

[研究論文] 4倍精度および8倍精度演算プログラムの開発と性能評価

平山弘・加藤俊二

自動車システム開発工学科

Development of Double-double and Quad-double Precision Arithmetic Program and Performance Evaluation

Hiroshi HIRAYAMA, Shunji KATO

Abstract

The arithmetic program for double-double precision and quad-double floating point numbers which can consist of two and four double precision floating point numbers were created respectively. The Auxiliary program created by C++ language for the input and output of these numbers.

The calculation function of double-double and quad-double precision can be given to C++ language which does not usually have double-double and quad-double precision by using this program. By this arithmetic routine, not only addition, subtraction, multiplication and division but absolute value, square root, exponential and logarithmic function, trigonometric functions and its inverse function, hyperbolic function and its inverse function were prepared.

The existing C++ program can be easily converted to double-double and quad-double precision program by using this program library. Many programs can be executed in high precision. We also examined the performance of these programs.

Keywords: Double-double and quad-double precision number, C++ Programming Language

1. はじめに

これまでの数値計算の多くは、計算機のハードウェアで実行できる倍精度浮動小数点数の演算で済むが、計算速度の増加とともに丸め誤差が大きくなり、もう少し高い精度の計算が簡単にできる環境が望まれてきている。計算の品質をあげるための高精度計算の要求も高まって来ている。

最近開発された多くの計算機は、性能が非常に良くなっているが4倍精度をハードウェ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。このような状況がある程度克服するため、Bailey⁽⁶⁾⁷⁾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。

計算方法は単純で容易に四則演算等のプログラムを作成できる。これらの数値の入出力する部分の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。

4倍精度を持つプログラム言語では、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。プログラムが単純であるためか、プログラム言語の4倍精度より高速に計算ができる。4倍精度数を持たない多くのC++言語等では、このため、4倍精度数を使うことが非常に難しいものになっている。

本論文では、C++言語で作成された簡略化した多倍長演算プログラム³⁾を利用して、4倍精度数と8倍精度数の入出力ルーチンを作成し、使いやすい高精度演算ルーチンを作成することができた。また、それらのプログラムの有用性および性能を例を挙げて示した。

ここで取り上げた例題の計算時間は、Intel i7-4790K CPU 4.00GHzで実行し、測定した時間である。

2. double-double型4倍精度数の計算アルゴリズム

Baileyのdouble-doubleアルゴリズム⁴⁾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、下位桁を 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 を二つの倍精度変数 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。

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

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 倍精度はソフトウェアで作成されているため、計算速度はハードウェ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⁵⁾。

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。

この場合、厳密に $a+b=s+e$ が成り立ち $\frac{1}{2} ulp(s) \geq |e|$ とする。ups(s) は数値 s の最下位ビットを表す。

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) の乗算は、まず倍精度数を二つに分割することから始める。

定数 $con = 2^n + 1$ 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=27 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、 $a * b = p + e$ が成り立ち $\frac{1}{2} ulp(p) \geq |e|$ となる。もし、C99 以降の言語で定義されている

fma(fused multiply add) 関数如果使用できれば、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 の下位ビットが得られる。

プログラム 5 : 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、ハードウェ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と期待できる。

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 6、プログラム 7 に示す。

これから加算、乗算の演算数はそれぞれ 11flops, 24flops であることがわかる。

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。

プログラム 6：4 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 7：4 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に作成できる。ここで利用したプログラムは主に長谷川²⁾にある 8 倍精度のプログラムを参考に作成した。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

IEEE 型の 4 倍精度の浮動小数点数を持つ処理系では、この 4 倍精度を使って入出力が行えるので、簡単に入出力ができる。多くの C++ 言語では 4 倍精度の浮動小数点を持っていないため、そのための入出力プログラムを準備しなければならない。ここでは、簡易化された多倍長計算プログラムを利用して、入出力プログラムを作成した。

入力は、文字列として入出力し、その文字列を double-double 型の 4 倍精度浮動小数点に変換する。これを使えば、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 精度用関数

加算と乗算は前節のアルゴリズムで計算できる。

減算は符号を変えた数値を加算することで出来る。

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。

これらの関数は近似式¹⁾を使わないで、Taylor 展開などの式を利用して計算している。

3. 4 倍精度 8 倍精度演算プログラム使用法

4 倍精度 8 倍精度演算のライブラリの使用は、出来るだけ double 型や float 型などの通常の型と同じように使えるように作成した。ここでは、4 倍精度数の名前を real16、8 倍精度数の名前を real32 としている。

