Vol.2017-HPC-158 No.8
2017/3/8

# 相加相乗平均による初等超越関数の計算

平山 弘[1,a)]　加藤 俊二[1,b)]

**概要**：1976 年に Breant は、初等超越関数 ($\exp x$, $\log x$, $\tan^{-1} x$, $\sin x$, $\cosh x$, etc) は、計算精度 $n$ を無限大にしたときの極限 ($n \to \infty$) において、$O(n(\log n)^2 \log(\log n))$ の演算量計算できることを示した。この計算法は、楕円積分の理論、相加相乗平均法によるものである。
この方法は、高精度の計算において効果的であることがわかるが、どの精度で効果的であるかあまり知られていない。本論文では、Brent のアルゴリズムでの初等超越関数の計算と通常の計算アルゴリズムの計算を比較し Brent アルゴリズムの有効精度を求めた。

**キーワード**：多倍長高精度計算，初等超越関数，相加相乗平均法

# Calculation of Elementary Transcendental Function by Arithmetic-Geometric Mean

Hiroshi Hirayama[1,a)]　Shunji Katoh[1,b)]

***Abstract:*** In 1976, Brent showed that the elementary transcendental functions ($\exp x$, $\log x$, $\tan^{-1} x$, $\sin x$, $\cosh x$, etc) can be calculated in $O(n(\log n)^2 \log(\log n))$ operations with relative error $O(2^{-n})$ as $n \to \infty$. This algorithms depends on the theory of elliptic integrals, using the arithmetic-geometric mean method. It's learned that this method is effective when calculating by the high precision, but It isn't known in what kind of area this way is effective. In this paper, Elementary transcendental function was calculated and its validity was checked using algorithm of this Brent.

***Keywords:*** multi-precision, elementary transcendental functions, the arithmetic-geometric mean

## 1. はじめに

これまで多くの高精度計算プログラムが開発され来たが、相加相乗平均法である高速な Brent のアルゴリズム [1] を実装したプログラムは開発されて来なかった。このアルゴリズムは極限において非常に効率的なものであるが、100 桁とか 1000 桁程度の計算では、あまり有効ではなかったためではないかと思われる。最近計算機が速くなり記憶装置も十分な大きさを持つようになったことから、Brent のアルゴリズムの有効性を調べた。

Brent の文献にもあるが、Salamin が提案した計算法に相加相乗平均を使う円周率の高速な計算法 [2] がある。この計算法は一時期円周率の計算によく使われた計算法である。その当時、それと類似した方法で対数関数の計算 [3] の研究などもある。最近は、分割法 [7][8] によって円周率の計算が行われるようになって来ている。関数の計算も将来的には、その方法で計算が行われると思われる。

本論文では、高精度の計算でしか有効でないと言われた相加相乗平均法を使って、初等超越関数を計算し、どの程度有効かを調べ、その有効範囲を決定する。

以下の計算には、計算機として Intel i7-4770K 3.5GHz を使用し、コンパイラーとしては、Microsoft Visual Studio 2013 C++を使用した。

[1] 神奈川工科大学創造工学部自動車システム開発工学科
Department of Vehicle System Engineering, Faculty of Creative Engneering, Kanagawa Institute of Technology, Shimo-Ogino 1030, Atsugi, Kanagawa, 243-0292, Japan
[a)] hirayama@kanagawa-it.ac.jp
[b)] katoh@kanagawa-it.ac.jp

## 2. FFT による高精度数の乗算

高精度数の演算で問題になるのは計算時間である。このため高速アルゴリズムは必要不可欠である。その中で、高精度数の乗算は、高速フーリエ変換 (FFT) を使うと高速に行うことができることが知られている。$n$ 桁の数値の乗算には通常 $n^2$ のオーダーの計算時間が必要であるが、FFT を使うと $n \log n$ のオーダーで計算できる。相加相乗平均法で高速に計算するためにも、FFT を利用した乗算が不可欠である。

Karatsuba による高速乗算法 [6] もある。低精度の乗算では通常の乗算法が有効で、高い精度計算では FFT を利用した乗算方法が有効であることが知られているので、その中間の精度で Karatsuba の乗算方法が有効ではないかと推定される。この方法で高精度計算プログラムを作成してみたが、自作のプログラムでは有効な範囲が見つからなかった。このため、自作プログラムでは、Karatsuba の乗算方法は、複素数の乗算には使っているが、多倍長精度の数値の乗算には使っていない。

