464	2.468	1.551
mul	1.965	1.153	1.744	1.131
sloppy mul	1.016	0.860	1.177	0.875
div	5.267	6.440	8.210	6.699
sloppy div	4.080	4.163	6.200	4.979
sqrt	23.646	15.003	21.415	16.174
MPFUN				
add	5.729	5.362	—	4.651
mul	7.624	7.630	—	5.837
div	10.102	10.164	—	9.180

表 1: QD アルゴリズムの性能結果。値は全てマイクロ秒である。比較のため，MPFUN [2] の性能も載せてある。PowerPC 環境では Fortran 90 コンパイラが提供されていないため，MPFUN の評価値は載せていない。

- Intel Pentium II, 400 MHz, Linux 2.2.16, g++ 2.95.2 コンパイラ, コンパイル時オプションは `-O3 -funroll-loops -finline-functions -mcpu=i686 -march=i686`。
- Sun UltraSparc 333 MHz, SunOS 5.7, Sun CC 5.0 コンパイラ, コンパイル時オプションは `-xO5 -native`。
- PowerPC 750 (Apple G3), 266 MHz, Linux 2.2.15, g++ 2.95.2 コンパイラ, コンパイル時オプションは `-O3 -funroll-loops -finline-functions`。
- IBM RS/6000 Power3, 200 MHz, AIX 3.4, IBM xlC コンパイラ, コンパイル時オプションは `-O3 -qarch=pwr3 -qtune=pwr3 -qstrict`。

注: 幾つかの理由により，GNU C++ コンパイラ (`g++`) では乗算に関してとんでもなく性能が悪くなる。SUN の環境では標準 CC コンパイラを使った結果より 15 倍も低速になる。

多くのルーチンは C で実行すると目に見えて高速になる。おそらく，C++の演算子オーバーロード機構のオーバーヘッドにおける，大量の QD 数のコピーが関係しているのではないかと思われる。この現象が起きるのは，演算子オーバーロードがこの処理を考慮していないからであろう。例えば下記のコードは

```
c = a + b;
```

C++コンパイラでは次のように置き換えられる。

```
qd_real temp;
temp = operator+(a, b);      // 加算
operator=(c, temp);          // 和を c にコピー
```

これが C になると，下記のように書けるので，コピー処理は不要になる。

```
c_qd_add(a, b, c);           // (a+b) を c に格納する
```

ここで，加算ルーチンは演算結果を直接置く場所を指定されているものとしている。この問題はインライン化を行うことで若干緩和できるが，完全に除去できるものではない。この種のコピー処理を防ぐ手立てはあることはあるが [12]，オーバーヘッドそのものは残る上に，32Byte しかデータを消費しない QD 数に対しては実用的なものとは言えない [8]。

9 今後の課題

現状では，基本ルーチンすべてに対して正確性を完全には保証できておらず，再正規化過程が正常に動作しているということで良しとしている。これは，入力値が 51 ビットもオーバーラップしておらず，3 つのコンポーネントが 1 ビットもオーバーラップしていなければ，再正規化は正確に動作することを Priest が証明しているからである。もしそのようなオーバーラップが本文書で述べたアルゴリズムの中で起こっている可能性があれば，正確性の保証が別に必要となる。

改良すべき点があるとすれば，剰余演算の中で，与えられた QD 数 a, b に対して $a - \mathrm{round}(a/b) \times b$ を求める部分である。現状では，単純に，除算→丸め→乗算→減算という手順を行っているだけである。これにより，a が b よりずっと大きな値になる場合は桁落ちが発生することになる。このルーチンは指数関数，対数関数，三角関数の計算で引数のリダクションに使用されていることから，修正が必要である。

この修正のためには，QD 精度以上の計算桁数をつぎ込むのが自然である。QD 加算と乗算のアルゴリズムをより大きい計算桁数に適応させることは可能であるが，コンポーネント数が増えることから，漸近的に高速となる S. Boldo と J. Shewchuk らが提唱する高速アルゴリズムを使用することをお勧めする（アルゴリズム 14)。

このように桁数を増やすことは可能であるが，指数部は変化しないため，倍精度と同じ範囲の実数しか扱えないことになる。従って，高々2000 ビット (39 コンポーネント) 程度が使用できる最大の計算桁数となる。先頭のコンポーネントがオーバーフローギリギリで，最後尾のコンポーネントもアンダーフローギリギリである時にのみ，この計算桁数限界が必要となってくる。