独自に開発した数値を使ってプログラムに直接書く場合、通常の型のように書くことはできない。

プログラムに記述された数値 (数値文字列) は、コンパイラによってコンパイラの仕様にある数値型だと仮定してコンパイルされるためである。

例えば

```
real16 p=3.14159265358979323846264338327950288;
```

と書くと、数値文字列は最も精度の高い double 型の数値と解釈され、15 桁程度の精度の数値に丸められる。

その数値が 4 倍精度数に代入される。これを避けるには、数値を次のように文字列としてプログラムに記述しなければならない。

```
real16 p=real16("3.14159265358979323846264338327950288");
```

このように記述すると、プログラムの実行時に、文字列から 4 倍精度数に変換作業が入る。これを避けるために、次のように書くこともできる。

```
real16 p={3.1415926535897931e+00, 1.2246467991473532e-16} ;
real32 p={3.1415926535897931e+00, 1.2246467991473532e-16,
-2.9947698097183397e-33, 1.1124542208633653e-49} ;
```

コンパイラーは、記述された 10 進数の数値に最も近い 2 進数表示の数値に変換するから、上のように宣言時に値を設定できる。倍精度の数値は 10 進数で約 16 桁の精度であるから、17 桁以上指定すれば、正確な値を設定できるはずである。このため、ここでは 17 桁指定している。これは使用している C++コンパイラーでは有効数値を 17 桁以上出力するように指定しても 18 桁以上はすべて 0 が出力されるからでもある。

使用例として、次の 2 次方程式を解くプログラムを 4 倍精度のプログラムに変換する。次のプログラムで示したプログラムは、2 次方程式のプログラムである。

このプログラムを 4 倍精度に変換したプログラムをプログラム 8 に示す。

また、8 倍精度で計算した結果も示す。

プログラム 8 : 倍精度 2 次方程式の解法

```
1: #include <iostream>
2: #include <cmath>
3: using namespace std;
4: int main()
5: {
6:     double a, b, c, d, x1, x2;
7:     a=2; b=7.5; c=-12.2;
8:     d=b*b-4*a*c; d = sqrt(d);
9:     x1=(-b+d)/(2*a); x2=(-b-d)/(2*a);
10:    cout << "x1=" << x1 << endl;
11:    cout << "    " << a*x1*x1+b*x1+c << endl;
12:    cout << "x2=" << x2 << endl;
13:    cout << "    " << a*x2*x2+b*x2+c << endl;
14: }
```

プログラム 9 : 4 倍精度 2 次方程式の解法

```
1: #include "r32.h"
2: int main()
3: {
4:     real16 a, b, c, d, x1, x2;
5:     a=2; b=7.5; c=real16("-12.2");
6:     d=b*b-4*a*c; d = sqrt(d);
7:     x1=(-b+d)/(2*a); x2=(-b-d)/(2*a);
8:     set_format("%35.32g");
9:     cout << "x1=" << x1 << endl;
10:    cout << "    " << a*x1*x1+b*x1+c << endl;
11:    cout << "x2=" << x2 << endl;
12:    cout << "    " << a*x2*x2+b*x2+c << endl;
13: }
```

4 倍精度のプログラムに変更するために、main 文の前にある宣言部分を変更、double を real16 に変更し、定数を real16 関数を使って変更した。

インクルードファイル“r32.h”では 4 倍精度と 8 倍精度の宣言なされているのでこのファイルを読み込むことによって、これらの宣言を行う。

```
a=2; b=7.5;
```

の部分は、通常変更されるが、定数 2 は整数として、誤差のない数値にコンパイルされる。

同様に定数 7.5 も倍精度数として誤差のない数値と解釈されるので、変更なしで利用できる。しかし、-12.2 のような場合、倍精度数としてコンパイルされると-12.2 は 2 進数としては無限に小数になるので、53 ビットに丸められ倍精度数としても厳密に表現出来ない。

