多倍長数値計算についての考察
— GNU MP と BNCpack —

静岡理工科大学　幸谷智紀
tkouya@cs.sist.ac.jp

2002 年 9 月 20 日 (金)
Linux Conference 2002

1　初めに

本稿では，仮数部の桁数 (bit 数) が可変の浮動小数点数を用いた数値計算 (以下，多倍長数値計算と略記) について議論することを目的とする。

現在の PC や WS では CPU が直接処理することの出来る，IEEE754 standard の定める浮動小数点数を用いて数値計算を行うことが主流である。これは単精度 (仮数部 24bit)，倍精度 (仮数部 53bit) といった大まかな区分があるだけの，仮数部の桁数が固定された浮動小数点数しか使用していない。高速な演算が可能である一方，数値を入力する際に発生する 10 進→ 2 進変換誤差や，演算中に発生する丸め誤差に対して敏感な問題，もしくはアルゴリズムに対しては精度が不足することもある。そのような，いわゆる悪条件問題，もしくはアルゴリズムを「そのまま」適用する際にはより多くの桁数を確保出来る多倍長計算を行って，変換誤差や丸め誤差を小さく抑える必要が出てくる。

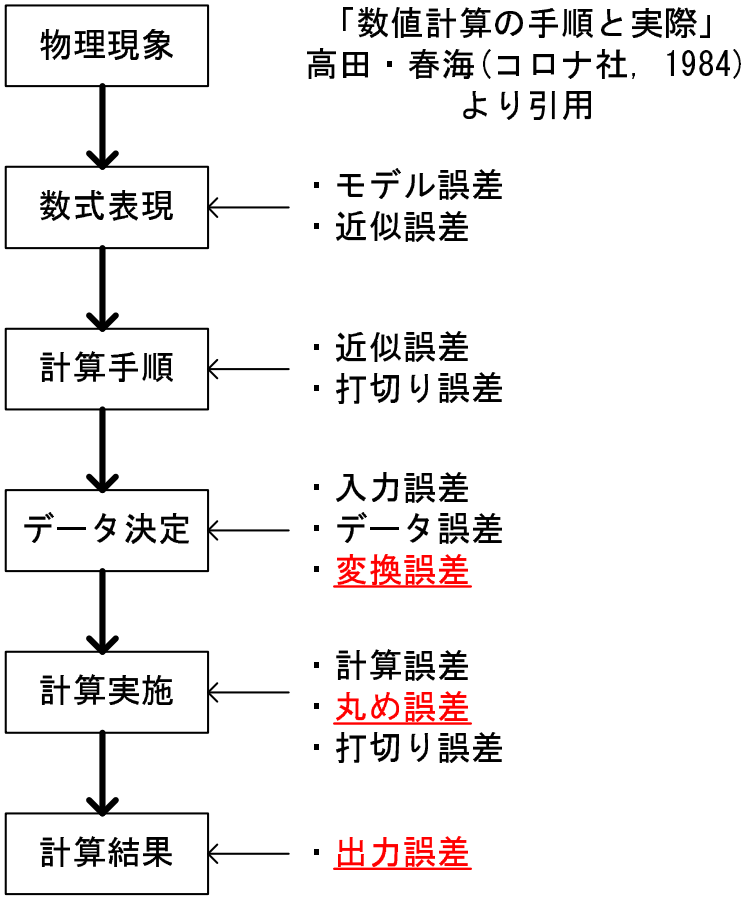


図 1: 数値計算における誤差

多倍長計算はソフトウェアで実装する他なく，IEEE754 倍精度の計算であれば一つの機械語命令で済む所でも，多数の命令を積み重ねて実現する必要が出てくる上，より多くのメモリ領域を確保しなければならない。そのため，多倍長計算はより多くの計算時間を必要とし，数値計算及び数値解析の専門家の間では，それに頼らずに現行の IEEE754 倍精度程度の桁数でも安定かつ高速な計算が可能にするための研究が王道となっている。

