オーバー・アンダーフローを抑えた高精度かつ高速な2ノルム計算手法

原山 赳幸[1]　工藤 周平[3]　椋木 大地[3]　今村 俊幸[3]　高橋 大介[2]

概要：科学技術計算などで幅広く利用されているユークリッドノルム計算はオーバーフロー・アンダーフローや丸め誤差の蓄積等の影響で，IEEE 754 に準拠した倍精度を用いた素直な実装では誤差が非常に大きい．そこでオーバーフロー・アンダーフローを抑えるため，入力値の絶対値を三つの領域に分割しスケーリングを行う Blue による手法がある．この手法を改良し二つの領域に分割することで条件分岐を減らし高速化が期待できる二分割法を提案する．二分割法は Blue による手法と比較してオーバーフロー・アンダーフローしやすいが，入力によってスケーリング値を動的に設定することでこの特性を緩和する．また，丸め誤差の蓄積に対して Double-Double 演算を利用することに加え，Lange および Rump による Double-Double 演算の高速化を実装した．性能評価ではオーバーフロー・アンダーフローせずに処理できる入力範囲とベクトルサイズ，計算結果の相対誤差および性能の三つの観点で比較する．

1. はじめに

浮動小数点数の精度には限界があり，**表 1** に示す IEEE 754 における単精度では十進約 7 桁，倍精度では十進約 16 桁であり，現在多くのプロセッサがこの単精度と倍精度をサポートしている．倍精度における表現可能な最大値（DBL_MAX）は約 1.798×10^{308}，最小値（DBL_MIN）は約 2.225×10^{-308} である．よって，1.342×10^{154} や 1.492×10^{-154} の自乗はオーバーフローもしくはアンダーフローが発生し，計算結果が Inf や非正規化数などになってしまい精度が低下したり計算結果が利用できなくなってしまう．また，浮動小数点演算では丸め誤差の影響により倍精度でも精度が不足し，計算結果が厳密な値と比較して大きく異なる場合がある [1]．さらに数値解析分野などでは計算時間が膨大な場合があり，詳細な解析を行う際に問題となるため，利用される数値計算ライブラリは可能な限り計算機の性能を有効活用し高速な実装をする必要がある．

特にユークリッドノルムはさまざまな科学技術計算に広く利用されている．ユークリッドノルムが必要になる場合の例として，有限要素法や体積積分法のような数値解析分野の他，CG 法やグラムシュミットの直交化法などの線形計算に利用される．式 (1) で示されるベクトル $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ のユークリッドノルムは値域や精度を考慮しなければ極めて単純な計算である．

$$|\boldsymbol{x}| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \quad (1)$$

しかし，高精度かつ広い値域の入力に対応させるにはアルゴリズムが複雑化し高速化も困難になる．そこで本稿では，Blue による手法を基に値域をほぼ変化させずに高速化が期待される二分割法を提案する．Blue による手法および二分割法はオーバーフロー・アンダーフローを抑えてユークリッドノルムを計算するためのアルゴリズムであり，丸め誤差の蓄積や情報落ちに対しては有効ではないため精度拡張方法として Double-Double 演算（倍々精度演算）を利用する．加えて Lange および Rump による Double-Double 演算の高速化アルゴリズムを利用する．本稿においては IEEE 754 に準拠した倍精度について述べ，倍々精度演算を含む全ての実装で倍精度演算を利用する．

本稿は次の構成となっている．第 2 章で関連研究について述べ，第 3 章でオーバーフロー・アンダーフローの発生を抑えかつ高速に計算するための Blue による手法と提案手法である二分割法について述べる．第 4 章で丸め誤差や情報落ちを抑えるための精度拡張方法として利用する Double-Double 演算の概要と Lange および Rump による高速化手法について述べる．第 5 章で性能評価の方法と結果および考察について述べ，第 6 章でまとめと今後の課題について述べる．

2. 関連研究

前述のように，倍精度における絶対値の最大値は約

1 筑波大学大学院システム情報工学研究科
2 筑波大学計算科学研究センター
3 理化学研究所計算科学研究センター

ⓒ 2020 Information Processing Society of Japan

表 1 IEE 754 で定義されている浮動小数点数

	指数部ビット数	仮数部ビット数	表現可能な指数の最小値	表現可能な指数の最大値
binary32（単精度）	8	23	−126	127
binary64（倍精度）	11	52	−1022	1023
binary128（四倍精度）	15	112	−16382	16383

1.798×10^{308}，最小値は約 2.225×10^{-308} であるため，1.342×10^{154} や 1.492×10^{-154} の自乗はオーバーフローもしくはアンダーフローしてしまう．そのため，自乗する値に定数を掛けるスケーリングによって指数部の値をあらかじめゼロに近づける必要がある．スケーリングには Blue による手法 [2][3] や Kahan による手法 [2] などがある．Kahan による手法はベクトル $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ に対して x_i までの入力値における絶対値の最大値によってスケーリング値を変更し続ける手法であり，分岐が存在するだけでなく演算速度の遅い除算を頻繁に実行するため性能面に問題がある．Blue による手法は次節で説明する．

また，Intel MKL[4] および OpenBLAS[5] では 80 ビット浮動小数点数を利用する x87 命令を用いてユークリッドノルムを実装している．倍精度においてオーバーフロー・アンダーフローが発生する値の自乗であっても，80 ビット浮動小数点数ではオーバーフロー・アンダーフローせずに全て表現可能となる．これにより，オーバーフロー・アンダーフローを抑える Blue による手法や Kahan による手法などを利用せずにユークリッドノルムを計算可能である．倍精度演算におけるノルム計算では Bytes/Flop 比が大きくメモリ律速になるため，x87 命令と AVX-512 命令のどちらを利用しても大きく性能は変わらない可能性があることから，より精度が高い x87 命令が用いられている．

