Vol.2021-HPC-179 No.6
2021/5/14

4倍精度および8倍精度演算ライブラリの開発とその性能評価

平山 弘[1,a)]　小宮 聖司[1,b)]

概要：倍精度浮動小数点数を 2 個および 4 個で表現できる 4 倍精度および 8 倍精度浮動小数点数用の演算ライブラリーとその複素数演算ライブラリを作成した。これらの数値を C++言語で入出力するための補助プログラムも準備した。このライブラリを使うことによって、通常 4 倍精度や 8 倍精度を持たない C++言語に 4 倍精度や 8 倍精度の演算機能を持たせることができるだけでなく、4 倍精度と 8 倍精度の複素数演算を可能にする。これらのライブラリを使って、連立一次方程式の計算と基数 8 の実数フーリエ変換のプログラムを実行しその性能を評価した。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 倍精度や 8 倍精度の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。

Development of Double-double and Double-double Precision arithmetic libraries and their Performance Evaluation

Hiroshi Hirayama[1,a)]　Komiya Seiji[1,b)]

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 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 library. By using this library, not only can C ++ languages that normally do not have double-double or quad-double precision have double-double or quad-double arithmetic functions, but also double-double and quad-complex complex arithmetic operations. Using these libraries, we ran programs of simultaneous linear equations and real Fourier transform of radix 8 to evaluate these performance. By using this library, existing C ++ language programs can be easily converted to double-double or quad-double precision programs. Many programs can be calculated with high accuracy.

Keywords: Quad precision complex number, C++ Programing Language

1. はじめに

現在の計算機の演算装置部部分は、非常に高性能になっているが、小型演算装置 (マイクロプロセッサ) を起源とするためか 4 倍精度 (128bit) の浮動小数点演算をハードウエアで実行できるものが一部を除いて、ほとんどないのが現状である。演算装置の高性能化と計算速度の増加に伴い、大量の計算を行うため丸め誤差が大きくなり、そう少し高い精度の計算が簡単にできる環境が望まれて来ている。汎用計算機と呼ばれた初期の計算機の多くが 4 倍精度の演算機能をハードウエアで実行できたことを考えると、ベクトル機能やマルチコアなどが実装できるのに、なぜ現在の計算機が 4 倍精度の計算ができない疑問を感じる。

このような状況をある程度克服するため、Bailey[7][8] に

[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)] kom@eng.kanagawa-it.ac.jp

よって提案されている倍精度を二つ組み合わせた 4 倍精度数の演算プログラムを作成した。さらに 4 つ組み合わせた 8 倍精度数の演算プログラムも作成した。四則演算だけでなく、平方根、指数対数関数、三角関数等を作成しライブラリ化した。これらの関数の複素数用の関数も作成した。

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プログラムは、かなり長いプログラムになるため作成が難しいものとなる。IEEE754-2008 の 4 倍精度を持つプログラム言語では、プログラム言語のこの 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 C++言語等では、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。

これらの数値の加減乗除算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ため、この高精度数の利用が加速した。

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した。また、それらのプログラムの有用性および性能を調べた。

ここで取り上げた計算例の計算時間は、Intel i9-10900K CPU 3,7GHz で実行に要した時間である。

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。[6]。

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 $\frac{1}{2}\mathrm{ulp(s)} \geq |\mathrm{e}|$ 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同様の加算できる。

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 ) ;
}
```

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。

C99 言語で定義されている fma(fused multiply add) 関数を使用すると、二つの倍精度数の乗算は簡単に書くことができる。fma(a,b,c)=a*b+c と定義される関数である。この関数では a*b の計算は 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 3：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 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と期待できる。Intel 社の CPU では、$\pm a * b \pm c$ と表される 4 種の計算できるので、符号を合う命令を使えば少し高速化ができる。これを利用するには Intel 社に依存するコードを書かなければならないので、ここでは、プログラム言語で定義されている fma だけを利用した。これによってすべての C++言語によって動作可能なプログラムになる。

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。そのプログラムをプログラム 4、プログラム 5 に示す。これから加算、乗算の演算数はそれぞれ 11flops，12flops 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 4：4 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 5：4 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

4 倍精度型、8 倍精度型の高精度数の定数をプログラムの中に書くには、倍精度数の場合と同様に次のように書くことができる。

