Vol.2018-HPC-166 No.11
2018/9/28

4倍精度および8倍精度の数学関数プログラムの開発

平山 弘[1,a)]

概要：C++言語の規格では、ガンマ関数や誤差関数等を容易に利用することができるようになっている。2個の倍精度数を利用した double-double 型 4 倍精度数や 4 個の倍精度数を利用した 8 倍精度数は整数を使ったソフトウエアで実装されたものより高速であるなどの利点はあるが、これらの関数がなければ、これらの高精度浮動小数点数の活用が難しくなると思われる。
本論文では、C++言語、C 言語や Fortran 言語の規格でも準備されているガンマ関数、誤差関数とそれに関連する関数を作成した。
これらの関数が準備されることによって、多くの分野でこれらの高精度浮動小数点数が使われると思われる。

Development of Mathmatical Function for Double-double and Quad-double Precision Arithmetic Program

Hiroshi Hirayama[1,a)]

Abstract: In C language and C ++ language standards, Gamma functions, Error functions, and the like can be easily used.
Double-double type quad precision number using two double precision numbers and Quad-double type octal precision numbers using four double precision numbers have advantages such as being faster than those implemented with integer software. Without these functions, it seems difficult to utilize these high precision floating point numbers.
In this paper, we have prepared Gamma functions, Error functions and related functions prepared for C ++ language, C language and Fortran language standards as well.
By preparing these functions, it seems that these high precision floating point numbers are used in many fields.

Keywords: Quad precision number, Octal precision number, Gamma function, Error function

1. はじめに

多くの数値計算は、計算機のハードウエアで実行できる倍精度浮動小数点数の演算で済む場合が多いが、計算速度の高速化に伴い丸め誤差が大きくなり、もう少し高い精度の計算が望まれてきている。計算の品質をあげるためにも高精度計算の要求も高まって来ている。最新の計算機の性能は、高速に動作し非常に使い易くなってきているが 4 倍精度をハードウエアで実行できる計算機がほとんどないため、これらの要求に答えられない状況になっている。

このような状況をある程度克服するため、Bailey[10][11]によって提案されている倍精度を二つ組み合わせた 4 倍精度数の高速演算プログラムを作成した。この計算方法は、単純なので非常に短く容易に四則演算等のプログラムを作成できる。この論文には、指数対数関数や三角関数などの計算法も一部紹介されている。プログラムが単純であるためか、プログラム言語で準備されている 4 倍精度より高速に計算ができる。

これらの数値の入出力するプログラムはかなり長いものになり作成が難しいものとなるが、4 倍精度を持つプログ

[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

ラム言語では、プログラム言語の 4 倍精度数を使って入出力を行うことができるので、簡単に利用できる。

4 倍精度数を持たない多くのプログラミング言語等では、4 倍精度数を使うことが少し難しいものになっている。

最新の C 言語や C++言語では、Bessel 関数やリーマンのゼータ関数等を準備することが要求されている。将来的には、このような関数を準備することになると思われる。

本文では、C 言語やほかのプログラミング言語でサポートされている関数で、Bailey の論文で計算法が紹介されていない数学関数であるガンマ関数や誤差関数等のプログラム作成法を述べる。これらの関数を準備することによって、double-double 型の 4 倍精度や quad-double 型の 8 倍精度の浮動小数点の利用範囲が大きく広がると思われる。

ここで取り上げた例題の計算時間は、Intel i7-8700K CPU 3.70GHz で実行し、測定した時間である。使用したコンパイラーは Microsoft Visual Studio 2017 C++である。

2. double-double 型 4 倍精度数の計算アルゴリズム

ここで簡単に、Bailey の double-double 型の 4 倍精度数について述べる。Bailey の double-double アルゴリズム [5] では，4 倍精度浮動小数点数 (real16) を二つの倍精度浮動小数点数を使い上位桁を m0、下位を m1 で表し、次のような構造体で表す。

```
class real16 {  double  m0, m1 ; }
```