$n$ が十分大きな数の場合すなわち高い精度の場合、FFT を使った計算法は非常に効率的である。多くの計算機で、10 進数で約 6000 桁程度を越えれば、FFT アルゴリズムを使った計算法が、通常の計算法より高速に計算できる。

FFT を使って、高精度数の積を計算するには、浮動小数点を使用する方法と整数演算を使う二つの計算方法が考えられる。有限体を使って整数演算で計算する方法と浮動小数点を使って計算する方法である。有限体を使った計算方法の場合、整数演算だけを使うので、誤差の入らない厳密な計算結果が得られる。

浮動小数点演算を使う方法は、三角関数などの近似値を扱うために厳密な計算ができない。しかし、最終結果は整数であることが分かっているので、誤差が十分に小さいならば、丸め処理によって厳密な計算が可能である。これまでの多くの計算機では、最も精度の高い整数は、32 ビットであり、浮動小数点数は、64 ビットであった。大型計算機では、128 ビットの浮動小数点数もある。このため、計算精度の高い浮動小数点数を使った計算方法が多く使われてきた。最近のマイクロプロセッサでは、64 ビット精度の整数を扱うことが出来るので有限体を使った計算法も効率的である可能性はある。

### 2.1 実数用 FFT による高精度数の乗算法

高精度整数 $x$ を $b$ 進数 $m$ 桁で次のように表現する。

$$x = \sum_{k=0}^{m-1} x_k b^k \tag{1}$$

このとき高精度整数 $x$ と $y$ の積 $z$ の $j$ 桁は

$$z_j = \sum_{k=0}^{j} x_k y_{j-k} \tag{2}$$

となる。この計算は畳み込み演算と呼ばれ高速フーリエ変換によって効率よく計算できることが知られている [4]。この場合、倍精度実数を利用した FFT を利用する計算方法がよく使われる。FFT にも多くの計算法があり、今回利用したプログラムには、実数用で基数が 2,4 および 8 の FFT プログラムである。FFT には、複素数用と実数用があるがここで扱うデータが実数であるので、実数用を利用した。実数用は、同じデータ数であるとき複素数用の約 2 倍高速であり、メモリの使用量も約半分である。基数 2 の FFT は、プログラムは短く、よく使われる計算法である。Bergland[5] って発表された基数 8 のプログラムを使えば、さらに高速に計算できるので、FFT には基数 8 のものを使用した。

実数用 FFT の計算では、計算の途中で打切り誤差が入る。この誤差は、最後の丸め処理によって厳密な値になるためには、次のような関係式を満たさなければならない。この式は Henrici[4] によって導かれたものである。$b$ 進数 $m$ 桁の数値を厳密に乗算できるには、計算精度の相対誤差を $\eta$ (マシン・イプシロン) とすると

$$\eta \leq \frac{1}{192m^2(2\log_2 m + 7)b^2} \tag{3}$$

を満たさなければならない。この式から基数 $b$ が大きいほど要求精度が高くなることがわかる。桁数 $m$ も大きくなると要求精度が高くなる。この式は、相対誤差の 2 乗以上の高次の項を省略する方法で、誤差を評価しているので、十分条件になる。

現実にはもっと緩い条件でもでも計算可能である。たとえば、$b = 10000$ 、$\eta = 2.22 \times 10^{-16}$(IEEE 方式の倍精度浮動小数点) の場合、この式では計算可能桁数は 428 桁となる。実際には 1000 万桁以上の数も計算可能である。(3) 式より精密な評価を得るために、区間演算などを試みたが、若干適用範囲が増加したが、実用的な範囲にはならなかった。

(3) 式を誤差の評価式と見たとき、誤差は、対数関数部分を省略すると、基数 $b$ の 2 乗、桁数 $m$ の約 2 乗におおよそ比例することがわかる。この場合の基数 $b$ とは、1 語の中に入る最大の整数という意味になるので、誤差が最大になるのは、各語に $b-1$ の数値を入れたとき最大の誤差になることがわかる。上の例では、$b = 10000$ であるから、各語に 9999 を入れたとき最大の誤差が生じることになる。$b = 2^n$ のときは、各語に $2^n - 1$ を入れたとき最大の誤差が生じる。計算結果は、整数であることが分かっているので、誤差が 0.5 より小さいならば、四捨五入の計算によって、厳密な計算が可能である。誤差は、最大の数値を入れて計算することによって計算が可能なので、計算できる限