```
s=0.5*a*h ;
t=3.45*r*r*r ; // 定数を real16("3.45") とする。
s=real16::M_PI*r*r;//real16::M_PI はπの高精度数
```

プログラム内では、上の式は次のように解釈される。

```
s=real16(0.5)*a*h ;
t=real16(3.45)*r*r*r;//定数を real16("3.45")
                      とする。
s=real16::M_PI*r*r;//real16::M_PI はπの高精度数
```

最初の式の 0.5 は高精度数に変換しても誤差なしで変換できるので問題が怒らない。2 番目の式の 3.45 は約 15 桁精度の倍精度数と解釈されそれを高精度数に変換します。このため誤差が発生しますので、円周率などの一部の定数はプログラム言語ですでに定義されている。例えば円周率は高精度定数 real16::M_PI を使えば約 32 桁の高精度を指定できる。

IEEE 型の 4 倍精度の浮動小数点数を持つ処理系では、この IEEE 型の 4 倍精度を使って入出力が行えるので、4 倍精度数の入出力が容易にできるが、残念ながら、8 倍精度数の入出力はできない。

多くの C++言語のコンパイラーは、4 倍精度の浮動小数点を持っていないため、4 倍精度数用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 (string) 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。高精度数は、書式を指定して文字列に変換する関数を準備している。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が、文字列をを通して出来るようになっている。

出力の場合は、書式に従って文字列に変換し、その文字列を出力することになる。入力の場合は、文字列として入力し、それを高精度数に変換する。

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("%30.25f") ;
```

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

```
string s = to_string( a, "%30.25e" ) ;
cout << s ;
```

という形式で書式を指定して文字列に変換し、それを出力する。規定書式を変更するには、次のように set_format 関数を使う。

```
set_format("%30.25e" ) ;
set_format("%30.25e", 2 ) ;
```

最後の数は、指数部の桁数である。省略すると 3 桁となる。

2.1.1 複素数の入出力

複素数の定数は、プログラムの中では次のように書くことができる。complex16、complex32 はそれぞれ 4 倍精度複素数、8 倍精度複素数を意味する。

```
complex16 a=complex16( 3, 4 ) ;
                // 4 倍精度の 3+4i を a に代入
complex32 b=complexl32( 3.5, 4.56 ) ;
                // 8 倍精度の 3.5+4.56i を b に代入
```

同じことを次のように書くこともできる。

```
complex16 a=complex16("(3,4)" ) ;
                // 4 倍精度の 3+4i を a に代入
complex16 a=complex16("3+4i" ) ;
                // 4 倍精度の 3+4i を a に代入
complex32 b=complexl32( "(3.5, 4.56)" ) ;
                // 8 倍精度の 3.5+4.56i を b に代入
complex32 b=complexl32( "3.5+4.56i" ) ;
                // 8 倍精度の 3.5+4.56i を b に代入
```

最後の二つはどちらも 3.5+4.56i を 8 倍精度複素数 b に代入する。上の例はすべて文字列から変換されるため、精度がよくなります。

複素数の入力では、二つの形式で入力できる。(a,b) と a+bi 形式である。すなわち (3.45,-6.78) や 1.23e45-7.8e-9i などが直接入力できる。

```
complex16 s ;
cin >> s ;
```

出力は、実部と虚部を既定書式で (a,b) 形式で出力する。a+bi 形式で出力するには、to_abi(x,form) として、文字列に変換してから出力します。

2.2 4 倍・8 倍精度用関数

4 倍精度・8 倍精度計算ライブラリでは、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 C++言語で準備されている数学関数を実数および複素数で計算するものを準備している。floor 関数や絶対値などの関数も準備した。

これらの関数は一部最良近似式 [1] を使っているが、主に Taylor 展開などの式を利用して計算している。

また、数学定数を 17 個準備した。以下の表 1 にそれを示す。この定数は、real16::M_PI と書くと円周率 32 桁書いたものと同じになる。real32::M_PI と書けば 64 桁の円周率となる。