4 精度変数 a を二つの倍精変数 a.m0(上位データ) および a.m1(下位データ) を用いて次のように表す。

$$\mathrm{a} = \mathrm{a.m0} + \mathrm{a.m0} \quad (\frac{1}{2}ulp(\mathrm{a.m0}) \geq |\mathrm{a.m1}|) \qquad (1)$$

ここで、$ulp(x)$ は x の最小ビット (unit in the last place) を意味する。このとき，a.m0 および a.m1 は通常の倍精度浮動小数点数である。このため仮数部の精度は 53bit であり、2 つの倍精度浮動小数点数を利用することで 106bit の精度で表現できる．そのため，double-double アルゴリズムは IEEE754-2008 の 4 倍精度と比較すると 8bit 分だけ精度が劣る．しかし，IEEE754-2008 の 4 倍精度はソフトウエアで作成されている場合が多いため、計算速度はハードウエアの計算をする部分が多い double-double 型 4 倍精度数が速く計算が出来るので、実用的な方法であると言える [8]。

4 倍精度加算および乗算を double-double アルゴリズムを利用して計算する方法を説明する。まず、double 型の数値 2 個 (a,b) の加算は、$|a| > |b|$ ならば、プログラム 1 の方法で高速に計算できる。厳密に $a+b=s+e$ が成り立ち $\frac{1}{2}ulp(\mathrm{s}) \geq |\mathrm{e}|$ となる。

プログラム 1 ：高速加算

```
void fast_two_sum( const double a,
       const double b, double &s, double &e )
{
    s = a + b ;
    e = b-( s - a ) ;
}
```

この加算プログラムには、$|a| > |b|$ の条件が付くが、次のように書くと計算量は増えるがこの条件なしで加算できる。

プログラム 2 ：加算

```
void two_sum( const double a,
       const double b, double &s, double &e )
{
    double v ;
    s = a + b ;
    v = s - a ;
    e = ( a - ( s - v ) ) + ( b - v ) ;
}
```

double 型の数値 2 個 (a,b) の乗算法は、まず倍精度数を二つに分割することから始める。定数 $\mathrm{con} = 2^{\mathrm{n}} + 1$ とする。この定数を使って、

プログラム 3 ：分割

```
void split( const double a, double &ah, double &al )
{
    double t, v, con ;
    t = con * a ;
    v = t - a ;
    ah = t - v ;
    al = a - ah ;
}
```

al には、a の下位 n ビットが入り、ah に残りの上位ビットが入る。n=26 とすると con=134217729.0 となる。この場合、ah と al はほぼ同じビット数の数値に分割される。それを使って以下のように、二つの倍精度数の乗算を行うことができる。

プログラム 4 ：乗算

```
void two_prod( const double a,
       const double b, double &p, double &e )
{
    double ah, al, bh, bl ;
    p = a * b ;
    split( a, ah, al ) ;
    split( b, bh, bl ) ;
    d = (( ah * bh - p ) + ah * bl + al * bh )
      + al * bl ;
}
```

この計算によって、$\mathrm{a} * \mathrm{b} = \mathrm{p} + \mathrm{e}$ が成り立ち $\frac{1}{2}ulp(\mathrm{p}) \geq |\mathrm{e}|$ となる。もし、C99 言語以降で定義されている fma(fused multiply add) 関数が使用できれば、この関数は中身は次の 2 行で書ける。fma(a,b,c)=a*b+c と定義される関数である。この関数では計算は 128 ビットで行い最終的に 64 ビッ

トに丸めた数値を返す関数である。したがって fma(a,b,-a*b) を計算することによって、a*b の下位 64 ビットが得られる。

プログラム 4 - 1 ：乗算

```
void two_prod( const double a,
       const double b, double &p, double &e )
{
    p = a * b ;
    e = fma( a, b, -p ) ;
}
```