このため、4 倍精度の数値としては精度が不十分である。次のように変更する。

```
c=real16("-12.2");
```

このように記述すれば、途中に倍精度数が入らないので、文字列から直接 4 倍精度数に変換されるので、4 倍精度の精度を持つ数値に変換される。

これらのプログラムを実行すると、それぞれ以下のように出力される。x1 および x2 は解であり、それを元の 2 次方程式に代入した結果がその下に示している。

倍精度計算結果	4 倍精度計算結果
x1=1.22591	x1= 1.2259071253425182195488491564024
-1.77636e-015	1.97215226305252951352932141e-31
x2=-4.97591	x2=-4.9759071253425182195488491564024
-3.55271e-015	-1.97215226305252951352932141e-31

代入結果を見ると、倍精度では約 15 桁、4 倍精度では約 31 桁の精度で計算ができることがわかる。

4 倍精度プログラムの real16 を real32 に変更し、計算結果を 64 桁出力するために書式を“%55.50g”に変更し、実行する。

8 倍精度計算結果

```
x1= 1.22590712534251821954884915640243278289051258756912
5.1655888536933305186254437946417046918962054134892e-63
x2= -4.97590712534251821954884915640243278289051258756912
1.8231490071858813595148625157558957736104254400550e-63
```

この結果から、8 倍精度で計算すると、約 63 桁の精度で計算できることがわかる。

4. 4 倍精度 8 倍精度演算の性能

ここでは、いろいろな計算を行ったときの性能評価を行った。

4.1 4 倍精度の性能

プログラム言語が提供するアセンブラーで記述された 4 倍精度の演算と本論文で扱っている二つの倍精度浮動小数点数で実現されている 4 倍精度との速度比較を行った。C++言語で 4 倍精度を扱えるコンパイラーはあまりな

いので、今回は無料で使える 64 ビットの gfortran コンパイラーを使って比較した。

次のような n 元連立一次方程式を解いた。今回では $n=250, 500, 750, 1000$ の場合を計算した。その結果を表 1 に示す。

$$Ax = b$$

行列 A の係数 $a_{i,j}$ として

$$a_{i,j} = \begin{cases} i + 10 & i = j \\ 1 & i \neq j \end{cases}$$

を与えた。 b は $b_i = i (i = 1, \dots, n)$ とした。

表 1 倍精度および 4 倍精度による N 元連立一次方程式の計算時間(秒)

N	real (16)	real16	real (8)
250	0.611	0.294	0.031
500	4.960	2.368	0.303
750	22.596	11.020	1.803
1000	60.771	30.445	9.213

ここで、 N は方程式の大きさ、real (16) はプログラム言語が提供する 4 倍精度、real16 は double-double 型の 4 倍精度、real (8) は倍精度を示す。この実行結果は、FMA 命令を使っていない結果である。FMA 命令を使えばもう少し速い結果が得られると思われる。

この結果を見ると、double-double 型の 4 倍精度は、プログラム言語にある 4 倍精度の約 2 倍の速さで計算できることがわかる。

この結果から、double-double 型の 4 倍精度が注目され、使用されるようになったと思われる。

4.2 連立一次方程式

行列のテンプレートを使って、前節と同じ問題を解く。

計算手順は少し異なり、係数行列の逆行列を計算し、逆行列を使って解く。以下にそのプログラムを示す。

このとき使ったコンパイラーは Visual Studio 2013 の 64 ビット C++ コンパイラーを使用した。コンパイラー・オプションとして、/EHsc /Ox を使用した。

```
1: #include "matrix_template.h"
2: typedef matrix_template<double> matrix;
3: void main()
4: {
5:     int n=500;
6:     matrix a(n,n), b(n), c(n,n), x(n);
7:     for( int i=1; i<=n; i++){
8:         for( int j=1; j<=n; j++){
9:             a(i,j)=1;
10:        }
```

```
11:        a(i,i)=10+i;
12:        b(i)=i;
13:    }
14:    c=invers(a); // 逆行列の計算
15:    x=c*b; // 逆行列を掛けて解を計算
16:    cout << x << endl; // 結果を出力
17: }
```

このプログラムを倍精度、double-double 型の 4 倍精度、quad-double 型の 8 倍精度で実行した。double-double 型および quad-double 型の数値ではハードウェアの FMA を使用したプログラムと未使用のプログラムでそれぞれ実行した。以下の表 2 にその結果を示した。

表 2 4 倍精度および 8 倍精度の計算時間(秒)

N	double	real16	FMA16	real32	FMA32
250	0.009	0.115	0.082	1.166	0.975
500	0.062	0.895	0.520	9.110	7.601
750	0.215	2.898	2.898	30.618	25.462
1000	0.501	6.818	6.818	72.360	60.090
1250	1.028	13.377	13.377	141.040	117.167
1500	1.912	23.143	23.143	242.843	203.001