精度に着目すると内部演算に Double-Double 演算を利用する XBLAS[6] の他，GMP（GNU Multi Precision Library）[7]，MPFR[8]，Bailey による QD ライブラリ [9][10] などから浮動小数点演算を任意に指定することができる MPLAPACK[11] などがある．

3. Blue による手法と二分割法

3.1 Blue による手法のアルゴリズム

Blue による手法は入力値の絶対値の大きさに応じて三つの領域で場合分けを行い，それぞれ別々にスケーリングなどを行う手法である．Algorithm 1 に Blue による手法，Algorithm 2 に素直なノルム計算手法の擬似プログラムを示す．Algorithm 1 の内部で用いられている定数を**表 2** に示す．絶対値を 2^{300} 以上と 2^{300} 未満 2^{-300} 以上，2^{-300} 未満の三つの領域に分割し，それぞれ 2^{-600}，2^{0}，2^{600} によってスケーリングする．これにより，2^{300} 以上の入力はスケーリング後には 2^{424} 未満 2^{-300} 以上になり自乗してもオーバーフロー・アンダーフローしない．残りの二つの領域も同様に自乗してもオーバーフロー・アンダーフローしない．この手法は乗算が一つ増える程度で素直なノルム計算手法と類似しているが，分割のための分岐を必要とするためベクトル化などの性能向上手法が適用しづらいことが欠点である．

Algorithm 1 Blue による手法

Require: Vector x, size n
Ensure: The l_2 norm of x

1: $a_{\text{big}} = a_{\text{sml}} = a_{\text{med}} = 0$;
2: overFlowLimit = DBL_MAX $* s_{\text{h}}$;
3: **for** $i \leftarrow 0, n-1$ **do**
4:     **if** $|x_i| \geq \gamma$ **then**
5:         $a_{\text{big}} += (x_i * s_{\text{h}})^2$;
6:     **else if** $|x| \geq$ low **then**
7:         $a_{\text{med}} += x_i^2$;
8:     **else**
9:         $a_{\text{sml}} += (x_i * s_{\text{l}})^2$;
10:     **end if**
11: **end for**
12: **if** $a_{\text{sml}} == 0$ **then**
13:     **if** $\sqrt{a_{\text{big}}} \geq$ overFlowLimit **then**
14:         **return** (∞);
15:     **else if** $a_{\text{med}} == 0$ **then**
16:         **return** $(\sqrt{a_{\text{big}}}/s_{\text{h}})$;
17:     **else**
18:         $y_{\text{min}} = \min(\sqrt{a_{\text{med}}}, \sqrt{a_{\text{big}}}/s_{\text{h}})$;
19:         $\text{res} = \max(\sqrt{a_{\text{med}}}, \sqrt{a_{\text{big}}}/s_{\text{h}})$;
20:     **end if**
21: **else if** $a_{\text{big}} == 0$ **then**
22:     **if** $a_{\text{med}} == 0$ **then**
23:         **return** $(\sqrt{a_{\text{sml}}}/s_{\text{l}})$;
24:     **else**
25:         $y_{\text{min}} = \min(\sqrt{a_{\text{med}}}, \sqrt{a_{\text{sml}}}/s_{\text{l}})$;
26:         $\text{res} = \max(\sqrt{a_{\text{med}}}, \sqrt{a_{\text{sml}}}/s_{\text{l}})$;
27:     **end if**
28: **else**
29:     **return** $(\sqrt{a_{\text{med}}})$;
30: **end if**
31: **if** $y_{\text{min}} \leq \sqrt{\epsilon} * \text{res}$ **then**
32:     **return** $(\text{res} * \sqrt{1 + (y_{\text{min}}/\text{res})^2})$;
33: **else**
34:     **return** (res);
35: **end if**

Algorithm 2 素直なノルム計算手法

Require: Vector x, size n
Ensure: The l_2 norm of x

1: $sum = 0$;
2: **for** $i \leftarrow 0, n-1$ **do**
3:     $sum += x_i^2$;
4: **end for**
5: **return** $(\sqrt{sum})$;

表 2　Blue による手法と二分割法における定数

	γ	low	s_{h}	s_{l}
Blue	2^{300}	2^{-300}	2^{-600}	2^{600}
二分割法	1.0	—	2^{-498}	2^{498}
二分割法（動的スケーリングあり）	1.0	—	2^{-511}	$2^{496} \sim 2^{512}$

3.2　二分割法の概要と問題点

二分割法は Blue による手法を単純化したアルゴリズムであり，三つの領域に分割する代わりに入力値 x_i の絶対値に対して 1.0 以上と 1.0 未満の二つの領域に分割する．n は入力ベクトルのベクトルサイズである．二分割法は Blue による手法と比較して条件分岐が少ないため高速化が期待されることが最大の利点である．

しかし，二分割法は絶対値を二つの領域に分割しスケーリングを行うため，扱いきれない領域が存在する．これにより，一部の極めて狭い領域で Blue による手法では計算可能であるが二分割法では計算不能になることがある．例えば，二分割法の 1.0 未満の入力に対するスケーリング値は 2^{498} であるため，2^{-1009} 以下の値を入力すると自乗の計算結果が 2^{-1022} となりアンダーフローが発生する．また，$2^0 - 2^0 \times \epsilon$（$\epsilon$ は倍精度における計算機イプシロン）の入力に対してスケーリングを行うと $2^{996} \times (2^0 - \epsilon)$ となるため，同じような入力が 2^{28} 回入力された場合にオーバーフローしてしまう．この特性は次節に示す動的スケーリングによって緩和することができる．