この fma 命令は、最近の Intel 社の CPU では、ハードウエア命令になっているので、それを使えば、高速に乗算の計算できる。

これらのプログラムを利用して、double-double 型 4 倍精度数の加算と乗算のプログラムを作成出来る。プログラム 5、プログラム 6 に示す．このプログラムは、double-double 型 4 倍精度数 (quad) である a,b の積を計算する C++言語で記述したものである。このプログラムでは、計算結果は最下位ビットまで正確には計算されないため、このプログラムを再帰的に適用して 8 倍精度の計算はできない。

プログラム 5 ：4 倍精度加算

```
quad add( const quad &a, const quad &b )
{
  real16 c ;
  double s1, s2 ;
  two_sum( a.m0, b.m0, s1, s2 ) ;
  s2 = s2 + a.m1 + b.m1 ;
  fast_two_sum( s1, s2, c.m0, c.m1 ) ;
  return c ;
}
```

プログラム 6 ：4 倍精度乗算

```
quad mul( const quad & a, const quad &b )
{
  real16 c ;
  double z1, z2 ;
  two_prod( a.m0, b.m0, z1, z2 ) ;
  z2 = z2 + a.m0 * b.m1 + a.m1 * b.m0 ;
  fast_two_sum( z1, z2, c.m0, c.m1 ) ;
  return c ;
}
```

8 倍精度のプログラムも同様な方法で作成できる。ここで利用したプログラムは主に長谷川 [3] にある 8 倍精度のプログラムを参考に作成した。この 8 倍精度プログラムの四則演算の中に if 文などの条件文がないので、Intel 社の CPU のベクトル機能による並列化容易である思われる。double 型、4 倍精度型、8 倍精度型間の自由に計算も出来るようにプログラムを作成した。

2.1 入出力

入出力は、文字列として入出力し、その文字列を double-double 型の 4 倍精度浮動小数点や 8 倍精度浮動小数点に変換する。これを使えば、C++言語の入出力の方法で、4 倍精度浮動小数点数 a を

```
cin >> a ;          cout << a ;
```

として、入出力できる。C 言語の関数 scanf や printf を使って直接 4 倍精度数を入力することはできない。この場合は、一旦文字列として出力し、その文字列を出力することになる。上のように書式を指定しないで出力すると、既定の書式での出力される。既定の書式は、たとえば

```
set_format("%50.40f") ;
```

のように書いて設定できる。既定書式でない書式で、出力したい場合には

```
cout << to_string( a, "%50.40e" ) ;
```

という形式で書式を指定して出力できる。

2.2 4 倍精度および 8 倍精度用関数

減算は符号を変えた数値を加算することで行っている。この 4 倍精度計算プログラムでは、平方根、指数対数関数三角関数、逆三角関数、双曲線関数などの数学関数を準備した。floor 関数や絶対値などの関数も準備した。

3. 最良近似式

ある関数の計算を何回も計算するような場合、その関数の定義通り計算するより、多項式または有理関数でその関数を近似し、その近似式 [2] を計算した方が高速に精度よく計算できる場合がある。このような関数として、計算機に組み込まれている三角関数や指数関数、ライブラリーになっているいろいろな特殊関数などがある。

連続関数の多項式近似については、Weierstrass[6] が、「閉区間において連続な関数は、多項式によっていくらでも高精度で近似できる。」ことを証明している。この定理の証明方法と同じ方法で多項式を作ると、非常に高次の多項式となり、実際の計算には使いものにならない。実際の応用には、低次で精度の高い近似式を求める必要がある。区間 $[a,b]$ において、与えられた関数 $f(x)$ を、n 次の多項式 $p_n(x)$ で近似することを考える。このとき、$f(x)$ と $p_n(x)$ の距離 L を

$$L(f,p_n) \equiv \sup\{|f(x)-p_n(x)|; a \le x \le b\} \qquad (2)$$

と定義する。$p_n(x)$ の次数 n を一定に保つとき、$L(f,p_n)$ を最小にする関数を、最良近似関数という。関数 $f(x)$ を近似する多項式 $p_n(x)$ の差を $e_n(x)$ とする。すなわち

