ハイパフォーマンス
コンピューティング 58－9
（1995．10．18）

多倍長平方根の高速計算法

高橋大介† 金田康正††

東京大学大学院理学系研究科情報科学専攻†
東京大学大型計算機センター††

本稿では，億を超える多倍長桁数の平方根を高速に計算する方法について述べる．多倍長桁数の平方根の計算は，平方根の逆数に収束するニュートン法を適用することにより，多倍長桁数の加減乗算に帰着される．n 桁の多倍長桁数の乗算はFFT（高速フーリエ変換）を用いれば $O(n\log n)$ のオーダーで求まるが，多倍長桁数の平方根を求めるにあたって，一度に多倍長桁数を求めるのではなく，計算桁数を分割することで，全体の計算量を減らすことができること，さらにフーリエ変換されたデータを再利用することで，多倍長桁数の平方根が高速に求まることを示す．その結果，1億桁の$\sqrt{2}$の計算では，およそ1.5倍程度高速化されることが分かった．

Fast Multiple-Precision Calculation of Square Root

Daisuke TAKAHASHI† Yasumasa KANADA††

Department of Information Science, Graduate School of Science, University of Tokyo†
Computer Centre, University of Tokyo ††

This paper describes for the fast multiple-precision calculation of the square root. By using Newton method to reciprocal of the square root, the calculation of the square root can be reduced to the multiple-precision addition, subtraction and multiplication. And, n digits multiple-precision multiplication can be realized the computing complexity of $O(n\log n)$ with FFT(Fast Fourier Transform). In this paper, we show that the computational complexity for the square root can be reduced by dividing the multiple-precision multiplication, and fast multiple-precision calculation of the square root is also realized by reusing the Fourier transformed intermediate data. According to the experimental results for 100 million decimal digits calculation of sqrt(2), our method is 1.5 times as fast as the conventional method.

1 はじめに

平方根は，あらゆる科学技術計算のなかで最も基本的な演算であり，また使用頻度の高い初等関数でもある．

平方根の超高精度計算のアルゴリズムに関しては，これまでに Dutka[1] は平方根の連分数近似を拡張した 2 次収束法を提案し，それを用いて$\sqrt{2}$を実際に 100 万桁計算した．また，小沢，海野 [2, 3] は Dutka[1] のアルゴリズムをより高次のものへと拡張してきた．

一方，金田 [4] は平方根の逆数に収束するニュートン法により，$\sqrt{2}$を 1600 万桁計算した．

平方根の逆数に収束するニュートン法により多倍長桁数の平方根を計算する際，n 桁の多倍長桁数の乗算は FFT（高速フーリエ変換）を用いれば $O(n\log n)$ のオーダーで求まることが知られている．

本稿では，多倍長桁数の平方根を求めるにあたって，一度に多倍長桁数を求めるのではなく，計算桁数を分割することで，全体の計算量を減らすことができること，さらにフーリエ変換されたデータを再利用することで，多倍長桁数の平方根が高速に求まることを示す．以下，2 章でニュートン法による平方根の計算について，3 章で多倍長桁数の計算アルゴリズムについて述べる．4 章で多倍長桁数の平方根計算アルゴリズム，5 章で数値実験の結果について述べる．

2 ニュートン法による平方根の計算

ニュートン法は，方程式

$$f(x)=0 \tag{1}$$

を解くための反復法で，ある解αの“十分良い”近似値 x_0から出発して

$$x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)} \tag{2}$$

により次々と $x_1, x_2, \cdots$を計算していく方法である．

これより，$\sqrt{a}$を求めるには，$\sqrt{a}$を $f(x)=x^2-a=0$ の解として，$f'(x)=2x$ より，

$$x_{n+1}=x_n-\frac{x_n^2-a}{2x_n} \tag{3}$$

となる．ここで，式 (3) を少し変形した，

$$x_{n+1}=\frac{1}{2}\left(x_n+\frac{a}{x_n}\right) \tag{4}$$

の形が，教科書で紹介されることが多い．

式 (3),(4) には x_nによる除算が含まれており，多倍長の除算は，一般的に乗算より時間がかかるので，この方法は多倍長桁数の$\sqrt{a}$を求める場合は効率が悪い．