3.3　動的なスケーリング値の変更

前述のように，二分割法では入力値が過大・過小のとき，計算できない場合がある．これは，浮動小数点数の自乗が元の値域に対して指数的に見て 2 倍よりも大きいことが一因である．また，ユークリッドノルムの計算には自乗和が必要であり，「和」によって部分和が増大していく分も考慮に入れる必要がある．つまり，n と計算可能な入力値の範囲，スケーリング値には関係性がある．そこで，入力値の長さ n に応じて最も計算可能な入力値の範囲が大きくなるスケーリング値を設定する動的スケーリング手法を提案する．**表 3** に二分割法の 1.0 未満の入力値に対するスケーリング値を変更した場合の入力可能なベクトルサイズと入力可能な範囲を示す．1.0 未満の入力に対するスケーリング値 (s_{l}) は式 (2) の r を用いて $s_{\mathrm{l}} = 2^{512-r}$ となる．

$$r \geq \lceil (\log_2 n)/2 \rceil. \tag{2}$$

n がスケーリング値に対する入力可能なベクトルサイズである 4^r 以下であれば，オーバーフロー・アンダーフローせずに処理できるが，スケーリング値が小さくなると入力可能な範囲が狭くなるため，可能な限りスケーリング値を大きくすべきである．そのため，r は $(\log_2 n)/2$ を満たす値かつ，取り得る値の中で最小の値となる．$r \geq \lceil (\log_2 n)/2 \rceil$ になる根拠は n が入力可能なベクトルサイズ以下になる必要があることに加え，1.0 未満の入力において入力される最大値は $1.0 - 1.0 \times \epsilon$ であり，2^0 に限りなく近い値である．2^0 に対して 2 の累乗のスケーリング値を掛けて自乗すると必ず 2 の偶数乗になるため，倍精度で表現できる最大値 $2^{1024} - 2^{1024} \times \epsilon$ に対して 2 の偶数乗回の加算が可能であるためである．Algorithm 3 は動的スケーリングありの二分割法の擬似プログラムであり，内部で利用されている定数を表 2 に示す．

1.0 以上の入力に対しては入力可能なベクトルサイズによってスケーリング値が変動するのではなく，入力値の範囲によって入力可能なベクトルサイズが変動するため，スケーリング値を動的に変更せず 2^{-511} で固定とし入力範囲を制限することでオーバーフロー・アンダーフローを防ぐことができる．今後，二分割法と表記した場合は動的スケーリングありの二分割法を指し，スケーリング値が固定の二分割法は動的スケーリングなしの二分割法と表記する．

3.4　入力可能な範囲とベクトルサイズ

ノルム計算においては自乗を計算する際に最もオーバーフロー・アンダーフローしやすい．これに対して各手法が掛けるスケーリング値と絶対値の分割領域が異なるため，入力可能な範囲とベクトルサイズが異なる．各手法に対して最もオーバーフロー・アンダーフローが発生しやすい入力に対して，入力可能なベクトルサイズと入力可能な範囲について述べる．まず，各手法について入力可能な範囲とベクトルサイズを**表 4** に示す．素直なノルム計算手法においては 2^{-510} もしくは 2^{511} が連続で入力された場合に 2^0 回しか演算できない．

動的スケーリングの有無に関わらず，二分割法ではベクトルサイズによって入力可能な範囲が異なる．入力可能な範囲のみが入力された場合の動的スケーリングなしの二分割法におけるスケーリング値は 2^{498} である．この場合の最悪な入力値は $2^0 - \epsilon$ である．$2^0 - \epsilon$ に対してスケーリング値 2^{498} を掛けて自乗すると $2^{996} \times (2^0 - \epsilon)$ となるため，DBL_MAX（約 2^{1024}）を超えずに $2^{996} \times (2^0 - \epsilon)$ を 2^{28} 回足し合わせることができる．よって 2^{28} 回入力可能である．二分割法における最悪な入力値も $2^0 - \epsilon$ であり設定されるスケーリング値の最小値は 2^{496} であるため，$2^0 - \epsilon$ が連続で入力された場合に 2^{32} 回入力可能である．これにより二分割法の方が動的スケーリングなしの二分割法と比較して入力可能なベクトルサイズが大きくなり，より幅広い入力に対応可能である．