$$e_n(x) = f(x) - p_n(x) \qquad (3)$$

と定義する。$p_n(x)$ が最良近似式であるとき、 $e_n(x)$ は次の性質 [7] をもつことが知られている。

1). $e_n(x)$ の相隣る極大極小値は符号が異なる。
2). $e_n(x)$ の極値は、区間内で少なくとも (n+2) 個ある。
3). $e_n(x)$ の極大極小値の絶対値はすべて等しい。
この性質から、$p_n(x)$ を求めることができる。このような性質は、多項式の最良近似式だけでなく、有理関数の最良近似式も同様な性質をもつ。区間 $[a,b]$ において、与えられた関数 $f(x)$ を、分母が m 次の多項式 $q_m(x)$ 、分子が l 次の多項式 $p_l(x)$ である有理関数 $\frac{p_l(x)}{q_m(x)}$ で近似することを考える。分数は分子と分母に定数を掛けても同じものになるので、$q_m(0)=1$ として不定部分をなくすものとする。$f(x)$ とこの有理関数の差 $e(x)$ を

$$e(x) = w(x)\left\{f(x) - \frac{p_l(x)}{q_m(x)}\right\} \tag{4}$$

と定義する。$w(x)$ は、重み関数で区間 $[a,b]$ でゼロにならない関数である。このとき、$e(x)$ は上の $e_n(x)$ と同様な性質をもっている。ただし、$n = l+m$ である。多項式と同様に、有理関数の場合も同様に求めることができる。

最良近似は一般に解析的には求められないので、逐次近似法で求める。以下で示すプログラムは、山下 [9] による方法を高精度プログラム [4] を利用して作成したものである。

逐次近似計算を始めるには、初期値を設定しなければならない。まず反復計算で使う極値の位置を求める。誤差関数 $e(x)$ は、Chebyshev 多項式と似ている関数と仮定できるので、極値の位置を Chebyshev 多項式の極値と同じと仮定して、

$$x_j = (\frac{b-a}{2})\cos(\frac{2j}{n+1}\frac{\pi}{2}) + \frac{b+a}{2} \tag{5}$$

と置く。同様に誤差関数 $e(x)$ の零点 x_i も計算できる。Chebyshev 関数の零点と同じと仮定すると、

$$x_i = (\frac{b-a}{2})\cos(\frac{2i+1}{n+1}\frac{\pi}{2}) + \frac{b+a}{2} \tag{6}$$

が得られる。$p_l(x)$ 、$q_m(x)$ を以下のような多項式と仮定する。

$$p_l(x) = \sum_{k=0}^{l} a_k x^k \tag{7}$$

$$q_m(x) = \sum_{k=0}^{m} b_k x^k \tag{8}$$

(6) で示される誤差関数 $e(x)$ の零点で

$$f(x_i) - \frac{p_l(x_i)}{q_m(x_i)} = 0 \tag{9}$$

が成り立つ。この式を変形すると

$$p_l(x_i) - f(x_i)\{q_m(x_i) - 1\} = f(x_i) \tag{10}$$

となる。この式に、(8)、(8) を (10) に代入すると

$$\sum_{k=0}^{l} x_i a_k + \sum_{k=1}^{m}(-f(x_i)x_i^k)b_k = f(x_i) \tag{11}$$

が得られる。(11) を解くことによって、a_k と b_k の初期値が得られる。

何回かの補正を行って得られた近似有理関数を $\frac{p_l(x)}{q_m(x)}$ とし、最良近似式を $\frac{p_l^*(x)}{q_m^*(x)}$ とする。これらの誤算関数をそれぞれ $e(x)$ 、$e^*(x)$ とする。これらは

$$e(x) = w(x)\left\{f(x) - \frac{p_l(x)}{q_m(x)}\right\} \tag{12}$$

$$e^*(x) = w(x)\left\{f(x) - \frac{p_l^*(x)}{q_m^*(x)}\right\} \tag{13}$$

と定義する。$\Delta p_l(x)$ 、$\Delta q_m(x)$ をそれぞれ $p_l(x)$ 、$q_m(x)$ の補正として、

$$p_l^*(x) = p_l(x) + \Delta p_l(x) \tag{14}$$

$$q_m^*(x) = q_m(x) + \Delta q_m(x) \tag{15}$$

が成り立つ。また、3) の条件から最良近似の誤差関数 $e^*(x)$ の極大または極小は絶対値が同じになるから、その絶対値の値を ρ とする。x_j を $e^*(x)$ の極値をとる位置だとすると $e^*(x_j) = (-1)^j\rho$ が成り立つ。このとき、

