Vol.2014-HPC-143 No.22
2014/3/4

C++言語用 double-double 型 4 倍精度演算ライブラリの開発とその応用

平　山　　弘[†1]

倍精度浮動小数点数を 2 個で表現できる 4 倍精度数用の演算ルーチンを作成した。これらの数値の入出力にために C++言語で作成した多倍長演算ルーチンを利用した。このプログラムを使うことによって、通常 4 倍精度を持たない C++言語に 4 倍精度の演算機能を持たせることができる。この演算ルーチンでは、四則演算だけでなく、絶対値、整数部、指数対数関数、三角関数およびその逆関数、双曲関数およびその逆関数を準備した。

このライブラリを利用することによって、既存の C++言語のプログラムを容易に 4 倍精度プログラムに変換することができる。多くのプログラムを高精度で計算出来る。

Double-double Type Quadruple Precision Arithmetic Library for C++ Languages and its Application

Hiroshi Hirayama[†1]

The arithmetic routine for quadruple precision floating point numbers which can consist of two double precision floating point numbers was created. The multiple precision arithmetic routine created by C++ language for the input and output of these numbers was used.

The calculation function of quadruple precision can be given to C++ language which does not usually have quadruple precision by using this program.

By this arithmetic routine, not only addition, subtraction, multiplication and division but absolute value, integer part, 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 quadruple precision program by using this program library. Many programs can be executed in high precision.

1. はじめに

これまでの数値計算の多くは、計算機のハードウエ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要求されてきている。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[5] 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、簡単にできる。

4 倍精度数を持たない多くの C++言語では、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++言語で作成された多倍長演算プログラム[2] を利用して、4 倍精度数の入出力ルーチンを作成し、使いやすい 4 倍精度演算ルーチンを作成することができることを示す。また、それらのプログラムの有用性を示す例を示した。

2. Bailey の double-double アルゴリズム

Bailey の double-double アルゴリズム[3] では，4 倍精度浮動小数点 (quad) の変数 a を二つの倍精変数 a.high(上位データ) および a.low(下位データ) を用いて

$$a = \text{a.high} + \text{a.low} \quad (\frac{1}{2}ulp(\text{a.high}) \geq |\text{a.low}|) \tag{1}$$

と表現し，4 倍精度演算を倍精度変数の演算で実現する方法である．$ulp(x)$ は x の最小ビット (unit in the last place) を意味する。このとき，a.high および a.low 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 52bit であり、2 つの倍精度浮動小数点数

†1 神奈川工科大学
Kanagawa Institute of Technology

を利用することで 104bit 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ため、計算速度はハードウエアで計算をする部分が多い double-double 型 4 倍精度数が数倍速く、実用的な方法である[4)]。仮数部分が 100 ビット程度の浮動小数点すなわち 4 倍精度の浮動小数点の計算では、4 倍精度の浮動小数点数のハードウエアを持たない場合、最も高速の計算法と思われる。

4 倍精度加算演算および乗算を double-double アルゴリズムを利用して計算するためのプログラムをそれぞれ図 1、図 2 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 の積を計算する関数である。プログラムの中にある 134217729.0 は $2^{26}+1$ の数値で、浮動小数点数の仮数部を分離するために使用している数値である。

図 1：4 倍精度加算のプログラム

```
quad add( const quad &a, const quad &b )
{
  quad c ;
  double sh, eh, v, ss ;
  sh = a.high + b.high ;
  v = sh - a.high ;
  eh = (a.high -(sh-v)) + (b.high - v) ;
  eh = eh + a.low + b.low ;
  c.high = sh + eh ;
  c.low = eh-(c.high-sh) ;
  return c ;
}
```

図 2：4 倍精度乗算のプログラム

```
quad mul( const quad & a, const quad &b )
{
  quad c ;
  double u1, p1, p2, r1, r2, s1 ;
  c.high = a.high * b.high ;
  u1 = 134217729.0 * a.high ;
  p1 = u1 - (u1 - a.high) ;
  p2 = a.high - p1 ;
  u1 = 134217729.0 * b.high ;
  r1 = u1 - (u1 - b.high) ;
  r2 = b.high - r1;
  c.low = ((p1*r1-c.high) + p1*r2 + p2*r1) + p2*r2 ;
  c.low = c.low + (a.high * b.low +a.low * b.high) ;
  s1 = c.high +c.low ;
  c.low =c.low - (s1 - c.high) ;
  c.high =s1 ;
  return c ;
}
```