ⓒ 2020 Information Processing Society of Japan

表 3　1.0 未満の入力に対する二分割法のスケーリング値，入力可能なベクトルサイズ，範囲の関係

s_l	最大値へのスケーリング	最小値へのスケーリング	入力可能なベクトルサイズ	入力可能な範囲
2^{512}	$((2^0-\epsilon)\times 2^{512})^2 \approx 2^{1024}$	$(\text{DBL_MIN}\times 2^{512})^2 \approx 2^{-1020}$	2^0	DBL_MIN 以上
2^{511}	$((2^0-\epsilon)\times 2^{511})^2 \approx 2^{1022}$	$(\text{DBL_MIN}\times 2^{511})^2 \approx 2^{-1022}$	2^2	DBL_MIN 以上
2^{510}	$((2^0-\epsilon)\times 2^{510})^2 \approx 2^{1020}$	$(2^{-1021}\times 2^{510})^2 = 2^{-1022}$	2^4	$2^{-1021}\times(2^0-\epsilon)$ 以上
2^{509}	$((2^0-\epsilon)\times 2^{509})^2 \approx 2^{1018}$	$(2^{-1020}\times 2^{509})^2 = 2^{-1022}$	2^6	$2^{-1020}\times(2^0-\epsilon)$ 以上
2^{508}	$((2^0-\epsilon)\times 2^{508})^2 \approx 2^{1016}$	$(2^{-1019}\times 2^{508})^2 = 2^{-1022}$	2^8	$2^{-1019}\times(2^0-\epsilon)$ 以上
2^{497}	$((2^0-\epsilon)\times 2^{497})^2 \approx 2^{994}$	$(2^{-1008}\times 2^{497})^2 = 2^{-1022}$	$2^{30}\approx 1.1\times 10^9$	$2^{-1008}\times(2^0-\epsilon)$ 以上
2^{496}	$((2^0-\epsilon)\times 2^{496})^2 \approx 2^{992}$	$(2^{-1007}\times 2^{496})^2 = 2^{-1022}$	$2^{32}\approx 4.3\times 10^9$	$2^{-1007}\times(2^0-\epsilon)$ 以上

これに対して Blue による手法の最悪な入力値は DBL_MIN であり，DBL_MIN が連続で入力された場合に 2^{600} でスケーリングを行うため 2^{176} 回入力可能である．よって，入力可能なベクトルサイズにおいては動的なスケーリング値の変更を行った二分割法でも Blue による手法と比較して劣るが，一般的な科学技術計算などにおいてユークリッドノルムが利用される場合に二分割法で扱いきれないような入力はほぼされない．

Algorithm 3 動的スケーリングありの二分割法

```
Require: Vector x, size n
Ensure: The l_2 norm of x
 1: a_big = a_sml = 0;
 2: overFlowLimit = DBL_MAX * s_h;
 3: r = ⌈(log_2 n)/2⌉;
 4: s_l = 2^(512−r);
 5: for i ← 0, n − 1 do
 6:     if |x_i| ≥ γ then
 7:         a_big += (x_i * s_h)^2;
 8:     else
 9:         a_sml += (x_i * s_l)^2;
10:     end if
11: end for
12: if a_sml == 0 then
13:     return (√a_big / s_h);
14: else if a_big == 0 then
15:     return (√a_sml / s_l);
16: else
17:     if √a_big ≥ overFlowLimit then
18:         return (∞);
19:     end if
20:     y_min = min(√a_sml / s_l, √a_big / s_h);
21:     res = max(√a_sml / s_l, √a_big / s_h);
22:     if y_min ≤ √ε * res then
23:         return (res * √(1 + (y_min/res)^2));
24:     else
25:         return (res);
26:     end if
27: end if
```

4. 倍々精度演算

4.1 倍々精度演算の概要

丸め誤差の蓄積を抑え，より正確な計算を実現するため浮動小数点演算の精度拡張が必要である．今回は低コストで精度拡張を実現可能な Double-Double 演算（倍々精度演算）を用いる．Double-Double 型（DD 型）は倍精度変数を二つ用いて倍々精度の演算を可能にする手法である．しかし，図 1 に示すように IEEE 754 に定義されている 4 倍精度（binary128）と比べると指数部は 4 ビット，仮数部は 8 ビット少ないため正確な 4 倍精度ではないが，ソフトウェアエミュレーションの binary128 よりも高速に動作する．指数部のビット数は倍精度と同じであるため，DD 型の表現可能な範囲は倍精度と同じである．

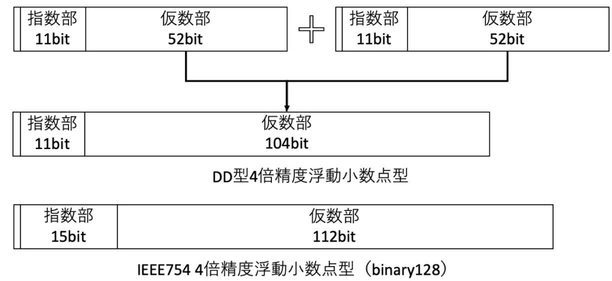

図 1　DD 型と binary128 のフォーマットの違い

倍々精度演算は Knuth[12] と Dekker[13] による丸め誤差を考慮した浮動小数点演算をもとに Bailey が開発した QD ライブラリによる実装が知られている．まず，Knuth による加算を Algorithm 4 に示す．これは，倍精度浮動小数点数 a と b の加算結果を s とし，加算時に発生した誤差を e とすることで誤差のない倍精度加算を $a+b=s+e$ と表現する．次に Dekker による加算を Algorithm 5 に示す．Knuth による加算との違いは，この加算は $|a| \geq |b|$ が成り立つときにのみ使用でき，Knuth による加算より演算回数が少なく高速に計算できる．次に Dekker による丸め誤差を考慮した倍精度乗算を Algorithm 7 に示す．倍精度演算 $a \times b$ の丸め誤差を考慮しない結果を p とし，発生した誤差を e とする．誤差のない倍精度乗算は $a \times b = p + e$ と表現できる．Algorithm 6 の Split では，倍精度浮動小数点数 a を上位ビット h と下位ビット l に分割する．分割した二つの浮動小数点数 h と l は $a = h + l$ を満たす．プロセッサに Fused Multiply-Add（FMA）命令が搭載されている場合，Algorithm 8 に示すような乗算が利用できる [14]．$a \times b + c$ を計算する場合に倍精度演算では 2 回の丸めが