$$\begin{aligned} e(x_j) - e^*(x_j) &= w(x_j)\left\{f(x_j) - \frac{p_l(x_j)}{q_m(x_j)}\right\} \\ &\quad - w(x_j)\left\{f(x_j) - \frac{p_l^*(x_j)}{q_m^*(x_j)}\right\} \\ &= w(x_j)\left\{\frac{p_l^*(x_j)}{q_m^*(x_j)} - \frac{p_l(x_j)}{q_m(x_j)}\right\} \\ &= w(x_j)\frac{\Delta p_l^*(x_j)q_m(x_j) - \Delta q_m^*(x_j)p_l(x_j)}{q_m(x_j)\{q_m(x_j) + \Delta q_m(x_j)\}} \quad (16) \\ &= e(x_j) - (-1)^j\rho \quad (17) \end{aligned}$$

$\Delta q_m(x)$ が $q_m(x)$ に比べて、非常に小さいとすると、(17)、(17) から

$$w(x_j)\frac{\Delta p_l^*(x_j)q_m(x_j) - \Delta q_m^*(x_j)p_l(x_j)}{q_m(x_j)^2} = e(x_j) - (-1)^j\rho \tag{18}$$

が得られる。相隣る極値での式 (18) を加えると

$$\begin{aligned} & w\ (x_j)\frac{\Delta p_l^*(x_j)q_m(x_j) - \Delta q_m^*(x_j)p_l(x_j)}{q_m(x_j)^2} \\ & + w(x_{j+1})\frac{\Delta p_l^*(x_{j+1})q_m(x_{j+1}) - \Delta q_m^*(x_{j+1})p_l(x_{j+1})}{q_m(x_{j+1})^2} \\ & = e(x_j) + e(x_{j+1}) \end{aligned} \tag{19}$$

$\Delta p_l(x)$ 、$\Delta q_m(x)$ は多項式であるから

$$\Delta p_l(x) = \sum_{k=0}^{l} \Delta a_k x^k \tag{20}$$

$$\Delta q_m(x) = \sum_{k=1}^{m} \Delta b_k x^k \tag{21}$$

と置ける。これを (19) に代入して、

$$\sum_{k=0}^{l}\left\{\frac{w(x_j)x_j^k}{q_m(x_j)} + \frac{w(x_{j+1})x_{j+1}^k}{q_m(x_{j+1})}\right\}\Delta a_k$$

$$-\sum_{k=1}^{m}\left\{\frac{w(x_j)x_j^k p_l(x_j)}{q_m(x_j)^2}+\frac{w(x_{j+1})x_{j+1}^k p_l(x_{j+1})}{q_m(x_{j+1})^2}\right\}\Delta b_k = e(x_j)+e(x_{j+1}) \qquad j=0,1,\cdots,n \tag{22}$$

この (22) の $(n+1)$ 元の方程式を解くことによって、Δa_k および Δb_k を求める。この結果を (21)、(21) に代入することによって、$\Delta p_l(x)$ 、$\Delta q_m(x)$ を計算し、それぞれ $p_l(x)$ 、$q_m(x)$ を補正する。

得られた $p_l(x)$ 、$q_m(x)$ から、極値と極値の位置を二分法によって求める。極値の値の絶対値の最大値と最小値の比が小さいならば、最良近似式が求まったと判断するそうでない場合には、(22) の式をもう一度計算し有理式を補正する。