このプログラムでは、double-double 型 4 倍精度数を quad と記述してある。上位の double 型数値を high、下位を low と記述してある。この二つのプログラムで 4 倍精度数 a,b の和と積を計算することができる。

これらプログラムがうまく動作するためには、丸め処理が最近接数への丸めである必要がある。インテル社製の 32 ビット CPU 用のコンパイラーでは、64 ビットの浮動小数点数だけでなく、80 ビットの拡張精度数をサポートしているためか、必ずしも 64 ビット数の演算の丸め処理が最近接数への丸めになっていないので、その場合は、64 ビット数の演算の丸め処理を最近接数への丸めに変更する。

32 ビットコンパイラーで 64 ビット浮動小数点数の演算の丸め処理が最近接数への丸めでない場合、80 ビット浮動小数点数の演算の丸め処理が最近接数への丸めになっている場合が多い。この場合、上のプログラムで、double を 80 ビットの long double 型に変えることによって、160 ビットの浮動小数点数の計算が可能になる。ただし、乗算のプログラムの中にある定数 134217729.0($=2^{27}+1$) は 4294967297.0($=2^{32}+1$) に変更する必要がある。

このように計算された 4 倍精度数の丸め処理は、最近接丸め処理にはなっていない。このため、この 4 倍精度数を二つ組み合わせて、8 倍精度数を定義しても、上記のプログラムを使って 8 倍精度の計算は出来ない。

3. 4 倍精度演算プログラム

3.1 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、そのための入出力ルーチンを準備しなければならない。ここでは、多倍長計算プログラムを利用して、入出力ルーチンを作成した。入力は、文字列として入力し、その文字列を 4 倍精度浮動小数点に変換するプログラムを準備した。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、簡単に入出力できる。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、一旦文字列に変換して入出力することになる。上のように出力を記述すると、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") ;
```

のように書いて変更できる。既定書式でない書式で、出力したい場合には

```
cout << to_string( a, "%40.30e" ) ;
```

という形式で書式を指定して出力できる。to_string 関数を利用して、double-double 型 4 倍精度数を文字列に変換できる。

3.2 4 倍精度関数

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。除算は、逆数が満たす方程式を Newton 法を使って解く。数値 a の逆数が満たす方程式は次のようになる。

$$\frac{1}{x} - a = 0 \tag{2}$$

この方程式を Newton 法を使って解くことによって計算する。この場合の Newton 法の計算は、次の様になる。

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

この計算には除算は含まれないので加減乗算だけで計算できる。このときの Newton 法の初期値には、倍精度で計算した逆数の値を使う。この初期値を求める部分に除算が含まれる。この数値を使えば、1 回の Newton 法の反復計算で 4 倍精度の計算結果が得られる。逆数が得られるので容易に除算が計算できる。

平方根の計算も同様に計算できる。数値 a の平方根の逆数は、次の方程式を満たす。

$$\frac{1}{x^2} - a = 0 \tag{4}$$

この場合の Newton 法の計算は、次のようになる。

$$x_{n+1} = x_n + 0.5(x_n - ax_n^3) \tag{5}$$

逆数の計算と同様に加減乗算だけでできる。2 で割る計算は 0.5 を掛ける計算に変えている。

この 4 倍精度計算プログラムでは、指数対数関数 (exp(x)，log(x)，log10(x)) 三角関数 (sin(x)，cos(x),tan(x))、逆三角関数 (asin(x)，acos(x)，atan(x))、双曲線関数 (sinh(x),cosh(x),tanh(x),asinh(x),acosh(x),atanh(x)) などの数学関数を準備した。整数部 (floor(x)) を取る関数や絶対値 (abs(x)) などの関数を準備した。

一部の数学関数を高速に計算するため、ミニマックス近似（最良近似）等[1] を使うがこれらの近似関数は、一般に不規則な係数を持つ。これらの定数は 4 倍精度の数値なので、プログラムに直接書き込むことが出来ない。たとえば、4 倍精度の変数 pi に円周率を代入するには、次のように書けない。

```
pi = 3.14159265358979323846264338327950288 ;
```

C++言語では、右辺の数値は倍精度数と解釈するので、4 倍精度数の代入が行われないため精度低下が起こる。

```
pi = quad("3.14159265358979323846264338327950288") ;
```

と書けば、文字列を 4 倍精度に変換してくれるので問題は起こらないが、文字列を 4 倍精度数に変換するため、かなりの計算時間を要する。このような方法で係数を与えたのでは、高速計算するためにミニマックス近似式を使っているのに逆に遅くなる結果になる可能性がある。この場合、次ように書くことによって、高速化を維持できる。