界も容易にわかる。

### 2.2 FFT による数値の乗算例

ここでは、簡単な数値で FFT による乗算例をあげる。ここでは簡単数値の掛け算 $6153 \times 4753 = 29159655$ を行う。基数 $b$ を 100 とすると、$6135 = 61 \times 100 + 35$、$4753 = 47 \times 100 + 53$ と表せる。

これらをそれぞれ $[35, 61, 0, 0]$,$[53, 47, 0, 0]$ と配列で表す。配列の後ろに 0 を追加したのは、計算した結果が桁数が増加するのでその準備のためである。

これを FFT で変換すると、

$$FFT([35, 61, 0, 0]) = [96, 35 + 61i, -26, 35 - 61i]$$
$$FFT([53, 47, 0, 0]) = [100, 53 + 47i, 6, 53 - 47i]$$

2 番目と 4 番目の数値は、共役複素数となるので、計算を省略できる。このように作られた FFT のプログラムを実数 FFT(RFFT) と呼ばれ計算時間、メモリの量を節約できる。

これらの配列の要素毎掛け算すると、$[9600, -1012 + 4878i, -156, -1012 - 4878i]$ となる。これを逆変換すると

$$IFFT\ ([9600, -1012 + 4878i, -156, -1012 - 4878i]) = [1855, 4878, 2867, 0]$$

となる。FFT を行い、さらにそれの逆 FFT を行うと、元のデータのデータ数倍になる。ここでは、IFFT の計算でデータ数 4 で割っている。結果は $[1855, 4878, 2867, 0]$ となる。これから、次の結果が得られる。

$$1855 + 4878 \times 100 + 2867 \times 100^2 = 29159655$$

各桁数 $a$ が、基数 $b$ の半分以上なら、$a = a - b$ として、1 を桁上げし、そうでなければ、そのままにすると、$b$ が半分になるのと同じ効果を上げられる。これを上の例題で行う。

6135 と 4753 は 100 進数で表し、この計算を行うとそれぞれ $[35, -39, 1, 0]$,$[-47, 48, 0, 0]$ と配列で表される。

これを FFT で変換すると、

$$FFT([35, -39, 1, 0]) = [-3, 34 - 39i, 75, 34 + 39i]$$
$$FFT([-47, 48, 0, 0]) = [1, -47 + 48i, -95, -47 - 48i]$$

これらの配列の要素毎掛け算すると、$[-3, 274 + 3465i, -7125, 274 - 3465i]$ となる。これを逆変換すると

$$IFFT\ ([-3, 274 + 3465i, -7125, 274 - 3465i]) = [-1645, 3513, -1919, 48]$$

となる。これから、つぎのように上と同様の結果が得られる。

$$-1645 + 3513 \times 100 - 1919 \times 100^2 + 48 \times 100^3 = 29159655$$

### 2.3 最大誤差の計算

前節で述べたことを実際に行って、IEEE 方式の倍精度浮動小数点をもつ計算機を使って誤差を計算した。$b = 10000$ と固定して、桁数 $m$ を増やす。このときの誤差は、表 1 のようになる。

**表 1** 桁数 $m$ を増やしたときの誤差

| m | error | 増加倍率 |
|---|---|---|
| 8 | 1.24e-13 | - |
| 16 | 3.14e-13 | 2.53 |
| 32 | 1.36e-12 | 4.33 |
| 64 | 3.87e-12 | 2.85 |
| 128 | 1.09e-11 | 2.82 |
| 256 | 2.64e-11 | 2.42 |

このときの計算方法は、基数 2 の FFT である。桁数 $m$ が 2 倍になると、誤差は約 3 〜 4 倍となる。$m$ が小さいため、省略した $\log_2 m$ の影響か約 4 倍とはいえないが、上の公式がかなり成り立つことがわかる。

次に、桁数 $m = 256$ と固定して、基数 $b$ を変化させる。このとき、誤差は表 2 のようになる。このときの計算法は、基数 2 の FFT である。

**表 2** 基数 $b$ のビット幅を増やした時の誤差 ($m = 256$)

| $b$ | error | 増加倍率 |
|---|---|---|
| 8 | 1.73e-11 | - |
| 16 | 8.00e-11 | 4.62 |
| 32 | 3.27e-10 | 4.09 |
| 64 | 1.34e-09 | 4.10 |
| 128 | 5.56e-09 | 4.15 |
| 256 | 2.14e-08 | 3.85 |