4. ガンマ関数の作成

ガンマ関数 [1] は階乗を拡張したもので、次のような積分で定義される。これは C 言語では、tgamma(x) として定義されている関数である。

$$\Gamma(x)=\int_0^\infty t^{x-1}e^{-t}dt$$

ガンマ関数は階乗の拡張なので、x が整数値のとき部分積分法によって次の式が得られる。

$$\Gamma(x)=(x-1)!$$

この関数は、次のように漸近展開 (Stirling の公式)[1] できる。

$$\log(\Gamma(x)) = (x-\frac{1}{2})\log x - x + \frac{1}{2}\log(2\pi) + \sum_{m=1}^{\infty}\frac{B_{2m}}{2m(2m-1)x^{2m-1}} \tag{23}$$

この級数は、発散級数なので、級数の値を計算するには、何項かまで計算し、項の絶対値が増加し始めるところで計算を打ち切らなければならない。この級数の値と計算値の違いは、打ち切った項の大きさ程度となるから、級数の項が十分に小さくなるかまたは出来るだけ小さくなったところで計算を打ち切るようにしなければならない。級数の項を小さくなる様にするには、x を大きくする必要がある。級数の項が小さくなれは、漸近級数を精度よく計算できる事になる。x を大きくするには、ガンマ関数の性質

$$\Gamma(x+1)=x\Gamma(x)$$

を何回か使って、n を整数とすると次の関係式が得られる。

$$\Gamma(x)=\frac{\Gamma(x+n)}{x(x+1)(x+2)\cdots(x+n-1)} \tag{24}$$

この公式を使って、計算するガンマ関数の x の値をいくらでも大きくすることができる。大きくした x を使って、(24) の式からガンマ関数の値を求め、(24) の式から元の x のガンマ関数の値を計算する。x を十分に大きいならば、(24) の級数は十分に速く収束し、容易に計算出来る。(24) によって元の x の値の関数値に戻すのが大変になる。この兼ね合いから、(24) の式を計算する最適な x の値を決めることができる。この値は、計算精度に依存する。

この漸近級数を利用し、高精度計算して、要求する精度の最良近似式 (minimax 近似式) を作成した。

(24) の級数にはベルヌイ数 B_n が含まれている。ベルヌイ数は次の式で定義される有理数である。

$$\frac{t}{e^t-1}=\sum_{n=0}^{\infty}B_n\frac{t^n}{n!}$$

$B_0=1, B_1=-\frac{1}{2}, B_2=\frac{1}{6}, B_3=0, B_4=-\frac{1}{30},\cdots$ である。

ガンマ関数 $\Gamma(x)$ を区間 $[2,3]$ で有理関数近似式で近似する。この区間では、ガンマ関数は単調増加で、大きな変化がないことから選んだ。具体的には $\Gamma(x+2)$ を区間 $[0,1]$ を近似する。4 倍精度の精度は、10 進数でおよそ 32 桁の精度なので、近似式は、元の関数との差は最大でも 10^{-32} 以下にする必要がある。この条件を満たす次のような有理関数最良近似式 $R(x)$ を計算する。

$$R(x)=\frac{p_0+p_1x+p_2x^2+\cdots+p_lx^l}{1+q_1x+q_2x^2+\cdots+q_mx^m}$$

分子の次数 $l=12$、分母の次数 $m=11$ とすると、誤差 $\rho(x)=|R(x)-\Gamma(x+2)|$ は、区間 $[0,1]$ で 25 箇所で極値を持つ。その値の絶対値はほぼ一定で約 $1.883018108679\times10^{-33}$ であった。このことから、得られた有理関数は最良近似式であることがわかる。

このときの分子の定数部分 p_0 は、

$$1.00000000000000000000000000000000018830181086 79\cdots$$

となる。ガンマ関数は、引数が正の整数のとき整数になるので、この値を 1.0 に変更した。このようにすれば、正の整数値のとき、正確に整数値になる。このように変更すると他の部分で関数値に誤差が入るが、誤差は許容範囲で問題が無いと判断した。

この 4 倍精度関数の精度等を確認するため、10 や 20 の関数値や厳密な値が知られているものを計算した。

```
Gamma(10)  =   362880.00000000000000000000000000000000000000
Gamma(20)  = 121645100408832000.00000000000000000000000000
Gamma(1.5) =   0.88622692545275801364908374167057324431552 0
```