これらの定数は、三角関数や指数対数関数の計算にも使われている。

表 1 数学定数

円周率関係		
M_PI	π	3.14159265358$\cdots$
M_2PI	2π	6.28318530717$\cdots$
M_PI_2	$\frac{\pi}{2}$	1.57079632679$\cdots$
M_PI_4	$\frac{\pi}{4}$	0.78539816339$\cdots$
M_1_PI	$\frac{1}{\pi}$	0.31830988618$\cdots$
M_2_PI	$\frac{2}{\pi}$	0.63661977236$\cdots$
M_1_2PI	$\frac{1}{2\pi}$	0.1591549430$\cdots$
M_1_SQRTPI	$\frac{1}{\sqrt{\pi}}$	0.5641895835$\cdots$
M_2_SQRTPI	$\frac{2}{\sqrt{\pi}}$	1.1283791670$\cdots$
指数・対数関係		
M_E	e	2.7182818284$\cdots$
M_LOG2E	$\log_2 e$	1.4426950408$\cdots$
M_LOG10E	$\log_{10} e$	0.4342944819$\cdots$
M_LN2	$\log_e 2$	0.6931471805$\cdots$
M_LN10	$\log_e 10$	2.3025850929$\cdots$
その他		
M_SQRT2	$\sqrt{2}$	1.4142135623$\cdots$
M_SQRT1_2	$\frac{1}{\sqrt{2}}$	0.7071067811$\cdots$
M_GAM	$\gamma = \lim_{n=\infty} \sum_{k=1}^{n} \frac{1}{k} - \log k$	0.5772156649$\cdots$

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。複素数の場合、4 倍精度数の名前を complex16、8 倍精度数の名前を complex32 としている。

C++言語では、これらの名前は容易に名前を変えることができるので適切な名前に変えて使用することを想定している。たとえば、次のように書けば、real16 を quad に変換できる。

```
typedef complex16 quad ;
```

高精度数値の定数を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定数（数値文字列）は、コンパイラによって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、数値文字列はコンパイラーの仕様で最も精度の高い double 型の数値と解釈され、約 15 桁程度の精度の数値に丸められる。その丸められた数値が 4 倍精度数に代入されるため精度が約 15 桁になる。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 pq=real16(3.1415926535897931e+00,1.2246467991473532e-16) ;
real32 po=real32(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 15 桁の精度であるから、16 桁以上指定すれば、正確な値を設定できる。このため、ここでは 17 桁指定している。

VS2019 C++言語では、上の定数を次のようにも書くことができる。

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

使用例として、2 次方程式 $(2.3+1.6i)x^2+(5.1-4.3i)x+(2.7+1.4i)=0$ を解くプログラムを 4 倍精度のプログラムに変換する。プログラム 6 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 7 に示す。また、8 倍精度で計算した結果も示す。

プログラム 6：倍精度複素 2 次方程式の解法

```
 1: #include <iostream>
 2: #include <cmath>
 3: #include <complex>
 4: using namespace std ;
 5: typedef complex<double> dcomplex ;
 6: int main()
 7: {
 8:   dcomplex a, b, c, d, x1, x2 ;
 9:   a=dcomplex(2.3,1.6) ;
10:   b = dcomplex(5.1,-4.3) ;
11:   c=dcomplex(2.7,1.4) ;
12:   d=b*b-4.0*a*c ; d = sqrt(d) ;
13:   x1=(-b+d)/(2.0*a) ; x2=(-b-d)/(2.0*a) ;
14:   cout << "x1=" << x1 << endl ;
15:   cout << "   " << a*x1*x1+b*x1+c << endl ;
16:   cout << "x2=" << x2 << endl ;
17:   cout << "   " << a*x2*x2+b*x2+c << endl ;
18: }
```