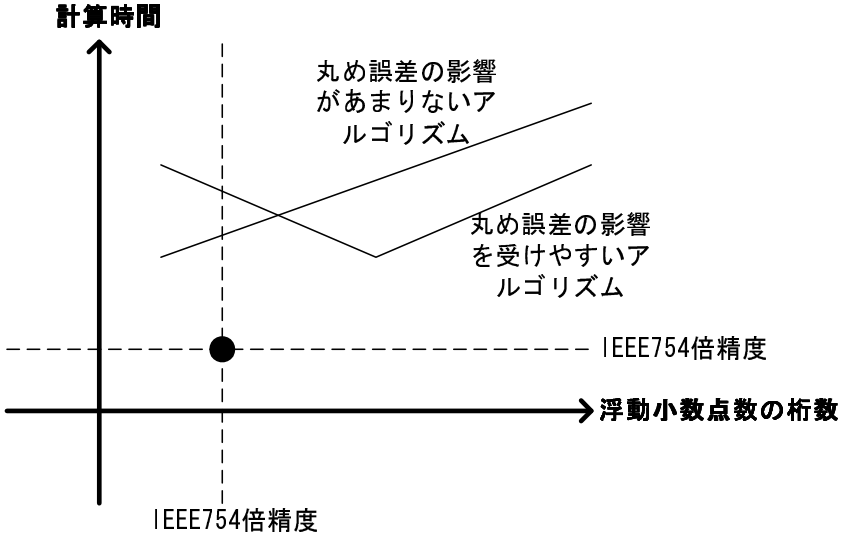


図 2: 浮動小数点数の桁数と計算時間との関係

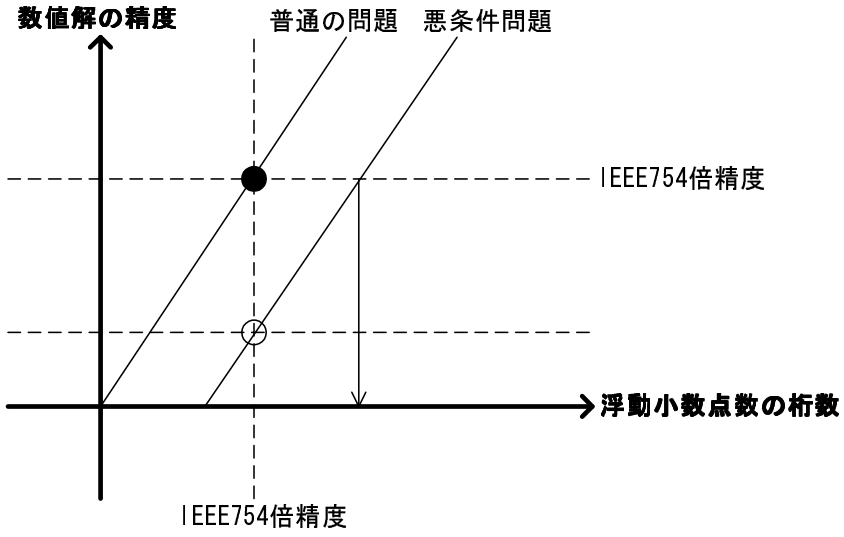


図 3: 浮動小数点数の桁数と数値解の精度との関係

しかし，多倍長計算が実行可能な環境は増えているし，一昔前のように全ての基本演算を自分で実装する必要はなくなってきている。PC の能力は格段に上がり，計算時間とメモリの問題も，大規模問題でなければそこそこ満足出来るレベルに落ち着きつつあると，私は判断している。

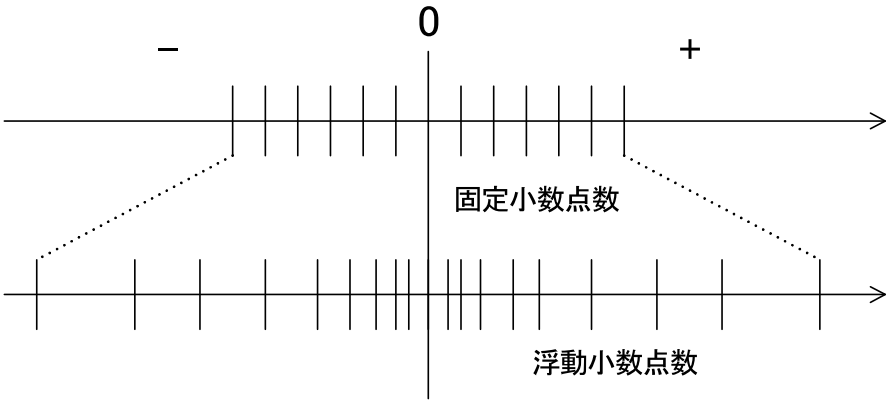
何よりも，桁数を増やすことで解決出来るのであれば，それは問題やアルゴリズムをひねり回す必要がなくなるという点で，ユーザーユレンドリーであると言える。言い方を変えれば「頭の悪い」解決策でもあるのだが，これだけ間口の広がった多倍長計算という存在を頭から無視するのも奇妙に思える。むしろ,「精度は高いが計算時間が遅い」多倍長計算と,「精度は低いが高速な」IEEE754 倍精度計算とを，数値計算ユーザの環境に配慮しつつ，きちんと比較，議論し，数値計算の多様性を広げることが望まれる。