ⓒ 2020 Information Processing Society of Japan

表 4　各手法に対する最悪な入力値と入力可能なベクトルサイズ

	最悪な入力値	入力可能なベクトルサイズ
素直なノルム計算手法	$2^{-511} \times (2^0 - \epsilon)$	2^0
二分割法 $sl = 2^{498}$（動的スケーリングなし）	$2^0 - \epsilon$	2^{28}
二分割法（動的スケーリングあり）	$2^0 - \epsilon$	2^{32}
Blue による手法	DBL_MIN	2^{176}

発生するが，FMA 命令では 1 回の丸めで計算可能であるため倍精度演算よりも丸め誤差の少ない結果が得られる．FMA 命令を用いることで Algorithm 7 に示す乗算よりも高速に計算できる．実装においては FMA 命令を利用する TwoProd-FMA を用いる．Algorithm 4 と Algorithm 5 を用いることで，Algorithm 9 に示す倍々精度加算が計算できる [9][10]．倍々精度加算 QuadAdd$(a_H, a_L, b_H, b_L, c_H, c_L)$ は $a = a_H + a_L$，$b = b_H + b_L$ および $c = c_H + c_L$ に対して $c = a + b$ を計算する．QuadAdd は計算結果の $c = c_H + c_L$ が $c_L \leq 0.5\mathrm{ulp}(c_H)$ を満たすように正規化する必要があり，正規化には Algorithm 5 の Quick-TwoSum を用いる．倍々精度加算を利用することで倍精度による加算に比較して丸め誤差の少ない結果を得ることができる．倍々精度乗算アルゴリズム [9][10] についても Bailey の QD ライブラリの実装が知られているが，本稿では利用しない．

また，Blue による手法および二分割法は 2 の累乗をスケーリング値として乗算を実行するためスケーリングと自乗の計算を単純化できる．2 の累乗による乗算は倍精度演算でも誤差が発生しないため，TwoProd-FMA を利用せずに倍精度演算で実行可能である．これにより，自乗の計算においても倍々精度乗算ではなく TwoProd-FMA で計算できる．そこで，倍々精度演算を用いて各手法を実装した場合の演算数について比較すると，Blue による手法および二分割法は素直なノルム計算手法と比べて，スケーリングのための倍精度乗算が 1 回増加するのみであり，ほとんど演算量の増加が発生しない．Kahan による手法に倍々精度を適用するとループの内側で倍々精度除算が必要となるため，実装が複雑化し高速化が望めない．よって，Blue による手法および二分割法は倍々精度演算との組み合わせによる有効性が高い．

Algorithm 4 Knuth による加算 [12]
```
1: function [s, e] = TwoSum(a, b)
2:     s = a + b;
3:     v = s − a;
4:     e = (a − (s − v)) + (b − v);
5: end function
```

Algorithm 5 Dekker による加算 [13]
```
1: function [s, e] = Quick-TwoSum(a, b)
2:     s = a + b;
3:     e = b − (s − a);
4: end function
```

Algorithm 6 Split[13]
```
1: function [h, l] = Split(a)
2:     t = 134217729.0 ∗ a;
3:     h = t − (t − a);
4:     l = a − h;
5: end function
```

Algorithm 7 Dekker による乗算 [13]
```
1: function [p, e] = TwoProd(a, b)
2:     p = a ∗ b;
3:     [a_H, a_L] = Split(a);
4:     [b_H, b_L] = Split(b);
5:     e = ((a_H ∗ b_H − p) + a_H ∗ b_L + a_L ∗ b_H) + a_L ∗ b_L;
6: end function
```

Algorithm 8 積和演算を用いた乗算 [14]
```
1: function [p, e] = TwoProd-FMA(a, b)
2:     p = a ∗ b;
3:     e = a ∗ b − p;                ▷ FMA
4: end function
```

Algorithm 9 倍々精度加算 [9]
```
1: function [c_H, c_L] = QuadAdd(a_H, a_L, b_H, b_L)
2:     [sh, eh] = TwoSum(a_H, b_H);
3:     [sl, el] = TwoSum(a_L, b_L);
4:     eh = eh + sl;
5:     [sh, eh] = Quick-TwoSum(sh, eh);
6:     eh = eh + el;
7:     [c_H, c_L] = Quick-TwoSum(sh, eh);
8: end function
```

Algorithm 10 CPairSum による倍々精度加算 [15]
```
1: function [c_H, c_L] = CPairSum(a_H, a_L, b_H, b_L)
2:     c_H = a_H + b_H;
3:     [t, s] = TwoSum(a_H, b_H);
4:     t = (t − c_H) + s;
5:     c_L = t + (a_L + b_L);
6: end function
```

表 5 測定環境

CPU	Intel Xeon Gold 6126（2.6GHz）× 2 基
メインメモリ	192GiB（DDR4）
OS	CentOS 7.7（x86-64）, kernel 3.10.0
コンパイラ	Intel Compiler 19.0.5.281