```
pi.high = 3.14159265358979310e+000 ;
pi.low  = 1.22464679914735320e-016 ;
```

上の式の右辺の数値は、double-double 型 4 倍精度数の上位と下位の double 型数値を十分な精度で出力したものである。double 型の数値の精度は 10 進数で約 15.3 桁であるから 16 桁以上を指定すれば正確に記述できる。余裕を見ても 17 桁以上を指定すれば倍精度の数値を厳密に表現できる。これは多くのプログラミング言語では、数値文字列で記述した 10 進数は、最も近い数に変換するからである。そのため、次のようなに初期値を容易に与えることができる。

```
quad pi ={ 3.14159265358979310e+000, 1.22464679914735320e-016 } ;
```

このような初期値を与える方法は、4 倍精度の計算プログラムで多数利用した。

4. 4 倍精度ライブラリの使用法

4 倍精度のライブラリの使用は、簡単でできるだけ、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。

前節で述べたように、独自に開発した数をプログラムに書く場合、通常の型のように書くことはできない。数値文字列は、コンパイラはコンパイラーの仕様にある数値型だと仮定してコンパイルするためである。

```
quad e =2.71828182845904523536028747135266249775724 ;
```

と書いても、数値文字列は最も精度の高い double 型の数値と解釈され、15 型程度の精度の数値になる。これを避ける方法は、前節に書いた通りである。

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。図 3 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図 4 に示す。

図 3：倍精度 2 次方程式の解法プログラム

```
#include <iostream>
#include <cmath>
using namespace std ;
int main()
{
    double a, b, c, d, x1, x2 ;
    a=2 ; b = 7.5 ; c=-12.2 ;
    d=b*b-4*a*c ;
    d = sqrt(d) ;
    x1=(-b+d)/(2*a) ;
    x2=(-b-d)/(2*a) ;
    cout << "x1=" << x1 << endl ;
    cout << "   " << a*x1*x1+b*x1+c << endl ;
    cout << "x2=" << x2 << endl ;
    cout << "   " << a*x2*x2+b*x2+c << endl ;
}
```

図 4：4 倍精度 2 次方程式の解法プログラム

```
#include "long_num.h"
int main()
{
    quad a, b, c, d, x1, x2 ;
    a=2 ; b = 7.5 ; c=quad("-12.2") ;
    d=b*b-4*a*c ;
    d = sqrt(d) ;
    x1=(-b+d)/(2*a) ;
    x2=(-b-d)/(2*a) ;
    set_format("%37.32g") ;
    cout << "x1=" << x1 << endl ;
    cout << "   " << a*x1*x1+b*x1+c << endl ;
    cout << "x2=" << x2 << endl ;
    cout << "   " << a*x2*x2+b*x2+c << endl ;
}
```

変更部分は、main 文の前にある宣言部分と double を quad 変更しただけである。インクルードファイル”long_num.h”では 4 倍精度で使うほとんどの宣言なされているので多くの宣言を省略できる。

```
a=2 ;
```

の部分は、通常変更されるが、定数 2 は整数でコンパイルされた後でも厳密な数なので変更しないでも誤差が生じることはない。もし、次のような場合、

```
c=-12.2 ;
```

-12.2 は倍精度数としてコンパイルされるから、4 倍精度の数値としては精度が不十分である。このような場合、

```
a= quad("-12.2") ;
```

と記述すれば、4 倍精度の計算が可能である。これらのプログラムを実行すると、それぞれ以下の図 5、図 6 のように出力される。

図 5：倍精度計算結果