プログラム 7：4 倍精度 2 次方程式の解法

```
 1: #include <iostream>
 2: #include "real1632.h"
 3: int main()
 4: {
 5:   complex16 a, b, c, d, x1, x2 ;
 6:   a=complex16("2.3+1.6i") ;
 7:   b = complex16("5.1-4.3i") ;
 8:   c=complex16("2.7+1.4i") ;
 9:   d=b*b-4.0*a*c ; d = sqrt(d) ;
10:   x1=(-b+d)/(2.0*a) ; x2=(-b-d)/(2.0*a) ;
11:   cout << "x1=(" << x1.re << endl ;
12:   cout << "    " << x1.im << ")" << endl ;
13:   d = a*x1*x1+b*x1+c ;
14:   cout << "   " << to_abi(d,"%11.3e") << endl ;
15:   cout << "x2=(" << x2.re << "," << endl ;
16:   cout << "    " << x2.im << ")" << endl ;
17:   d = a*x2*x2+b*x2+c ;
18:   cout << "   " << to_abi(d,"%11.3e") << endl ;
19: }
```

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、dcomplex を complex16 に変更し、定数を complex16 関数を使う形に変更した。インクルードファイル”real1632.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。VS2019 C++の complex のテンプレートは、double と float 型の専用になっているため

```
complex<real16> a, b, c ;
complex<real32> p, q, r ;
```

のようにして、4 倍精度の複素数は利用できない。gnu C++言語の complex のテンプレートは 4 倍精度や 8 倍精度数に対しても利用できるため、4 倍精度や 8 倍精度の複素数が容易に定義でき、複素数の計算ができる。

複素数の定数は、次のように使われる。

```
a=dcomplex(2.3,1.6);
b=dcomplex(5.1,-4.3);
c=dcomple(2.7,1.4);
```

これを単純に次のように変換する事によって 4 倍精度の複素数定数にすることができる。

```
a=complex16(2.3,1.6);
b=complex16(5.1,-4.3);
c=comple16(2.7,1.4);
```

このように変換すると、数値はすべて倍精度の数値になり、計算が 4 倍精度で計算しても結果は倍精度の結果になる。この場合次のように文字列で与えることによって、4 倍精度の複素数を与えることができる。

```
a=complex16("2.3+1.6");
b=complex16("5.1-4.3i");
c=comple16("2.7+1.4i");
```

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 倍精度複素数に変換されるので、4 倍精度の精度を持つ数値になる。このように書くことによって、4 倍精度の数値を変数に設定できる。これらの 2 次方程式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表示している。倍精度計算結果

```
 x1=(-0.122448,-0.379221)
    (8.88178e-16,-4.44089e-16)
 x2=(-0.495386,2.67858)
    (8.88178e-16,-4.44089e-16)
```

4 倍精度計算結果

```
 x1=(     -0.12244828878754455157355041388783
          -0.37922054713691835626636215328856)
    ( 9.861e-032, 2.465e-031)
 x2=(     -0.49538610611691404715256423579371,
    (       2.67858360446175912059757234437138378904466486757 15)
    (-4.930e-032,-4.684e-031)
```