4.2 倍々精度演算の高速化

倍々精度加算において，Lange および Rump による CPairSum[15][16] を用いることで Double-Double 演算の高速化が可能である．CPairSum の疑似プログラムを Algorithm 10 に示す．CPairSum は QuadAdd と比較して計算結果が $c_L \leq 0.5\mathrm{ulp}(c_H)$ を満たすという仮定を省略している．これにより，QuadAdd において 2 回実行される Quick-TwoSum を CPairSum では実行しない．したがって，CPairSum を用いた倍々精度加算は QuadAdd を用いた場合と比較して，わずかながら精度が低いが少ない演算回数で計算可能である．具体的には CPairSum は通常の倍精度加算によって計算した $a+b$ の値と丸め誤差を考慮した倍精度加算 TwoSum によって計算した $a+b$ の値の差を利用し，倍々精度加算を行う．また，QuadAdd では TwoSum を用いて下位ビットを計算するが，CPairSum においては倍精度演算を用いる．これらにより，通常の倍々精度加算は 20 回の演算が必要であるが，CPairSum を利用することによって 11 回の演算で倍々精度加算を計算できる．Lange および Rump による倍精度乗算アルゴリズムである CPairProd も知られているが倍々精度乗算と同様に本稿では利用しない．

5. 性能と相対誤差の評価

5.1 評価方法

倍精度演算を用いた素直なノルム計算手法，Blue による手法，二分割法と倍々精度演算を用いた同様の手法について，相対誤差，性能の二つの観点で評価する．測定環境を**表 5** に示す．

まず，演算結果のノルムに対する各手法の相対誤差の評価方法について述べる．今回は入力ベクトルのベクトルサイズ $n = 10, 100, 1000, 10000, 100000$ に対して $[0, 1]$ の一様乱数を入力し，各手法によって計算された値と MPFR を用いて仮数部 256 ビットで計算した値との相対誤差を比較する．また，素直なノルム計算手法，Blue による手法および二分割法に対して $[2^{-1014}, 2^{1014}]$ の乱数を入力した場合の相対誤差を比較し，二分割法がオーバーフロー・アンダーフローを防ぐことができているか評価する．どの程度の誤差が発生しているか強調するために MPFR を用いて計算された値と各手法によって計算された値の差を計算機イプシロンの半分の値で正規化する．

最後に性能の評価方法について述べる．性能は Blue による手法および二分割法について，QuadAdd を用いた場合と CPairSum を用いた場合を比較する．これにより，オーバーフロー・アンダーフローを防ぐ処理がどの程度性能に影響しているのかに加え，QuadAdd を用いた場合と CPairSum を用いた場合でどの程度性能が向上するかを比較することができる．測定時のベクトルサイズは 100000 とした．

5.2 評価結果

5.2.1 相対誤差

相対誤差の評価結果について述べる．倍精度演算を用いた素直なノルム計算手法，Blue による手法および二分割法の $[2^{-1014}, 2^{1014}]$ における相対誤差を**表 6** に示す．$[0, 1]$ の一様乱数に対する倍精度演算を用いた各手法の相対誤差を**表 7** に示す．同様に QuadAdd を用いた各手法の相対誤差を**表 8** に示す．最後に CPairSum を用いた各手法の相対誤差を**表 9** に示す．ここではアルゴリズム本来の相対誤差を評価するためにコンパイラによる自動ベクトル化をオフにしてコンパイルした結果同士を比較する．

まず，倍精度演算を用いた $[2^{-1014}, 2^{1014}]$ の入力に対する各手法の相対誤差から素直なノルム計算手法においては 2^{512} 以上または 2^{-511} 以下の値が入力された場合，もしくは自乗した値を足し合わせる際に DBL_MAX を超えてしまった場合にオーバーフロー・アンダーフローが発生するため相対誤差は Inf となる．Blue による手法と二分割法は $[2^{-1014}, 2^{1014}]$ の入力に対してベクトルサイズ $n = 100000$ まではスケーリングにより自乗和を計算する際にオーバーフロー・アンダーフローが発生しない．

次に倍精度演算を用いた $[0, 1]$ の入力に対する各手法の相対誤差について述べる．$[0, 1]$ の入力に対しては素直なノルム計算手法でもオーバーフロー・アンダーフローが発生しないため計算可能である．Blue による手法と二分割法は共に同程度の相対誤差と分散になっている．これは入力値がどちらの手法においても一つの領域となり，両者の計算手順が 2 の累乗によるスケーリングを除いて同一になってしまうためである．また，Intel Compiler（19.0.5.281）におけるコンパイルオプションである-O3，-xSKYLAKE-AVX512 を指定すると精度が向上する．これは Blue による手法および二分割法において 1.0 以上と 1.0 未満の入力に対するアキュムレータの数が変化するためである．総和を計算する場合にアキュムレータの数を変化させると丸め誤差の蓄積を緩和させることができる [17]．

次に QuadAdd を用いた素直なノルム計算手法，Blue による手法および二分割法の相対誤差について述べる．倍々精度は図 1 で述べたように指数部の値は倍精度と同じであるため，表現できる絶対値の最大値と最小値は変化しない．したがって，素直なノルム計算手法を倍々精度演算を用いて実装してもオーバーフロー・アンダーフローが発生する．倍々精度演算を用いることによって仮数部のビット

表 6 $[2^{-1014}, 2^{1014}]$ の入力に対する倍精度演算を用いた各手法の相対誤差

n	素直なノルム計算手法		Blue による手法		二分割法	
	平均	分散	平均	分散	平均	分散
10	N/A	N/A	0.1	0.0	0.1	0.0
100	N/A	N/A	0.3	0.1	0.3	0.1
1000	N/A	N/A	0.5	0.2	0.5	0.2
10000	N/A	N/A	1.3	1.0	1.3	1.0
100000	N/A	N/A	5.1	12.6	5.1	12.6