ここで、double は倍精度数、real16 は double-double 型の 4 倍精度数、FMA16 は double-double 型で FMA を使った 4 倍精度数、real32 は quad-double 型の 8 倍精度数、FMA32 は double-double 型で FMA を使った 8 倍精度数で実行した結果である。

4 倍精度の計算では、FMA 命令を使うことによって、約 1.66 倍程度の速度になった。8 倍精度では、約 1.2 倍程度の速度になった。

FMA 命令では、乗算は高速化されるが、加減算は高速化されない。このため、相対的に加減算の命令が多い 8 倍精度の計算では、4 倍精度に比べ速度の向上が少なかったと思われる。

Intel 社の CPU では、FMA を同時に複数同時実行できるが、今回のプログラムでは複数の FMA を使う命令は使用しなかった。

計算問題としては、この節の C++ 言語を使った計算法が時間がかかるはずであるが、前節の fortran を使用したものと比較して、約 4 倍以上高速であった。

これは、fortran で使われている命令が、64 ビット命令であるが、最近の CPU に搭載された命令が使用されていないからではないかと推測される。

4.3 数値積分

4 倍精度、8 倍精度の二重指数型数値積分公式を作成し、次の積分を計算した。結果がわかりやすい数値になり、 I_3 の積分を除いて積分区間の端点で微分不可能な関数を選んだ。

このような関数は、ガウスの数値積分法やシンプソンの公式等では計算が困難な問題である。

$$I_1 = \int_0^1 \sqrt{x} \, dx \quad I_2 = \int_0^1 \frac{dx}{\sqrt{x}}$$

$$I_3 = \int_0^1 \frac{4dx}{1+x^2}$$

これを計算すると次のような結果が得られる。

4 倍精度の数値積分計算

[illegible]

誤差 = 6.7792734e-032 標本点数 = 131

[illegible]

誤差 = 6.1629758e-032 標本点数 = 155

$T_3=3.14159265358979323846264338327923471$

誤差 = 2.7117094e-031 標本点数 = 131

8 倍精度の数値積分計算

T1=0.666

666666666666666666666666666666034

誤差 = 6.2670747e-064 標本点数 = 301

[illegible]

99999999999999999999999999999998443

誤差 = 1.5572731e-063 標本点数 = 347

T3=3. 14159265358979323846264338327950288

4197169399375105820974944588105

誤差 = 4.1970409e-063 標本点数 = 301

計算結果は、4 倍精度の計算では、4 倍精度の精度約 32 桁を超える 35 桁を表示させ、8 倍精度の計算では、8 倍精度の精度約 64 桁を超える 66 桁を表示した。

精度が2倍になるため、標本点数がおよそ2倍となり、理論的にも一致する結果であることがわかる。

4.4 複素数計算

C++言語にも浮動小数点数から複素数を構成するテンプレートが準備されている。このテンプレートプログラムは、float と double 専用のプログラムで、ユーザーが定義した数値に対しては使用することが出来ない。

複素数の絶対値を計算するために、オーバーフローを避けるために、複素数 $a + bi$ に対して、単純に $\sqrt{a^2 + b^2}$ を計算しない。

たとえば、 $|a|$ と $|b|$ の大きい数値を $|a|$ とすると、 $|a|\sqrt{1+\frac{b^2}{a^2}}$ と計算する。このとき、扱う数値を long double に変換して、計算を行っている。

ユーザー定義の数値は long double には変換できない数値があり得るので、この場合この複素数テンプレート・プログラムが利用できなくなる。

また、このプログラムの中には、無限大(inf)や非数(NaN)が使われており、プログラムとしては非常に完成度は高いが、通常ユーザー定義の数値では、通常無限大や非数のような数値を定義することはないので、このプログラムはほぼ使えないことになる。

float や double に依存しない複素数テンプレート・プログラムを作成し、4 倍精度数や 8 倍精度の複素数を利用できるようにした。この複素数を使って、次の 6 次の代数方程式の複素根を計算する。この方程式は、実根を持たない方程式である。

$$2x^6+x^4+3x^3+6x^2+x+4=0$$

この代数方程式を初期値 $10+10i$ として Newton 法で解く。このときの反復公式は、次のようになる。複素数でも実数でも公式は同じになる。

$$x_{n+1} = x_n - \frac{2x_n^6 + x_n^4 + 3x_n^3 + 6x_n^2 + x_n + 4}{12x_n^5 + 4x_n^3 + 9x_n^2 + 12x_n + 1}$$