10 謝辞

基本演算に関する建設的な議論をして頂いた Jonathan Shewchuk, Sylvie Boldo, James Demmel に感謝する。特に加算アルゴリズムの精度改良に関しては S. Boldo と J. Shewchuk にお世話になった。その他の諸問題に関しては J.Demmel に指摘して頂いた。

[8] より大きいデータ，例えばもっと桁数の多い演算ではデータのコピー処理に時間を要するため，この種のテクニックも役立つかもしれない。

A Quad-Double 加算の誤差限界 (補題 12)

補題 **12** 図 6 で示した QD 加算アルゴリズムにおける再正規化する前の 5 項展開式の誤差は $\varepsilon_{\mathrm{qd}}M$ 以下となる。ここで $M = \max(|a|, |b|)$。

Proof. 補題 10 と補題 11 を図中の Two-Sum と Three-Sum に個別に適用することでこの補題の証明ができる。今，$e_0, e_1, e_2, e_3, t_1, t_2, t_3, x_0, x_1, x_2, x_3, x_4, u, v, w, z, f_1, f_2, f_3$ を図 12 に使われている値とする。

この 5 項展開値 $(x_0, x_1, x_2, x_3, x_4)$ が $\varepsilon_{\mathrm{qd}}M$ 以下で抑えられることを示す必要がある。ここで $M = \max(|a_0|, |b_0|)$ である。誤差が Three-Sum 7 と Three-Sum 8 だけで発生することを考えると，低次のオーダー項 f_1, f_2, f_3 は切り捨てられる。

従って，$|f_1| + |f_2| + |f_3| \leq \varepsilon_{\mathrm{qd}}M$ であることを証明すればよい。

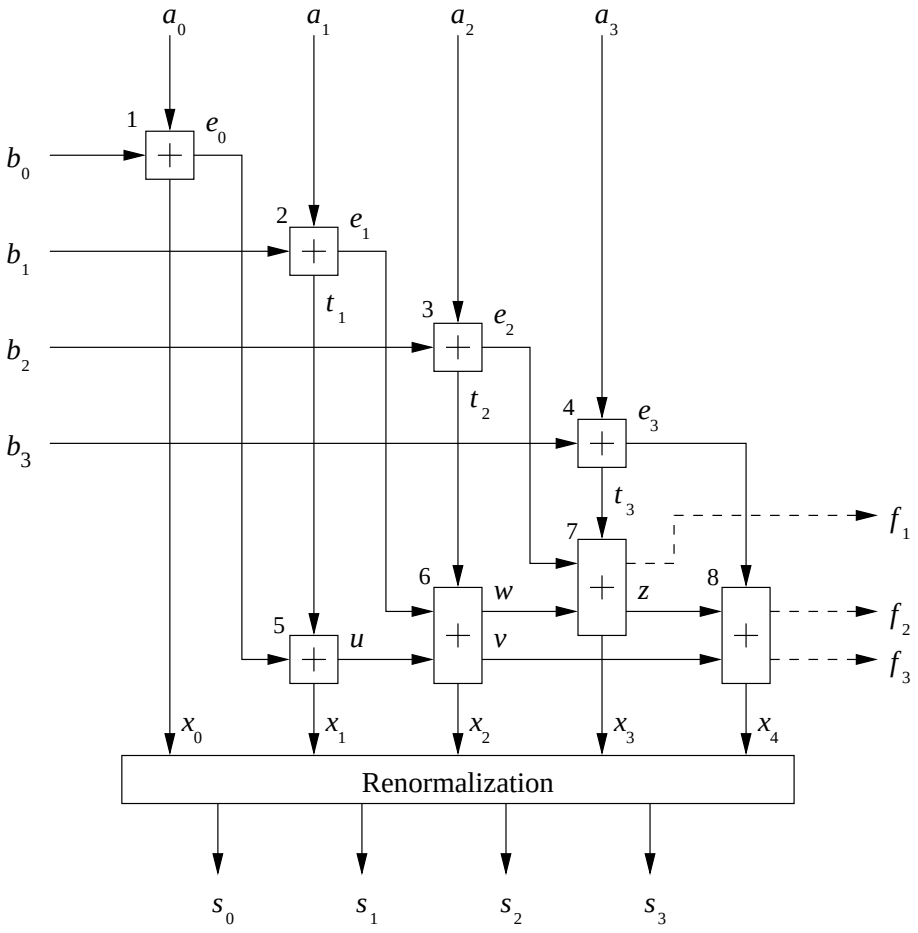

図 12: Quad-Double + Quad-Double

まず，入力値が正規化されていることから，$|a_1| \leq \varepsilon M$, $|a_2| \leq \varepsilon^2 M$，かつ，$|a_3| \leq \varepsilon^3 M$ であるこ