表 7 $[0, 1]$ の入力に対する倍精度演算を用いた各手法の相対誤差

n	素直なノルム計算手法		Blue による手法		二分割法	
	平均	分散	平均	分散	平均	分散
10	0.5	0.1	0.5	0.1	0.5	0.1
100	1.0	0.6	1.0	0.6	1.0	0.6
1000	3.3	6.2	3.3	6.2	3.3	6.2
10000	10.0	56.3	10.0	56.3	10.0	56.3
100000	26.5	439.5	26.5	439.5	26.5	439.5

表 8 $[0, 1]$ の入力に対する QuadAdd を用いた各手法の相対誤差

n	素直なノルム計算手法		Blue による手法		二分割法	
	平均	分散	平均	分散	平均	分散
10	0.4	0.1	0.4	0.1	0.4	0.1
100	0.4	0.1	0.4	0.1	0.4	0.1
1000	0.5	0.1	0.5	0.1	0.5	0.1
10000	0.3	0.0	0.3	0.0	0.3	0.0
100000	0.4	0.1	0.4	0.1	0.4	0.1

表 9 $[0, 1]$ の入力に対する CPairSum を用いた各手法の相対誤差

n	素直なノルム計算手法		Blue による手法		二分割法	
	平均	分散	平均	分散	平均	分散
10	0.4	0.1	0.4	0.1	0.4	0.1
100	0.4	0.1	0.4	0.1	0.4	0.1
1000	0.5	0.1	0.5	0.1	0.5	0.1
10000	0.3	0.0	0.3	0.0	0.3	0.0
100000	0.4	0.1	0.4	0.1	0.4	0.1

数が拡張され，丸め誤差の蓄積が出力である倍精度まで伝搬されなくなり Blue による手法と二分割法の相対誤差は共に倍精度演算を用いた場合の相対誤差と比較して小さくなっている．

最後に CPairSum を用いた各手法の相対誤差について述べる．CPairSum は倍々精度加算を高速に処理するアルゴリズムであるため，精度については QuadAdd を用いた場合と同程度となる．

5.2.2 性能

各手法に対する倍精度演算を用いた場合の性能を**表 10** に，QuadAdd を用いた場合の性能を**表 11** に示す．同様に CPairSum を用いた場合の性能を**表 12** に示す．ただし，素直なノルム計算手法においては $[0, 1]$ および $[2^{-300}, 2^{300}]$ 以外の入力範囲ではオーバーフロー・アンダーフローが発生するため性能比較から除外した．まず，オーバーフロー・アンダーフローを抑える処理がどの程度性能に影響しているかについて着目する．$[0, 1]$ の入力に対して倍精度・QuadAdd・CPairSum を用いた場合の素直なノルム計算手法と二分割法を比較すると，倍精度では約 2 倍の性能向上となっているが，QuadAdd・CPairSum を用いた場合では同程度の性能となっている．これはノルム計算は倍精度においてはメモリ律速であるが倍々精度を用いると計算律速になるためである．

倍精度を用いた場合，素直なノルム計算手法，二分割法，Blue による手法の順に高速である．これはコンパイラの自動ベクトル化によって各手法がベクトル化されるためである．各手法において倍精度演算を用いた場合にはコンパイラがベクトル化可能であるが，倍々精度演算を利用すると自乗を足し合わせる処理において依存性があるためコンパイラの自動ベクトル化は利用できない．また倍精度におけるベクトル化は素直なノルム計算手法，二分割法，Blue による手法の順にベクトル化率が高くなっており，条件分岐の数がベクトル化率に大きく影響するため Blue による手法に比べて二分割法の方が高速化が容易である．

QuadAdd を用いた場合には全ての入力範囲において二分割法の方が Blue による手法に比べて高速か同程度の性能になった．$[2^{-1014}, 2^{1014}]$ のように二分割法，Blue による手法共に全てのアキュムレータに値が格納される場合は二分割法の方が高速である．その他の入力範囲においては二分割法と Blue による手法共に同程度の性能となった．

CPairSum を用いた場合の性能は QuadAdd を用いた場合と比較して最大で約 3.8 倍高速になった．しかし，CPairSum を用いた場合に $[2^{-300}, 2^{299}]$ の入力では Blue による手法の方が二分割法と比較して高速となった．この原因として，予測不能な条件分岐の数と演算回数の関係が考えられる．この範囲の入力では，Blue による手法は一つの領域として処理されるが二分割法では 1.0 を境に二つの領域として処理される．よって二分割法は CPU の分岐予測が失敗しやすい条件となっている．そのため QuadAdd を用いた場合，CPairSum と比べて倍々精度加算に必要な演算量が多いため，この違いが表われにくいが，逆に CPairSum を用いた場合には分岐予測失敗のペナルティが結果に大きく反映されやすく，Blue による手法の方が二分割法よりも高速になったと考えられる．

6. まとめ

本稿ではオーバーフロー・アンダーフローを抑えてユークリッドノルムを計算する手法として二分割法を提案し，倍々

ⓒ 2020 Information Processing Society of Japan

表 10　各入力に対する倍精度を用いた場合の性能

入力範囲	素直なノルム計算手法 [GB/sec]	二分割法 [GB/sec]	Blue による手法 [GB/sec]
$[0, 1]$	49.7	24.8	18.6
$[2^{-1014}, 2^{1014}]$	N/A	13.5	10.6
$[2^{300}, 2^{1014}]$	N/A	13.5	10.6
$[2^{-300}, 2^{299}]$	29.8	13.5	10.6
$[2^{-301}, 2^{-1014}]$	N/A	13.5	10.6