整数値のガンマ関数は整数値となり容易に正しいことがわかる。最後の数値は厳密な値が知られている $\Gamma(\frac{3}{2})=\frac{\sqrt{\pi}}{2}$ である。この数値と関数の計算値を比較すると小数点以下 32 桁一致しており、4 倍精度数の限界まで正しいことがわかる。

8 倍精度の場合、4 倍精度の最良近似と異なりほとんど前例がない。ここでは、前節で説明した最良近似式を計算プログラムを利用して計算した。計算には高精度計算プログラムを利用した。8 倍精度用有理関数近似式の分子の次数 $l=21$、

分母の次数 $m = 21$ とすると、誤差 $\rho(x) = |R(x) - \Gamma(x+2)|$ は、区間 $[0, 1]$ で 43 箇所で極値を持つ。その値の絶対値はほぼ一定で約 $2.313790623591 \times 10^{-67}$ である。このことから、この有理関数は最良近似式であることがわかる。

8 倍精度用有理関数近似式の分子の定数部分 p_0 も 4 倍精度の場合と同じ理由で 1.0 とすることができる。

8 倍精度関数の精度等を確認するため、厳密な値が知られているものを計算した。

```
Gamma(20)  =   1216451004 08832000.00000000000000000000000
Gamma(1.5) = 0.8862269254 5275801364 9083741670 5725913987
             7472806119 3564106903 8949462643 91151
```

最後の数値は $\Gamma(\frac{3}{2}) = \frac{\sqrt{\pi}}{2}$ である。この数値と関数計算値を比較すると小数点以下 64 桁一致しており、8 倍精度数のほぼ限界まで正しいことがわかる。

4.1 ガンマ関数の対数

ガンマ関数の対数 $(\log(\Gamma(x)))$ は、C 言語では lgamma(x) と定義されている関数である。通常のガンマ関数と同様な区間で最良近似を求めた。すなわち関数 $\log(\Gamma(2+x))$ を区間 $[0, 1]$ で最良近似式を計算した。

4 倍精度の有理関数近似式の計算では、分子の次数 $l = 11$、分母の次数 $m = 10$ とすると、誤差 $\rho(x) = |R(x) - \log(\Gamma(x+2))|$ は、区間 $[0, 1]$ で 23 箇所で極値を持つ。その値の絶対値はほぼ一定で約 $5.290102040016 \times 10^{-34}$ である。このことから、この有理関数は最良近似式であり、$\Gamma(x+2)$ の近似式より次数の低い近似式で精度良く近似できることがわかる。

8 倍精度の有理関数近似式の分子の次数 $l = 21$、分母の次数を $m = 20$ とすると、誤差 $\rho(x) = |R(x) - \log(\Gamma(x+2))|$ は、区間 $[0, 1]$ で 43 箇所で極値を持つ。その値の絶対値はほぼ一定で約 $6.27898334 \times 10^{-66}$ である。このことから、この有理関数は最良近似式であり、8 倍精度でも $\Gamma(x+2)$ の近似式より次数の低い近似式で精度良く近似できることがわかる。

5. 誤差関数の作成

誤差関数とは次のように定義されている。

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

この関数は次のように級数展開できる。

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{n!(2n+1)} \tag{25}$$

この級数の収束半径は無限大であり、(25) の級数の x に数値を代入することによって、関数値を容易に計算できるように思えるが、x の値が少し大きくなると計算精度が極端に悪くなる。たとえば、$x = 5$ を代入するとこの級数の第 n 項

$$a_n = \frac{(-1)^n x^{2n+1}}{n!(2n+1)}$$

は第 23 項が絶対値最大となる。第 22 項、23 項、24 項の値は、次のようになる。

```
22 項     561915599.5981104
23 項    -584787279.8777744
24 項     584290011.7826403
```