そこで，$\sqrt{a}=a\cdot 1/\sqrt{a}$であることを利用すると，まず $1/\sqrt{a}$を $f(x)=1/x^2-a=0$ の解として求め，その結果に a を掛けることにより，$\sqrt{a}$を求める．具体的には，$f'(x)=-2x^{-3}$ より，

$$\begin{aligned} x_{n+1} &= x_n-\frac{1/x_n^2-a}{-2x_n^{-3}} \\ &= x_n+\frac{x_n^3(1/x_n^2-a)}{2} \\ &= x_n+\frac{x_n(1-ax_n^2)}{2} \end{aligned} \tag{5}$$

となる．ここで，式 (5) を少し変形した

$$x_{n+1}=\frac{x_n(3-ax_n^2)}{2} \tag{6}$$

が実際に多倍長桁数の$\sqrt{a}$を求める場合に使用されている [4].

さて，ニュートン法による$\sqrt{a}$の計算は 2 次の収束，つまり正しく求まる桁数が毎回前回の 2 倍になるので，始めは初期値で与えた 2 倍の桁数で計算を始め，毎回桁数を 2 倍にしていけばよく，始めから全桁数の多倍長計算を行う必要はない．

ニュートン法の 2 次収束性から，次に示す推量が直ちに導き出せる．

いま，$2s$ 桁の精度で解を得たいとする．x_{n+1}が $2s$ 桁目まで解αと一致するためには，x_nとしては s 桁目までαと一致するようなものが得られていればよい．すると，x_nと x_{n+1}とは始めの s 桁は一致しているから，$\Delta x_n=x_{n+1}-x_n$ は有効数字が s 桁くらいあればよい．

このような観点から式(5)を見直してみよう．x_nはs桁の数であるとする．x_{n+1}は$2s$桁の数である事を考慮し，$1-ax_n^2$の前半分のs桁は0になることから，

x_n^2: (s桁の数)×(s桁の数)⇒($2s$桁の数)

ax_n^2: ($2s$桁の数)×($2s$桁の数)⇒($2s$桁の数)

$1-ax_n^2$: $1-$($2s$桁の数)⇒(s桁くらいの数)

$\Delta x_n = x_n(1-ax_n^2)/2$:
(s桁の数)×(s桁の数)⇒(s桁の数)

$x_n+\Delta x_n$: (s桁の数)の後に(s桁の数)を
追加⇒($2s$桁の数)

というような計算を行えばよいことが分かる．つまり，x_n^2とax_n^2の計算の所だけ十分神経を使って$2s$桁まできちんと結果を出すようにしておけば，あとはみなs桁計算で済むというわけである．

これは，式(2)の$f(x_n)$に相当する所の計算さえしっかりやっておけば，あとは所望の最終桁数の半分程度の桁数の計算だけで実質的には済んでしまうということを示している[6, 7].

3 多倍長桁数の乗算アルゴリズム

ニュートン法によって，多倍長桁数の平方根を計算するためには，多倍長計算を行う必要がある．多倍長桁数どうしの加減算，比較，また多倍長桁数と単精度数の乗算，除算は自明だから省略する．問題は多倍長桁数どうしの乗算である．以下にそのアルゴリズムを説明する．

3.1 多倍長桁数の乗算

多倍長桁数の乗算のアルゴリズムは種々あるが，筆算で行うように部分積を計算する方法(筆算では九九を用いた1桁×1桁，計算機では例えば4桁×4桁）では，計算桁数をnとすれば$O(n^2)$の計算量が必要となり桁数が大きくなると膨大な計算量，計算時間が必要になる．高速のアルゴリズムとしては，積をとるべき数を上位と下位の桁に2分し再帰的なアルゴリズムを用いることによって，計算量を$O(n^{\log_2 3})$に減らすアルゴリズム[5]や，畳み込み積分をFFTを媒介にして計算することによって，計算量を$O(n\log n)$に減らすアルゴリズム[5]がある．