```
x1=1.22591
   -1.77636e-015
x2=-4.97591
   -3.55271e-015
```

図 6：4 倍精度計算結果

```
x1=    1.2259071253425182195488491564024
        1.9721522630525295135293214e-31
x2=   -4.9759071253425182195488491564024
       -1.9721522630525295135293214e-31
```

誤差の値から、それぞれ倍精度、4 倍精度計算であることがわかる。

5. 数値例および応用例

5.1 数 値 積 分

二重指数型数値積分法で次の積分を行った。結果がわかりやすい数値になり、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。

$$I = \int_0^2 \sqrt{4 - x^2} dx = \pi \tag{6}$$

計算結果は 3.1415926535897932384626433832795 となり、誤差 6.3e-27、標本点数 N=117 であった。この積分の被積分関数は四則演算と平方根だけで計算出来るが、計算方法として、二重指数関数を利用しているので双曲線関数を使っている。このようにいろいろな関数を含む計算も容易に計算できる。

5.2 連立一次方程式

計算速度を計るため、次のような 200 元連立一次方程式を解いた。

$$Ax = b \tag{7}$$

行列 A の係数 $a_{i,j}$ として

$$a_{i,j} = \begin{cases} 10 + i & i = j \\ 1 & i \neq j \end{cases} \tag{8}$$

を与えた。b は $b_i = i$ とした。計算法として、linpack(lapack) を元に作成した行列計算テンプレートを利用して計算した。この場合、プログラムは、次のようになる。