表 11　各入力に対する QuadAdd を用いた場合の性能

入力範囲	素直なノルム計算手法 [GB/sec]	二分割法 [GB/sec]	Blue による手法 [GB/sec]
$[0, 1]$	0.7	0.7	0.7
$[2^{-1014}, 2^{1014}]$	N/A	1.0	0.9
$[2^{300}, 2^{1014}]$	N/A	0.6	0.7
$[2^{-300}, 2^{299}]$	0.7	1.0	0.7
$[2^{-301}, 2^{-1014}]$	N/A	0.7	0.7

表 12　各入力に対する CPairSum を用いた場合の性能

入力範囲	素直なノルム計算手法 [GB/sec]	二分割法 [GB/sec]	Blue による手法 [GB/sec]
$[0, 1]$	2.3	2.3	2.3
$[2^{-1014}, 2^{1014}]$	N/A	1.5	1.1
$[2^{300}, 2^{1014}]$	N/A	2.3	2.3
$[2^{-300}, 2^{299}]$	2.3	1.3	2.3
$[2^{-301}, 2^{-1014}]$	N/A	2.3	2.3

精度演算を用いて高精度化を行った．さらに，CPairSum を用いて倍々精度加算の高速化を実装し，同様に CPairSum を用いた Blue による手法および素直なノルム計算手法と入力可能な範囲とベクトルサイズ，相対誤差および性能の観点で比較した．オーバーフロー・アンダーフローのしやすさでは二分割法は Blue による手法に劣るが，同程度の精度で計算が可能であり，倍々精度を用いた場合に $[2^{-1014}, 2^{1014}]$ の入力範囲では性能が約 43% 高速である．しかし，入力範囲によっては倍々精度を用いた二分割法は Blue による手法と比較して約 26% 低速もしくは同程度の性能となった．これに対する今後の課題として二分割法のベクトル化を検討している．ベクトル化では分岐の処理が大きく変わり，分岐の回数自体が大きく性能に影響を与えるため，二分割法の条件分岐の少ない利点が強調され，倍精度演算を用いた場合の性能のように全域で二分割法の方が高速に計算できる可能性がある．

謝辞　本研究の一部は筑波大学計算科学研究センターの学際共同利用プログラム（Cygnus）の助成を受けたものです．

参考文献

[1] 椋木大地，高橋大介：GPU における 3 倍・4 倍精度浮動小数点演算の実現と性能評価，情報処理学会論文誌コンピューティングシステム (ACS)，Vol. 6, No. 1, pp. 66–77 (2013).

[2] Hanson, R. J. and Hopkins, T.: Remark on Algorithm 539: A Modern Fortran Reference Implementation for Carefully Computing the Euclidean Norm, *ACM Trans. Math. Softw.*, Vol. 44, No. 3, Article 24 (2017).

[3] Blue, J. L.: A Portable Fortran Program to Find the Euclidean Norm of a Vector, *ACM Trans. Math. Softw.*, Vol. 4, No. 1, pp. 15–23 (1978).

[4] Intel Math Kernel Library, `https://software.intel.com/content/www/us/en/develop/tools/math-kernel-library.html`.

[5] Xianyi, Z., Qian, W. and Chothia, Z.: OpenBLAS, `http://www.openblas.net`.

[6] Li, X. S., Demmel, J. W., Bailey, D. H., Henry, G., Hida, Y., Iskandar, J., Kahan, W., Kang, S. Y., Kapur, A., Martin, M. C., Thompson, B. J., Tung, T. and Yoo, D. J.: Design, Implementation and Testing of Extended and Mixed Precision BLAS, *ACM Trans. Math. Softw.*, Vol. 28, No. 2, p. 152–205 (2002).

[7] GMP: GNU Multi Precision library, `https://gmplib.org`.

[8] Hanrot, G., Lefèvre, V., Pélissier, P., Théveny, P. and Zimmermann, P.: The GNU MPFR Library, `https://www.mpfr.org/`.

[9] Bailey, D. H.: QD: A double-double and quad-double package for Fortran and C++, `https://www.davidhbailey.com/dhbsoftware/`.

[10] Hida, Y., Li, X. S. and Bailey, D. H.: Algorithms for Quad-Double Precision Floating Point Arithmetic, *Proceedings of the 15th IEEE Symposium on Computer Arithmetic*, pp. 155–162 (2001).

[11] Nakata, M.: The MPACK (MBLAS/MLAPACK); a multiple precision arithmetic version of BLAS and LAPACK, `http://mplapack.sourceforge.net/` (2010).

[12] Knuth, D. E.: *The Art of Computer Programming, Volume 2 : Seminumerical Algorithms*, Addison-Wesley, MA, 3rd edition (1997).
[13] Dekker, T. J.: A floating-point technique for extending the available precision, *Numerische Mathematik*, Vol. 18, No. 3, pp. 224–242 (1971).
[14] 永井貴博，吉田仁，黒田久泰，金田康正：SR11000 モデル J2 における 4 倍精度積和演算の高速化，情報処理学会論文誌コンピューティングシステム（ACS），Vol. 48, No. 13, pp. 214–222 (2007).
[15] Lange, M. and Rump, S. M.: Faithfully Rounded Floating-Point Computations, *ACM Trans. Math. Softw.*, Vol. 46, No. 3, Article 21 (2020).
[16] Rump, S. M.: Error Bounds for Computer Arithmetics, *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*, pp. 1–14 (2019).
[17] Blanchard, P., Higham, N. J. and Mary, T.: A class of fast and accurate summation algorithms, *SIAM J. Sci. Comput.*, Vol. 42, pp. 1541–1557 (2020).

ⓒ 2020 Information Processing Society of Japan