約1000桁以上の乗算では，FFTを用いた乗算アルゴリズムが実際的には最も速いことから，今回はFFTによる乗算アルゴリズムを使用する場合の議論を行う．

n桁の数どうしを掛けるのに，筆算のような方法では$O(n^2)$の計算量が必要である．

ところが，高速フーリエ変換(Fast Fourier Transform)を用いると，多倍長の乗算が$O(n\log n)$の計算量で行えることが知られている．これは，多倍長の乗算が畳み込みと本質的には同一であることから，畳み込みはフーリエ変換を媒介にして行うことができ，この変換にFFTを用いて高速化を図るものである．以下，その方法を簡単に説明する．

まず，畳み込みが多倍長乗算と本質的に同じアルゴリズムであることを示す．説明のために，10進4桁の2整数A, Bの積を求めることを考える．

$$\begin{aligned} A &= 10^3a_0+10^2a_1+10a_2+a_3 \\ B &= 10^3b_0+10^2b_1+10b_2+b_3 \end{aligned}$$

a_i, b_iは各位の数で0～9の数をとる．AとBの積は，

$$\begin{aligned} A\cdot B = {} & 10^6a_0b_0+10^5(a_0b_1+a_1b_0) \\ & +10^4(a_0b_2+a_1b_1+a_2b_0) \\ & +10^3(a_0b_3+a_1b_2+a_2b_1+a_3b_0) \\ & +10^2(a_1b_3+a_2b_2+a_3b_1) \\ & +10(a_2b_3+a_3b_2)+a_3b_3 \end{aligned}$$

である．ここで積の上位からi位目は，

$$\sum_{m=0}^{i} a_m b_{i-m}$$

という形をしており，これは畳み込みの定義式と同じ形である．したがって多倍長の乗算を行うには，A, Bの畳み込みを計算し，その後基底を10にして正規化を行なえばよいことになる．

上記をまとめると，計算の手順としては，まず積をとる2数A, B両者のフーリエ変換を行

い，次に両者のフーリエ係数どうしの積をとり，最後に逆フーリエ変換および正規化を行うことになる．この場合，積をとる2数 A, B は10進数である必要は全くない．2進数でも，8進数でも何進数でも良いという事実は，実際の計算の上では大変に望ましいことである．

実際のプログラムでは，配列を用意し，8バイトの倍精度変数に10進で3桁ずつ格納し，配列の後半に0を代入して結果が循環畳み込みと呼ばれるものにならないようにしてFFTを行う．8バイトに10進で3桁の数値を格納するのは，メモリー効率が悪いようであるが，これは畳み込みにおいて，1億桁同士の乗算では3桁×3桁＝6桁の数値の約3300万項の和を求めるためで，中間的には15桁程度の整数値を正確に確保する必要があるからである．

倍精度浮動小数点の仮数部のビット数はIBM方式では56ビット（10進で約16.8桁），IEEE方式では52ビット（ケチ表現のため実際は53ビット＝10進で約15.9桁）であるので問題は無いが，CRAY方式では48ビット（10進で約14.4桁）であるので，1億桁同士の乗算をメモリ上で行うことは精度の点から無理であることが分かる．このような場合は，8バイトの倍精度変数に10進で2桁ずつ格納することになる．実際の計算においては，正規化を行う際に整数値からの誤差を監視しながら実行する必要がある．

FFTの計算量は $O(n\log n)$ であるので，結局多倍長乗算も $O(n\log n)$ のオーダの計算量で求まることになる．

3.2 多倍長桁数の乗算における分割

さて，ここでさらに計算量を減らすために，多倍長桁数をある程度分割して計算を行う場合について考えてみる．

n 点FFTの計算量を $c_{fft}\cdot n\log_2 n$ とし，n 点のフーリエ係数どうしの積を求める際の計算量を $c_{conv}\cdot n$ とすれば，分割をしない場合の乗算（ここでは，上位の桁だけを求めることにする）の計算量は順FFTが2回，逆FFTが1回であることから，

$$T = 3c_{fft}n\log_2 n + c_{conv}n$$

となるが，2分割をした場合は，

$$\begin{aligned} T_{div2} &= 3\cdot 2c_{fft}\frac{n}{2}\log_2\frac{n}{2} + \left(\sum_{i=1}^{2} i\right)\cdot c_{conv}\frac{n}{2} \\ &\quad + \left(\sum_{i=1}^{1} i\right)\cdot\frac{n}{2} \\ &= 3c_{fft}n(\log_2 n - 1) + \frac{3}{2}c_{conv}n + \frac{n}{2} \\ &= 3c_{fft}n\log_2 n + c_{conv}n \\ &\quad + \left(-3c_{fft} + \frac{1}{2}c_{conv} + \frac{1}{2}\right)n \end{aligned}$$