```
 1: #include "matrix_template.h"
 2: typedef matrix_template<double> matrix ;
 3: void main()
 4: {
 5:     int n=100 ;
 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 ;
17: }
```

このプログラムを使って倍精度と double-double 型の 4 倍精度で計算した。行列大きさを 100〜2000 にとって計算した。その結果を表 1 に示す。

表 1 倍精度および 4 倍精度による連立一次方程式の計算時間

行列の大きさ	4 倍精度 (msec)	倍精度 (msec)	4 倍精度/倍精度
100	9.88	0.86	11.46
200	76.19	6.10	12.48
300	254.	20.75	12.24
400	594.	48.34	12.29
500	1172.	92.75	12.64
600	2000.	162.12	12.34
700	3156.	258.	12.23
800	4703.	375.	12.54
900	6719.	539.	12.47
1000	9234.	742.	12.44
1100	12329.	1000.	12.33
1200	16111.	1328.	12.13
1300	20282.	1719.	11.80
1400	25329.	2219.	11.41
1500	31282.	2796.	11.19
1600	37907.	3485.	10.88
1700	45922.	4218.	10.89
1800	54267.	5078.	10.69
1900	63845.	6063.	10.53
2000	76064.	7219.	10.54

上のテスト問題のように、テンプレートを使わないで単純に行列の大きさ 200 の問題をガウスの消去法を使うと倍精度では 3.3msec、4 倍精度で 28.4msec の時間を要した。4 倍精度の時間は倍精度の約 8.6 倍であった。上のテンプレート使ったプログラムでは、元の行列を破壊しないようにするため、元の行列を他の行列にコピーしたり、連立一次方程式を解くために逆行列を計算したりしているため、同じ問題を解くために 2 倍以上の時間が必要である。

このとき使用した 64 ビットコンパイラは、Microsoft 社の Visual Studio 2012 C++コンパイラである。最適化オプションとして/Ox を使用した。実行に使用したマシンは、Intel Core i7-3930K CPU 3.2GHz である。

この結果は Fortran で作成した同等のプログラムよりやや高速であった。加減乗算のプログラムがインライン展開されるためと思われる。

多くのサイズの行列に対して、計算を行った。計算結果としては行列の大きさが小さい問題（1000 以下程度）では、4 倍精度は約 12 倍程度高速で、大きな問題（1000 以上程度）では、約 11 倍であった。結果が異なるのは、計算機のキャッシュの容量が関係すると思われるので、

5.3 複素数の計算

C++言語なので、複素数のテンプレートを使えば、複素数計算も容易に行えると期待できる。

5.3.1 代数方程式の複素根の計算

ここでは例として、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。

$$2x^6 + x^4 + 3x^3 + 6x^2 + x + 3 = 0 \tag{9}$$

この代数方程式を初期値 $100 + 20i$ 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。

$$x_{n+1} = x_n - \frac{2x_n^6 + x_n^4 + 3x_n^3 + 6x_n^2 + x_n + 3}{12x_n^5 + 4x_n^3 + 9x_n^2 + 12x_n + 1} \tag{10}$$

収束条件 $|x_{n+1} - x_n| < 1.0^{-25}$ を満たすまで計算した。33 回の反復計算で収束し
$x = 3.43398226413166602886078182862 \times 10^{-2} + 6.95732370761952500713470032782 \times 10^{-1} i$
が得られる。この得られた解を元の式に代入すると $-2.5 \times 10^{-32} - 9.2 \times 10^{-33} i$ となり、4 倍精度で計算されていることがわかる。

C++言語の中にある複素数用テンプレートは、float や double 型など言語仕様にある浮動小数点数で使う場合には、問題なく動作するが、4 倍精度のように言語仕様にない型で利用する場合うまく動作しない部分がある。絶対値を計算するところでオーバーフローを避けるためと推定される処理で、うまく動作しなかった。同じような現象は、2 種類の市販の C++言語コンパイラーで起こった。このため、複素数のテンプレートプログラムを自作し

実行した。上の結果は、このテンプレートを使用した場合の結果である。

5.3.2 超越関数を含む非線形方程式の複素根の計算

超越関数を含む方程式として、次の方程式を考える。

$$e^x - \sin x - 3x = 0 \tag{11}$$

この方程式は、2 個の実数解を持つが、ここでは $6 + 7i$ 付近の複素数解を Newton 法で解く。すなわち、次の反復公式

$$x_{n+1} = x_n - \frac{e^{x_n} - \sin x_n - 3x_n}{e^{x_n} - \cos x_n - 3} \tag{12}$$

を利用して解く。初期値は $6 + 7i$ とする。収束条件 $|x_{n+1} - x_n| < 1.0^{-25}$ を満たすまで計算した。次のように、11 回反復計算をすると表 2 のように収束し解が得られた。

表 2 $(x + \cos x)/(2 + \sin x)$ の原点での Taylor 展開係数

反復回数 (n)	計算結果 (x_n)
0	(6.00000000000000000000000000000, 7.00000000000000000000000000000)
1	(8.06356050949817037176753108876 62, 6.77081624048991473424579652365 86)
2	(7.24956693645301852206494746395 35, 6.77078352966745768431576613051 94)
3	(6.62887708321664572895407338306 02, 7.01318484354660924551712412517 73)
4	(6.65712832478652095352335579862 78, 7.52809862411864988826974230654 871)
5	(6.71801040918902491018356139428 61, 7.39054339363091258791858896651 134)
6	(6.73455140651535635771248785771 76, 7.39256516143329873421401514276 54)
7	(6.73438159294394686811271227403 79, 7.39266287915911336770074505561 06)
8	(6.73438159952738356181767483652 75, 7.39266290560755759441473189717 03)
9	(6.73438159952738372400087624626 53, 7.39266290560755709251375504036 68)
10	(6.73438159952738372400087624626 46, 7.39266290560755709251375504036 76)
11	(6.73438159952738372400087624626 53, 7.39266290560755709251375504036 68)

この結果を検算するため、元の方程式に代入すると $1.5 \times 10^{-28} + 5.6 \times 10^{-28} i$ となり、4 倍精度の解であることがわかる。

5.4 Taylor 展開の計算

C++言語の関数として記述された連続関数は、浮動小数点を係数に持つ Taylor 級数に容易に計算出来る。ここでは簡単な例を計算する。

次の関数 $f(x)$ を $x = 0$ において、4 次まで Taylor 展開する。

$$f(x) = \frac{x + \cos x}{2 + \sin x} \tag{13}$$

これを 10 次まで計算すると、係数は表 3 のようになる。このように容易に計算出来る。

表 3 $e^x - \sin x - 3x = 0$ を Newton 法での計算結果

次数	係数（4 倍精度)
0	0.50000000000000000000000000000000
1	0.25000000000000000000000000000000
2	-0.37500000000000000000000000000000
3	0.22916666666666666666666666666666
4	-0.07291666666666666666666666666666
5	0.00312500000000000000000000000000
6	0.01579861111111111111111111111111
7	-0.01236359126984126984126984126984
8	0.00552455357142857142857142857142
9	-0.00117979772927689594356261022920
10	-0.00043116870590828924162257495590

6. ま と め

倍精度数を二つ組み合わせた 4 倍精度演算のプログラムを作成した。その性能は、連立方程式の解法で倍精度計算時間の約 10 倍程度の時間を要するが、整数演算を使いソフトウエアで実現された IEEE の 4 倍精度に比べ、計算が速く使いやすいプログラムとなった。

この計算法は 80 ビットの精度を持つ拡張倍精度にも適用できる。この場合、指数部は IEEE の 4 倍精度と同じになり、指数部は同じで、精度は IEEE の 4 倍精度より 16 ビット多い 4 倍精度が作れる。しかし、最近の C++言語のコンパイラーは拡張倍精度をサポートしていないのが残念である。

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) 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
3) 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
4) 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)
5) Yozo Hida, Xiaoye S. Li, David H. Bailey, Library for Double-Double and Quad-

Double Arithmetic, Proc. 15th Symposium on Computer Algorithmetic,155-162