この計算結果から、誤差は、基数 $b$ のかなり正確に 2 乗に比例することがわかる。この誤差が、0.5 より小さければ、実数を使った FFT による高精度の数値の乗算が厳密に行うことができる。このようにして、限界を求めると次のようになる。FFT として基数 8 の Bergland のプログラム [5] を利用した。基数 として、 2 のべき乗と 10 のべき乗を使った。その結果は表 3 に示す。$b = 10000$ 場合の計算は、使用したプログラムの限界で、求められなかった。ここで示した $b = 10000$ の結果は、Ooura によって作成したプログラム [9] による結果である。

誤差評価では、$b$ は基数としたが、正確には各桁の数値の絶対値の大きさある。通常は、各桁の数値は、0 または正であるから、基数と一致するので、上の誤差評価は同じものになる。

もし各桁の数値 $a$ が $a \geq b/2$ 以上なら、$a$ を $a - b$ とし、上位桁に繰り上げるようにするれば誤差評価の $b$ は、半分になる。このため計算可能桁数は 4 倍の桁数まで増やすことができると思われる。

表 3　基数 8 の FFT を使った場合の計算可能桁数

| 基数 $b$ | 誤差 | 計算可能桁数 (10 進換算) |
|---|---|---|
| 1000 | - | >33554432 |
| 10000 | - | 33554432 |
| 100000 | 0.281 | 81920 |
| 1000000 | 0.203 | 768 |

### 2.4 FFT を使った乗算の計算時間

高速フーリエ変換 (FFT) を使うと高精度の数値を高速に乗算を行うことができる。同じ桁の数値の掛け算を行うとき、10000 進数で約 450 桁 (10 進数で約 3600 桁) 以上のとき、FFT を使った計算法を使い、それ以下では通常の乗算法を使っている。いろいろな桁数の計算時間を表 4 に示す。計算精度を上げていくと、途中再び通常の計算法が一旦速くなることもあるが、計算時間にそれほど大きな違いにならないので、その精度の時でも、FFT を使った計算法を使用している。通常の計算法と FFT を利用した計算法の境界の桁数は、高速計算機ほど、大きくなる傾向がある。

最初にこのプログラム作成した計算機 Intel 社 (i286+287) では、乗算の限界が 10 進数で約 700 桁であった。

表 4　乗算の実行時間 (単位 msec)

| 10 進の桁数 | FFT | 通常 |
|---|---|---|
| 10,000 | 0.33 | 0.67 |
| 20,000 | 0.67 | 2.61 |
| 50,000 | 1.50 | 16.31 |
| 100,000 | 3.13 | 64.76 |
| 200,000 | 6.84 | 259.04 |
| 500,000 | 13.93 | 1617.00 |
| 1,000,000 | 29.25 | 6437.00 |

## 3. 相加相乗平均法による初等関数の計算

相加相乗の平均の計算とは、二つの数列 $a_n, b_n$ に対して、次のような計算を次々と行うことである。

$$a_{n+1} = \frac{a_n + b_n}{2}, \quad b_{n+1} = \sqrt{a_n b_n} \tag{4}$$

この計算は、2 乗収束することが知られている。この計算法を利用した関数計算が Brent によって提案されている。Brent によると、関数 $U(m), T(m)$ は C 言語で書くと表 5 のようになる。ここで、$n$ ビット精度で計算する場合、$eps1 = 2^{-n/2}$, $eps2 = 2^{-n}$ である。また $pi = \pi$ である。このとき