代入結果を見ると、倍精度では約 16 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 complex16 を complex32 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 の実部=  -0.12244828878754455157355041388781543816744592021778
x1 の虚部=  -0.37922054713691835626636215328858124127396423062881
誤差= 4.786e-063-2.089e-063i
x2 の実部=  -0.49538610611691404715256423579371322425293624538732
x2 の虚部=    2.67858360446175912059757234437138378904466486757 15
誤差= 3.342e-063-2.355e-063i
```

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。

3.1 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。Visual Studio 2019 の C++言語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 $a+bi$ 対して、単純に $\sqrt{a^2+b^2}$ を計算することはない。この場合、$|a|$ と $|b|$ の大きい数値を a とすると、$|a|\sqrt{1+(\frac{b}{a})^2}$ と計算する。ユーザー定義の数値を long double に変換して、いろいろな計算を行っている。ユーザー定義の数値は long double には変換できない場合がある。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 double 型と float 型以外ではほぼ使えないことになる。

4 倍精度数や 8 倍精度の複素数をテンプレート使わないでこれらの複素数のプログラムを作成し利用できるようにした。

この複素数を使って、次の非線形方程式の複素根を計算する。この方程式は、実根 $(0.739085133\cdots)$ を持つが初期値選ぶことによって実根に収束しないようにした。

$$\cos(x) - x = 0 \tag{2}$$

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。

$$x_{n+1} = x_n + \frac{\cos(x) - x}{\sin(x) + 1} \tag{3}$$

収束条件 $|x_{n+1} - x_n| < 1.0^{-25}$ を満たすまで計算した。14 回の反復計算で収束し

$$x = -9.109987453936563006799219722885 6 + 2.950170861699436957762524869506 6i$$

が得られる。得られた複素数解を元の式に代入すると $1.48 \times 10^{-31} + 6.16 \times 10^{-31} i$ となり、この数値が解であることがわかる。誤差の値から 4 倍精度で計算されていることもわかる。

3.2 代数方程式の数値解法

次の代数方程式を解く。

$$f(x) = 6x^8+7x^7+3x^6-20x^5-19x^4-7x^3-37x^2+3x-17 = 0$$

この方程式は、8 次方程式なので、解の公式は存在しないので、数値的方法で解く必要がある問題である。

DK（(Durand-Kerner) 法を使って解く。この方法は、複素根を含め同時にすべての解を計算する方法である。1 個しか求める必要がない場合には無駄な計算が含まれることになる。

すべての根を含む半径 R の円周上に、等間隔に n 個の根の初期値 (推定値) を設定する。n は方程式の次数で、この例では 8 次の方程式なので 8 となる。

$$x_j^{(0)} = Re^{\frac{\pi(4j+1)}{2n}i} \quad (j = 0, \cdots, n-1) \tag{4}$$

この設定した値から出発し、$k = 0$ として、k を増加させて、x_i が収束するまで計算する。

$$x_i^{(k+1)} = x_i^{(k)} - \frac{f(x_i^{(k)})}{a_n \sum_{l=1}^{n} \prod_{\substack{j=1 \\ l \neq j}}^{n} \left(x_i^{(k)} - x_j^{(k)}\right)} \tag{5}$$

これを要求精度 10^{-28} で計算すると 22 回の反復計算で収束し、次のような結果となった。

```
1:( 1.57587929845062487752336861149 19e+00,
    1.85461658637746355102877714647 77e-33)
2:( 3.70381393876781583323416258490 75e-01,
    7.50028571009596974294700520456 75e-01)
3:(-9.94433708220688902354805553709 28e-01,
    1.41775609118971487671671924810 24e+00)
4:(-4.35318858666415906472610126644 81e-02,
    7.79036996245160825477933972714 85e-01)
5:(-1.40737756469619372483273466239 25e+00,
    1.83126993127121863395564315619 52e-33)
6:(-9.94433708220688902354805553709 27e-01,
   -1.41775609118971487671671924810 24e+00)
7:(-4.35318858666415906472610126644 72e-02,
   -7.79036996245160825477933972714 84e-01)
8:( 3.70381393876781583323416258490 76e-01,
   -7.50028571009596974294700520456 76e-01)
```

この解を代入すると、次のような値になる。この解が方程式の解であることがわかる。

```
1: 3.74708929979980607570571068509 32e-30
2: 6.57794824423833931798485466644 26e-31
3: 6.61847365178100885049015575162 39e-30
4: 2.55202578045903831972184295531 72e-31
5: 5.91645678915758854058796423962 09e-31
6: 1.73753777549902579118557910484 06e-29
7: 3.14796887387667013444837379279 11e-31
8: 3.55534805372912035696899512728 16e-31
```

4. 高精度数による性能評価

ここでは、N 元連立一次方程式の解を求める計算および実 FFT の性能評価を行った。

4.1 N 元連立一次方程式

プログラム言語が提供するアセンブラーで記述された IEEE 型の 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で、無料で使える 64 ビットの gnu C++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=16、32、64、128、256、512、1024、2048、4096 の場合を計算した。その結果を表 2 に示す。

$$AX = B \tag{6}$$

行列 A の係数 $a_{i,j}$ と右辺の b_i を次のように設定した。

もし $i \neq j$ のとき、$A_{i,j} = \mathrm{mod}(i+j-1, 7)+1$, $i = j$ のとき、$A_{i,j} = i + 10$ とした。$B_i = i + 20$ とした。ここで、$\mathrm{mod}(x, y)$ は C 言語で x % y と同じことで、x を y で割りその余りを求める関数です。上の問題を解いて、その解を代入し B との差を求め、その絶対値の最大値を求めた。

計算は、A と B の係数を設定して、解を得るまでの時間を何回か反復計算させ計算時間が 1 秒以上になるようにしてその実行時間を測定した。計算時間は測定時間を反復回数で割って求めたものである。計算機は Intel Core i9 10900K 3.7GHz, コンパイラーは tdm64-gcc-9.2.0 の C++を利用した。OS は Windos 10 である。次のようなコンパイラオプションでコンパイルした。以下に、prog.cpp をこのオプションでコンパイルするときのコマンドを示す。