となる．これから $3c_{fft} > \frac{1}{2}c_{conv} + \frac{1}{2}$ ならば分割をしない場合に比べて計算量が減ることが分かる．

上記の場合は2分割のケースであるが，d 分割をした場合を考え，d がいくらの場合に計算量が最小になるかを考えてみる．

d 分割をした場合の計算量は，2分割の場合と同様にして，

$$\begin{aligned} T_{div} &= 3d\cdot c_{fft}\frac{n}{d}\log_2\frac{n}{d} + \left(\sum_{i=1}^{d} i\right)\cdot c_{conv}\frac{n}{d} \\ &\quad + \left(\sum_{i=1}^{d-1} i\right)\cdot\frac{n}{d} \\ &= 3c_{fft}n(\log_2 n - \log_2 d) \\ &\quad + \frac{d(d+1)}{2}c_{conv}\frac{n}{d} + \frac{(d-1)d}{2}\cdot\frac{n}{d} \\ &= 3c_{fft}n\log_2 n - 3c_{fft}n\log_2 d \\ &\quad + \frac{d+1}{2}c_{conv}n + \frac{n(d-1)}{2} \end{aligned}$$

となる．

ここで最適な d，すなわち極値となる d を求める．

$$\begin{aligned} T'_{div} &= -3c_{fft}n\cdot\frac{\log_2 e}{d} + \frac{1}{2}c_{conv}n + \frac{n}{2} \\ &\equiv 0 \end{aligned}$$

であるから，

$$d = \frac{6c_{fft}\log_2 e}{c_{conv}+1}$$

となる d が最適であることが分かる．

実数FFTの計算量は $(5/2)n\log_2 n$ であるので，$c_{fft} = 2.5$ であり，フーリエ係数どうしの積の計算においては，実数FFTの性質を考

慮すると，$c_{conv}=3$ となる．したがって，最適な d は

$$\begin{aligned} d &= \frac{6c_{fft}\log_2 e}{c_{conv}+1} \\ &\approx 5.41 \end{aligned}$$

となる．

また，同様にして分割をする場合の自乗計算の場合の最適な d を求めると

$$d = \frac{4c_{fft}\log_2 e}{c_{conv}+1}$$

となるので，$c_{conv}=2.5$ を考慮すると，

$$d \approx 4.13$$

となる．

次に上位桁だけではなく，$2n$ 桁まで正確に求める自乗計算を行った場合の最適な d を求めることを考える．今までと同様の考え方で，d 分割をした場合の計算量は，

$$\begin{aligned} T_{div} &= (3d-1)c_{fft}\frac{n}{d}\log_2\frac{n}{d} + d^2 c_{conv}\frac{n}{d} \\ &\quad +(d-1)^2\frac{n}{d} \\ &\approx 3c_{fft}n\log_2 n - 3c_{fft}n\log_2 d \\ &\quad +\frac{d}{2}c_{conv}n + nd \end{aligned}$$

となる．

ここで，最適な d を求める．

$$\begin{aligned} T'_{div} &= -3c_{fft}n\cdot\frac{\log_2 e}{d} + \frac{1}{2}c_{conv}n + 1 \\ &\equiv 0 \end{aligned}$$

より，

$$d = \frac{3c_{fft}\log_2 e}{c_{conv}+1}$$

となる d が最適であることが分かる．

実数 FFT の計算量は $(5/2)n\log_2 n$ であるので，$c_{fft}=2.5$，$c_{conv}=2.5$ となる．したがって，最適な d は

$$\begin{aligned} d &= \frac{3c_{fft}\log_2 e}{c_{conv}+1} \\ &\approx 3.09 \end{aligned}$$