$$U(m) = \log(T(m)), \quad e^{U(m)} = T(m) \tag{5}$$

この式を使って指数関数 $e^x$ を計算するには、まず次の方程式を解く。

$$U(m) = x \tag{6}$$

方程式 (6) を解き解 $m_0$ を求める。上の関係式から次の式が成り立つ。

$$e^{U(m_0)} = T(m_0) \tag{7}$$

$m_0$ が求まれば、$T(m_0)$ を計算すれば、$e^x$ の値が計算できる。(6) の方程式を解くために、次のように Newton 法を使う。

$$m_{n+1} = m_n - \frac{U(m_n) - x}{U'(m_n)} \tag{8}$$

得られた $m$ を使って $T(m)$ を計算する。すなわち $e^x$ を求めることができる。逆に

$$T(m) = x \tag{9}$$

を解き、求められた $m$ を使って $U(m)$ を計算することによって $\log x$ を求めることができる。

表 5　Brent の $U(m)$ と $T(m)$ の関数

```
 1: double u( const double m ){
 2:     double a, b, c, s, eps1 ;
 3:     double pi=3.141592653589793238 ;
 4:     a = 1 ; b = sqrt(1-m) ;
 5:     while( a-b > eps1 ){
 6:          c = (a+b)/2 ;
 7:          b = sqrt(a*b) ;
 8:          a = c ;
 9:     }
10:     a = pi/(a+b) ;
11:     s = sqrt(m) ;
12:     while( 1-s > eps1 ){
13:          a = a*(1+s)/2 ;
14:          s = 2*sqrt(s)/(1+s) ;
15:     }
16:     return a*(1+s)/2 ;
17: }
18: double t( const double m ){
19:     double  s, v, w, eps2 ;
20:     v = 1 ; s = sqrt(m) ;
21:     while( 1-s > eps2 ){
22:          w = 2*s*v/(1+v*v) ;
23:          w = w/(1+sqrt(1-w*w)) ;
24:          w = (v+w)/(1-v*w) ;
25:          v = w/(1+sqrt(1+w*w)) ;
26:          s = 2*sqrt(s)/(1+s) ;
27:     }
28:     return (1+v)/(1-v) ;
29: }
```

## 4. 自動微分法による高速化

Brent[1] は Newton 法で使う微分係数を計算するために数値微分を使うことを提案している。このような数値微分法では、(8) の右辺を計算するために最低でも関数 を 2 回計算しなければならない。ところが上の式を見ればわかるように、除算や平方根の計算が含まれている。この場合、除算や平方根を含む式を計算する時間より少ない時間で微分係数を計算することができる。

次のような 2 行の単純なプログラムで x について微分することを考える。

```
p = 1+x^2  ;   q = sqrt(p)
```

p、q の x についての微分を dp、dq とすると

```
dp=2*x*dx  ;     dq=dp/(2*sqrt(p))
```

として計算される。関数 sqrt(p) は、すでに直前で計算されているものであるから、微分係数を計算するには平方根の計算は不要になり、高速に計算 [10] することができる。

実際の計算では、平方根を計算するには、平方根の逆数を計算し、その値から平方根を計算されているため、上の計算では平方根による除算も不要となり、次のように高速化ができる。

```
p=1+x^2 ; t=rsqrt(p); q=t*p
```

ここで、rsqrt(x) は平方根の逆数を与える関数である。

```
dp=2*x*dx  ;     dq=dp*t/2
```

と計算できる。高精度計算では、短い桁数の整数によるわり算は高速に行えるので、2 によるわり算は問題とならず、全体としてかなり高速化できる。

わり算を含む式の微分係数の計算は、乗算の回数は増えるものの除算が不要になるのでかなり高速化できる。指数関数 $e^x$ の微分係数は、この関数が微分しても同じ値となるため、この性質を使うことで高速化できる。対数関数 $\log x$ の微分係数は、この関数の微分が $\frac{1}{x}$ と単純な関数になることを利用して高速化できる。三角関数 ($\sin x$ 、$\cos x$) の計算は、これらの関数が同時に計算すると高速計算することができることから、高速化できる。この方法でプログラムすると $U(m)$ と $U'(m)$ が同時に計算できるプログラムを作ることができる。

今回の計算では、$U(m)$ が収束したとき、収束計算をやめるようにプログラムした。これは Newton 法では、微分係数をそれほど精度を必要としないこととこの計算法では非常に収束が早く、微分係数も十分早く収束すると判断したからである。

一般的には、どちらの数値も収束するまで、計算する必要があると思われる。

### 4.1 微分係数を使った効果

指数関数の計算例として、$e^{\sqrt{2}} = 4.1132503\cdots$ を 20000 桁と 50000 桁の精度で計算した。結果を以下に示す。計算は (6) の方程式を倍精度数を使い Newton 法で解き、高精度計算を始めるための初期値を計算した。その値を使い高精度数を使い、Newton 法で必要な精度を確保しながら計算した。Newton 法では、微分係数は計算精度の半分で十分なので、微分係数の計算は計算精度を半分にして評価した。