これらの項をすべて加えると、9 桁以上の桁落ちが起こり、計算値は、0.99999999999846254 となる。計算精度は通常 16 桁程度であるから、9 桁の桁落ちが生じ最終結果は 7 桁程度の精度となる。このためこの方法はあまり実用的であるとはいえない。

しかしながら、$|x| < 2$ の範囲では、収束良好で、各項が非常に大きくなることがないので、この関数を計算するためにこの級数を使うことができる。もし $|x| \leq 2$ ならば、作成したプログラムでも (25) の級数を利用して計算した。

$|x| > 2$ 場合、いろいろな近似式による計算法が提案されている。これらの方法を高次の最良近似に作成に試みたが、計算が収束しないなどの問題が起き計算することが出来なかった。

今回は、連分数による公式で計算するように作成した。

$$\sqrt{\pi} e^{-x^2} \mathrm{erfc}(x) = \frac{1}{x + K_{n=1}^{\infty} \frac{n/2}{x}}$$

ここで、$\mathrm{erfc}(x)$ は相補誤差関数と呼ばれ、$\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$ である。この連分数は、漸化式を使って計算するが、計算が収束する前に分子や分母が大きくなりオーバーフロー状態になる場合がある。この場合、分子と分母を共通の数値で割り絶対値が小さい数値にすれば、この問題は解決する。4 倍精度数や 8 倍精度数は、2^k を掛け算するとき、各桁を 2^k を掛ければ桁上がり等は起こらないので効率的に計算できる。

プログラムでは、$2^{-100} = 7.8886090522101180541e-31$ を掛けることによって、オーバーフローを避けるようにした。

x が大きい数の時、漸近展開を使用することを検討したが、4 倍精度数や 8 倍精度数の指数部の大きさ考えると、それほど有効でないようなので、今回は漸近展開は利用しなかった。

6. まとめ

C 言語や C++言語等で定義されている特殊関数の double-double 型の 4 倍精度数や quad-double 型の 8 倍精度数に対応したプログラムを作成した。これまでこれらの特殊関数が現れた時、これらの特殊関数を作成するか、特殊関数を持っている別な言語で記述し直さなければならない。このプログラムによって、そのようなことが無くなるため、これらの高精度数値による計算プログラムが容易になると思われる。

C 言語の最新版では、ベッセル関数やリーマンのゼータ

関数が準備されることになっているので、これらの特殊関数に対応するプログラムを作成する必要があると思われる。

参考文献

[1] M. Abramowitz, I. A. Stegun , Handbook of Mathmatical Functions, Dover (1972)
[2] 浜田 穂積, 近似式のプログラミング、培風館，(1995)
[3] 長谷川 秀彦, 高精度演算を用いた混合精度反復法, 応用数理学会三部会連携「応用数理セミナー」資料集, (2013),4–35
[4] 平山 弘, C++ 言語による高精度計算パッケージの開発、日本応用数理学会論文誌，5 (1995),307–318
[5] 小武守, 長谷川, 藤井, 西田, 反復法ライブラリ向け 4 倍精度演算の実装と SSE2 を用いた高速化, 情報処理学会誌 コンピューティングシステム,1(2008),73–84
[6] R. クーラン、D. ヒルベルト (斉藤監訳), 数理物理学の方法,　東京図書, (1959)
[7] 宇野利雄, 数値計算, 朝倉書店, (1963)
[8] 山田進, 佐々成正, 今村俊幸, 町田昌彦, 4 倍精度基本線形代数ルーチン群 QPBLAS の紹介とアプリケーションへの応用, 情報処理学会研究報告,vol.2012-HPC-137,No.23(2012)
[9] 山下眞一郎, 最良近似式を求めるプログラム, 情報処理、Vol.10,6(1969) ,pp.442-446
[10] Yozo Hida, Xiaoye S. Li, David H. Bailey, Library for Double-Double and Quad-Double Arithmetic, Proc. 15th Symposium on Computer Algorithmetic, (2007),155–162
[11] Yozo Hida, Xiaoye S. Li, David H. Bailey, Algorithms for Quad-Double Precision Floating Point Arithmetic, Lawrence Berkeley National Laboratory, (2000)