```
c:\>c++ -O3 -mavx2 -mfma -std=c++11 prog.cpp
    real1632.cpp -lquadmath
```

real1632.cpp は高精度計算ライブラリのソースプログラムである。その結果を表 2 に示す。N は行列の大きさ

ここで、double と d は倍精度型、real16 と r16 は 2 個の double 型で表現する 4 倍精度、float128 と f128 はコンパイラーが提供する IEEE754-2008 の 4 倍精度、real32 と r32 は 4 個の倍精度数

表 2 4 倍精度および 8 倍精度で N 元連立一次方程式の計算時間 (秒)

	double	real16		float128		real32	
N	時間	時間	r16/d	時間	f128/d	時間	r32/d
16	7.61E-07	5.53E-06	7.26	5.53E-05	72.66	9.05E-05	118.92
32	5.79E-06	3.83E-05	6.62	3.96E-04	68.34	6.57E-04	113.47
64	4.24E-05	3.02E-04	7.11	2.86E-03	67.35	5.09E-03	119.90
128	1.34E-03	2.51E-03	1.87	2.19E-02	16.34	4.37E-02	32.55
256	1.82E-02	3.47E-02	1.90	2.30E-01	12.63	3.56E-01	19.56
512	1.66E-01	2.76E-01	1.66	1.91E+00	11.51	2.85E+00	17.12
1024	1.53E+00	3.33E+00	2.18	1.73E+01	11.34	4.95E+01	32.43
2048	1.70E+01	8.97E+01	5.29	2.11E+02	12.46	5.96E+02	35.15
4096	1.84E+02	7.97E+02	4.32	2.19E+03	11.87	4.97E+03	26.92
8192	4.64e+03	9.49E+02	2.05	-	-	-	-

で表現する 8 倍精度数を示す。r16/d は (r16 の実行時間)/(d の実行時間)、f128/d は (f128 の実行時間)/(d の実行時間)、r32/d は (r32 の実行時間)/(d の実行時間) である。

real16 の 4 倍精度の計算では、N が 100 以下の小さな問題では、倍精度計算の 6〜7 倍の時間が掛かるが、N=1500 あたりまでは倍精度計算の 2〜3 倍とほぼハードウエア並みの速い速度で計算できる。これ以降は 2〜5 倍の時間がかかる。非常に大きな行列の場合は 2 倍程度で計算できるようである。

4.1.1 基数 8 の実数高速フーリエ変換 (FFT)

ここでは、基数 8 の実数 FFT の計算時間を double と real16 で測定した。プログラムは、ネット上で公開されている基数 8 の実数 FFT[5] を使用したこのプログラムは標本点数を int 型で指定しているので 2147483647($2^3 1-1$) まで指定できるようになっていいる。ここでは 4294967296($2^3 2$) 点まで計算できるようにするため、int 型を long long int 型に変更した。4 倍精度のプログラムに対しては、double 型を real16 型に変更し、定数を 4 倍精度数に変更した。

標本点数は、4=2^2 から 2 倍づつ、4294967296=$2^3 2$(4G) まで計算した。8 倍精度で最大の標本数の実 FFT を実行に必要なメモリは 64G バイトであった。

計算機およびコンパイラーは連立一次方程式の計算と同じものを使用した。

その計算結果を表 3 に示す。倍率とは、double 型の時間を 1 としたときの 4 倍精度 real16 型の時間の割合である。

FFT は、メモリアクセスがランダムに近いためか、データ数による double 型の計算時間と real16 型の計算時間の割合が約 5〜9 となり、連立一次方程式の計算と比較し、変化が少ない。

4.2 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。このプログラムは四則演算に関して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の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、12345^{12345} は 50510 桁の数値になることがわかる。現在、平方根、指数対数関数は完成しているが、三角関数は出来ていない。sin(1.2345e12345) を考えると、1.2345×10^{12345} を 2π で割り、余りを求めその sin 関数を計算するが、あまり意味がないように思えるからである。

5. まとめ

仮数部を 2 倍、4 倍にする 4 倍精度数と 8 倍精度数の計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。このプログラムを使えば、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、高精度複素数も作成した。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った。このプログラムを使って、連立一次方程式と基数 8 の実 FFT の性能を調べた。連立一次方程式は、元数によって、倍精度と高精度の性能差は大きく違う。実 FFT は倍精度と高