とを認識しておく。同様の不等式は b に対しても成立する。

補題 10 を Two-Sum 1, 2, 3, 4 にそれぞれ適用すると，下記の不等式を得る。

$$\begin{array}{rclrcl} |x_0| & \leq & 2M & |e_0| & \leq & 2\varepsilon M \\ |t_1| & \leq & 2\varepsilon M & |e_1| & \leq & 2\varepsilon^2 M \\ |t_2| & \leq & 2\varepsilon^2 M & |e_2| & \leq & 2\varepsilon^3 M \\ |t_3| & \leq & 2\varepsilon^3 M & |e_3| & \leq & 2\varepsilon^4 M. \end{array}$$

ここで，補題 10 を Two-Sum 5 に適用すると，$|x_1| \leq 4\varepsilon M$ and $|u| \leq 4\varepsilon^2 M$ を得る。さすれば，補題 11 を Three-Sum 6 に適用して下記の不等式を得る。

$$\begin{aligned} |x_2| &\leq 16\varepsilon^2 M \\ |w| &\leq 32\varepsilon^3 M \\ |v| &\leq 32\varepsilon^4 M. \end{aligned}$$

補題 11 を Three-Sum 7 に適用して

$$\begin{aligned} |x_3| &\leq 128\varepsilon^3 M \\ |z| &\leq 256\varepsilon^4 M \\ |f_1| &\leq 256\varepsilon^5 M \end{aligned}$$

を得る。

結局，補題 11 をもう一度 Three-Sum 8 に適用して下記の式を得る。

$$\begin{aligned} |x_4| &\leq 1024\varepsilon^4 M \\ |f_2| &\leq 2048\varepsilon^5 M \\ |f_3| &\leq 2048\varepsilon^6 M. \end{aligned}$$

従って，主張通り

$$|f_1| + |f_2| + |f_3| \leq 256\varepsilon^5 M + 2048\varepsilon^5 M + 2048\varepsilon^6 M \leq 2305\varepsilon^5 M \leq \varepsilon_{\mathrm{qd}} M$$

を得る。 □

参考文献

[1] David H. Bailey. A fortran-90 double-double library. Available at `http://www.nersc.gov/~dhbailey/mpdist/mpdist.html`.

[2] David H. Bailey. A fortran-90 based multiprecision system. *ACM Transactions on Mathematical Software*, 21(4):379–387, 1995. Software available at `http://www.nersc.gov/~dhbailey/mpdist/mpdist.html`.

[3] R. Brent. A Fortran multiple precision arithmetic package. *ACM Trans. Math. Soft.*, 4:57–70, 1978.

[4] K. Briggs. Doubledouble floating point arithmetic. `http://www-epidem.plantsci.cam.ac.uk/~kbriggs/doubledouble.html`, 1998.

[5] Siddhardtha Chatterjee. MPFUN++: A multiple precision floating point computation package in C++, 1998. Available at `http://www.cs.unc.edu/Research/HARPOON/mpfun++/`.

[6] T.J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, 18:224–242, 1971.

[7] GMP. `http://gmplib.org/`.

[8] Yozo Hida, Xiaoye S. Li, and David H. Bailey. Quad-double arithmetic: Algorithms, implementation, and application. Technical Report LBNL-46996, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, October 2000. Available at `http://www.nersc.gov/~dhbailey/mpdist/mpdist.html`.

[9] Donald E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison Wesley, Reading, Massachusetts, 1981.

[10] Douglas M. Priest. *On Properties of Floating Point Arithmetics: Numerical Stability and the Cost of Accurate Computations.* PhD thesis, University of California, Berkeley, November 1992. Available by anonymous FTP at `ftp.icsi.berkeley.edu/pub/theory/priest-thesis.ps.Z`.

[11] Jonathan R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18(3):305–363, 1997.

[12] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, Reading Massachusetts, 1997.