収束条件 $|x_{n+1} - x_n| < 10^{-50}$ を満たすまで計算した。

21 回の反復計算で収束し

$x=0.9087358725072740284570262477459685607528250$

+1.1723018875633025660028101307275121329547931i

が得られる。得られた複素数解を元の式に代入すると $-9.02 \times 10^{-64} + 2.23 \times 10^{-63}i$ となり、この数値が解であることがわかる。誤差の値から 8 倍精度で計算されていることもわかる。

4.5 指数部の拡張

4 倍精度、8 倍精度と仮数部分は大きくなるが、指数部は倍精度と同じ程度である。このために、指数部を 32 ビット等に拡張するテンプレート・プログラム (ee_real_template.h) を作成した。このプログラムは比較的簡単に作成することができる。

これを使えば、アンダーフローやオーバーフローことをあまり心配しないで、プログラムを作成できる。以下に 10000! と 12345^{12345} を計算するプログラムを示す。

```

1: #include "real1632.h"
2: #include "ee_real_template.h"
3: using namespace std ;
4: typedef ee_real_template<real32> ereal ;
5: void main()
6: {
7:     ereal x, y ;
8:     x = 1 ;
9:     for( int i=1 ; i<=10000 ; i++ )
10:    {
11:        y = ereal( i ) ;
12:        x *= y ;
13:    }

```

```

14:     cout << x << endl ;
15:     int n=12345 ;
16:     x = pow( ereal(n), n ) ;
17:     cout << x << endl ;
18: }

```

このプログラムを実行すると次の結果が得られる。

```
2.846259680917054518906413212119868890148051e+35659
```

```
2.867865225003669442826455604983179963965839e+50509
```

10000! は 35660 桁の数で、12345¹²³⁴⁵ は 50510 桁の数値になることがわかる。このプログラムを利用して、行列計算を行ってみた。このとき、計算時間は、指数部を拡張する前の数値の 5~20 倍になった。

プログラムが簡単なので、2 倍程度の時間で同じ計算問題を計算できると期待したが残念ながら予想よりかなり低速であった。このプログラムがなぜ低速かを調べ高速化をはかる必要がある。

5. まとめ

仮数部を 2 倍、4 倍にする 4 倍精度数と 8 倍精度数の演算プログラムとその入出力プログラムを C++ 言語を使って作成した。

4 倍精度のプログラムは、C 言語等で提供されるプログラムより高速に計算できるので、4 倍精度の計算の高速化が可能である。C++ 言語で作られているので、倍精度のプログラム等を容易に 4 倍精度や 8 倍精度のプログラムに変更できる。

4 倍精度や 8 倍精度の複素数計算するために、複素数化するためのテンプレート・プログラムを作成した。このプログラムによって、複素数プログラムも容易に計算が可能になる。

仮数部が長くなったので指数部を拡張するためのテンプレート・プログラムを作成した。

数学関数の計算は可能であるが、計算速度、計算精度等について、さらに検討する必要がある。

このプログラムを使うためのインクルード・ファイル (real1632.h) は C++ 言語で約 1100 行であった。

real1632.cpp のプログラムは、約 4100 行で約半分の 2100 行が入出力のプログラムで、4 倍精度 (real16) のプログラムは約 900 行で、8 倍精度 (real32) のプログラムは、約 1100 行であった。

参考文献

[1] 浜田 穂積, 近似式のプログラミング、培風館, (1995)

[2] 谷川 秀彦, 高精度演算を用いた混合精度反復法, 応用数学会三部会連携「応用数理セミナー」資料集, (2013), 4-35

[3] 平山 弘, C++ 言語による高精度計算パッケージの開発、日本応用数学会論文誌, 5 (1995), 307-318

[4] 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

[5] 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)

[6] Yozo Hida, Xiaoye S. Li, David H. Bailey, Library for Double-Double and Quad-Double Arithmetic, Proc. 15th Symposium on Computer Arithmetic, (2007), 155-162

[7] Yozo Hida, Xiaoye S. Li, David H. Bailey, Algorithms for Quad-Double Precision Floating Point Arithmetic, Lawrence Berkeley National Laboratory, (2000)