**表 6** AGM 法による指数関数の計算時間 (sec)

| | 20000 桁 | 50000 桁 |
|---|---|---|
| 差分法 | 1.87 秒 | 4.60 秒 |
| 自動微分法 | 1.15 秒 | 2.87 秒 |

微分係数を差分近似で計算する方法に比べ、約 38 パーセント高速になった。微分係数の計算が差分法に比べ、4 倍程度高速になったことになる。

## 5. 相加相乗平均法が有効な範囲

相加相乗平均法 (AGM 法) の有効範囲を求めて計算精度を上げて調べた。AGM 法として微分係数を利用した高速化した AGM 法を使った。比較の対象は自作の多倍長プログラム (MPPack) と比較した。ここでは、4.1 節の問題を使って計算した。

MPPack では、細かい点を除けば、$e^x$ の $x$ を $\frac{x}{2^n}$ として、引数を十分小さくし、Taylor 展開式を使って $e^{\frac{x}{2^n}}$ を計算し、その値を $n$ 回 2 乗して計算する。$n$ は計算する Taylor 展開式の項数と 2 乗計算の回数が 1:2 になりように選んでいる。

その計算結果を表 7 に示す。AGM 法は 100 万桁以下では通常の多倍長ルーチン (MPPack) より遅いことがわかった。200 万桁の計算でようやく速くなった。この多倍長ルーチンは、その当時の計算機が遅かったこともあり、10 万桁程度以下の数値想定して作成したものである。このため、さらに改良が行われると思われるが、現段階では AGM 法は 100 万桁を超えたところでは、効率的な計算法と言える。

**表 7** AGM 法と多倍長ルーチン (MPPack) の比較 (sec)

| 計算精度 | 高速化 AGM 法 | MPPack |
|---|---|---|
| 20,000 | 1.16 | 0.27 |
| 50,000 | 2.87 | 0.99 |
| 100,000 | 5.53 | 2.91 |
| 200,000 | 15.1 | 8.95 |
| 500,000 | 41.0 | 29.6 |
| 1,000,000 | 89.4 | 94.4 |
| 2,000,000 | 229. | 302. |

## 6. まとめ

相加相乗平均法 (AGM 法) による初等超越関数の計算法を解析的な微分係数の計算 (自動微分法) を導入することによって、高速化を行った。高速化された AGM 法で初等超越関数の計算を行い、その計算の有効範囲を調べた。その有効範囲は 100 万桁以上の範囲とわかった。この範囲では 1 個の関数値を計算するのに 1 ～ 2 分以上かかり、あまり実用的とは思えない範囲であった。

円周率の計算では、最近 AGM 法を使わない傾向があり、

それらに利用されている分割方法を使えば、通常の計算法がさらに高速化される可能性があり、さらに AGM 法による計算の有効範囲がさらに高精度に追いやられると思われる。

## 参考文献

[1] Brent, R. P., Fast multiple-precision evaluation of elementary functions, J. Assoc. Comput. Mach. 23(1976),242-251
[2] Salamin, E. , Computation of $\pi$ using arithmetic-geometric mean, Math. Comput. 30(1976), 13-19
[3] Sasaki T. and Kanada, Y., Practically Fast Multiple-Precision Evaluation of Log(x), J. Info. Proc. 4(1982),247-250
[4] Henrici. P., Applied and Computational Complex Analysis, Vol. 3, Chap. 13, John Wiley & Sons, New York(1986)
[5] Bergland. G. D., A Radix-Eight Fast Fourier Transform Subroutine for Real-Valued Series, IEEE Trans. A. E., AU17.2, pp.138-144
[6] Karatsuba, A. and Ofman, Y., Multiplication of mutidigit numbers on automata, Doklady Akad. Nauk SSSR, Vol.145, pp.293-294(1962)
[7] 後 保範, 金田康正, 高橋大介, 級数に基づく多数桁計算の演算量削減を実現す る分割有理数化法，情報処理学会論文誌，41(2000)1811-1819
[8] 平山 弘, 連分数の多倍長精度高速計算法, 情報処理学会論文誌、41(2000).85-91
[9] 大浦 拓哉, Ooura's Mathematical Software Packages, ttp://www.kurims.kyotou.ac.jp/ooura/index-j.html
[10] Rall,L. B., Automatic Differentiation Technique and Applications, Lecture Notes in Computer Science, Vol. 120, Springer Verlag, Berlin-Heidelberg-New York, (1981)