となる．

4 多倍長桁数の平方根計算アルゴリズム

多倍長桁数の平方根の計算には，初期値を x_0 として倍精度 (16 桁程度) で与えた後に，次の式を反復毎に桁数を倍々にして計算する．

$$x_{n+1} = x_n + \frac{x_n(1-ax_n^2)}{2} \qquad (7)$$

式 (7) において，x_n と x_{n+1} とは前半分の桁は一致しているので，その都度フーリエ変換の値を保存しておけば，各反復においては，精度の増加分だけのフーリエ変換を行えばよいことが分かる．さらに，a が多倍長桁数の場合，a のフーリエ変換については，一度求めた値は全体が再利用できる．

フーリエ変換の値の再利用は，2 分割や 4 分割を行った場合は，最後の 1〜2 回の反復の場合（$N/4$，$N/2$ 桁の計算に相当）において行うことになるので，実際の計算では有効である．このように，多倍長の乗算において，分割を行うことにより，フーリエ変換の再利用ができ，全体の計算量が減ることが分かる．

さらに，式 (7) において，x_n^2 を求めるときに x_n のフーリエ変換を求めるが，このフーリエ変換の値は x_n と $(1-ax_n^2)$ を掛けるときに再利用できる．

また，式 (7) により $1/\sqrt{a}$ が求まった後に a を掛けて $\sqrt{a}$ を求めるが，a が多倍長桁数の場合は，$1/\sqrt{a}$ を求める際に a のフーリエ変換の値を保存しておくことにより，フーリエ変換の回数をさらに削減することができる．

5 数値実験

数値実験は，$\sqrt{2}$ を従来のアルゴリズムと今回提案するアルゴリズムを用いて計算し，実行時間を比較することにより行った．

計算機としては，スーパーコンピュータ HITAC S-3800/480(主記憶 2GB) を用いた．計算に際しては，4 プロセッサのうち，2 プロセッサを使用し，2 プロセッサの合計の CPU TIME を測定することにより行った．

計算桁数は，1 億 ($\approx 3\times 2^{25}$) 桁とし，多倍長桁数の乗算を分割せず，フーリエ変換の値の

再利用も行わない場合（従来のアルゴリズム）と，多倍長桁数の乗算を4分割し，フーリエ変換の値の再利用を行った場合（提案するアルゴリズム）について，比較を行った．結果は次のようになった．

1億桁の$\sqrt{2}$の計算による実行時間の比較

	CPU時間（秒）
従来のアルゴリズム	21.93
提案するアルゴリズム	14.56

上の表から分かるように，従来のアルゴリズムに比べて，フーリエ変換の回数が減っていることにより約50%ほど高速化されていることが分かる．

なお，提案するアルゴリズムを用いて，$\sqrt{2}$の64億桁が約39分で求まったことを付記しておく．

6 まとめ

本論文では，億を超える多倍長桁数の平方根を高速に計算する方法について述べた．多倍長桁数の平方根を求めるにあたって，一度に多倍長桁数を求めるのではなく，計算桁数を分割することで，全体の計算量を減らすことができること，さらにフーリエ変換されたデータを再利用することで，多倍長桁数の平方根が高速に求まることを示した．

謝辞：$\sqrt{2}$の64億桁計算に際しては，東京大学大型計算機センターの業務の方々にお世話になりました．ここに深く感謝いたします．

参考文献

[1] Dutka, J.: The Square Root of 2 to 1,000,000 Decimals, Math. Comp.,Vol. 25, pp. 927-930 (1971).

[2] 小沢一文，海野啓明：連分数を用いた平方根の高速発生法とその計算効率，情報処理学会アルゴリズム研究会資料，SIG-AL 5-3, pp. 17-24 (1989).

[3] 小沢一文：多倍長演算のための平方根の高速計算法，情報処理学会論文誌，Vol. 31, No. 7, pp. 953-963 (1990).

[4] 金田康正：円周率800万桁及び1/sqrt(2), sqrt(2), 1600万桁の検証・統計結果，数理解析講究録，Vol. 498, pp. 66-88 (1983).

[5] D.E.Knuth: The Art of Computer Programming, Vol. 2 : Seminumerical Algorithms, Addison-Wesley, Reading, Massachusetts (1981).

[6] R.P.Brent: Fast Multiple-Precision Evaluation of Elementary functions, J. ACM, Vol.23, No. 2, pp. 242-251 (1976).

[7] 伊理正夫：ニュートン法の実際，数理科学，Vol. 218, pp.10-16 (1981).