表 3 4 倍精度および 8 倍精度で N 元連立一次方程式の計算時間 (秒)

標本点数 N	double		real16		
	時間	誤差	時間	誤差	倍率
4	1.31E-08	0.00E+00	6.84E-08	0.00E+00	5.20
8	1.61E-07	0.00E+00	8.59E-07	8.72E-33	5.32
16	1.95E-07	1.78E-15	1.56E-06	6.47E-32	8.00
32	4.27E-07	1.78E-15	2.87E-06	4.62E-32	6.72
64	7.86E-07	3.55E-15	5.97E-06	8.63E-31	7.59
128	1.61E-06	4.00E-15	1.29E-05	4.99E-31	8.03
256	3.48E-06	7.11E-15	2.74E-05	1.46E-30	7.89
512	7.12E-06	7.11E-15	5.93E-05	1.35E-30	8.33
1,024	1.44E-05	8.33E-15	1.25E-04	2.03E-30	8.70
2,048	3.06E-05	1.07E-14	2.65E-04	1.92E-30	8.68
4,096	6.19E-05	1.24E-14	5.59E-04	1.80E-30	9.03
8,192	1.46E-04	1.07E-14	1.18E-03	2.23E-30	8.10
16,384	2.91E-04	1.42E-14	2.44E-03	2.13E-30	8.38
32,768	6.28E-04	1.42E-14	5.19E-03	2.88E-30	8.26
65,536	1.44E-03	1.78E-14	1.08E-02	2.61E-30	7.51
131,072	3.05E-03	1.78E-14	2.22E-02	3.09E-30	7.29
262,144	5.90E-03	2.75E-14	4.59E-02	3.66E-30	7.79
524,288	1.28E-02	2.34E-14	9.64E-02	3.53E-30	7.55
1,048,576	2.46E-02	1.95E-14	2.00E-01	3.67E-30	8.12
2,097,152	5.89E-02	2.49E-14	4.30E-01	4.18E-30	7.30
4,194,304	1.28E-01	2.35E-14	8.90E-01	3.96E-30	6.95
8,388,608	3.15E-01	2.13E-14	1.88E+00	4.93E-30	5.96
16,777,216	6.27E-01	3.55E-14	3.83E+00	5.03E-30	6.11
33,554,432	1.47E+00	2.84E-14	8.03E+00	4.81E-30	5.48
67,108,864	2.79E+00	3.20E-14	1.68E+01	5.40E-30	6.00
134,217,728	6.59E+00	3.29E-14	3.50E+01	5.51E-30	5.31
268,435,456	1.33E+01	3.20E-14	7.14E+01	4.57E-30	5.38
536,870,912	2.85E+01	3.55E-14	1.47E+02	6.65E-30	5.14
1,073,741,824	5.68E+01	3.64E-14	2.95E+02	5.94E-30	5.20
2,147,483,648	1.18E+02	3.91E-14	6.19E+02	6.62E-30	5.26
4,294,967,296	2.38E+02	3.91E-14	1.26E+03	7.31E-30	5.29

精度数の性能は大きく違うが標本点数による違いは少ない。

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] 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
[3] 平山 弘, C++ 言語による高精度計算パッケージの開発、日本応用数理学会論文誌, 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] 大 浦,Ooura's Mathematical Software Packages, https://www.kurims.kyoto-u.ac.jp/ ooura/fft-j.html
[6] 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)
[7] 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
[8] Yozo Hida, Xiaoye S. Li, David H. Bailey, Algorithms for Quad-Double Precision Floating Point Arithmetic, Lawrence Berkeley National Laboratory, (2000)