以上のような観点に立ち，本稿ではまず，多倍長計算が必要である事由をきちんと述べることからはじめたい。多倍長計算は数値計算にまつわる全ての厄介事を解決できる万能薬ではないことを認識した上で，効果をもたらす具体的な問題・アルゴリズム例を提示する。その後，私が使用出来た範囲で，多倍長計算可能な open source software や proprietary software の環境の比較を行う。私自身は多倍長計算の一ユーザであって，深い知識は有していないが，どうやら現状では四則演算レベルの高速化は限界に近い，と感じている。その判断を下したのは，今のところ最高に近い性能をたたき出している GNU MP[6](以下，GMP と略記) の実装にある。その事についても若干触れる。最後に，GMP を利用した数値計算ライブラリの例として BNCpack[5] を取り上げ，その性能評価及び問題点を述べる。

2　何故，多倍長計算が必要なのか？

現在の計算機では，整数 (integer) 及び浮動小数点数 (floating-point number) の演算を CPU で直接処理できる。が，計算機のメモリは有限であるから，どちらも処理できるデータの長さ (bit 長) は固定されているのが普通である。一般に，実行される演算が多ければ多いほど，必要な桁数は増えていく。そのため整数型の bit 長が N であれば，絶対値が 2^{N+1} を超えるような整数を扱うことはできず，桁あふれ (overflow) を起こしてしまう。二つの整数型を組にしてそれぞれ分子と分母に割り当てれば有理数を表現することも可能であるが，これも事情は同じである。必要に応じて通分を行うにしても桁あふれを起こしやすいという事情は変わらない。

実数を無限小数と見立て，小数部分の絶対値の範囲を常に固定し，それを超える部分は基数 (base) のベキ乗で実数の絶対値を合わせる，というようにし，小数部分はデータ長の長さに応じて有限桁で打ち切れば，同じデータ長でも整数型よりずっと広範囲の実数を扱うことができる。これが浮動小数点数であり，小数部分を仮数部 (mantissa)，ベキ乗部分を指数部 (exponent) と呼ぶ。但し，整数型がデータ長の範囲内では正確に整数を表現できるのに対し，浮動小数点型は仮数部を有限桁で丸める (round) ため，そこに収まりきれない実数は誤差を含んだ形でになる。整数型は整数の正確な表現であるが，浮動小数点型は実数の近似として表現される。近似であるからこそ，整数型に比べて桁あふれを起こしにくくなっている，とも言える。



一般に，科学技術計算は膨大な演算を行う必要があるため，計算の途中で桁あふれなどが起こるようでは困る。よって，通常は浮動小数点数を用いることが多い。それもできうる限り高速に実行できることが望ましい。従って，単一もしくはごく少数の機械語命令で処理可能な浮動小数点数型を使うことになる。現在主流となっている浮動小数点型が IEEE754 standard で規定されたものである。基数は 2 であり，仮数部の長さが短いものを IEEE754 単精度，長い方を IEEE754 倍精度と呼ぶ。

	全長	符号部	指数部	仮数部
単精度	32	1	8	24(23)
倍精度	64	1	11	53(52)

単精度・倍精度の仮数部が実際の bit 列より 1bit 分長いのは，最初の bit が必ず 1 になるように正規化されるため，実際にはその分は表現されるからである[1]。基数が 2，仮数部が p 桁の浮動小数点数が実数を離散的に近似するのであるから，その分解能は相対的にマシンイプシロン ε_M

$$\varepsilon_M = 2^{-(p-1)} \tag{1}$$

という尺度で表すことが出来る。0 の近傍を除き，一般の実数はこの ε_M 程度の誤差を含んでいると言える。IEEE754 standard において，実数を表現するその分解能はそれぞれ

単精度 $2^{-23} \approx 1.1920928955078125 \times 10^{-7}$

倍精度 $2^{-52} \approx 2.22044604925031308084726333618164 \times 10^{-16}$

となる。それぞれ 10 進数で 7, 16 桁程度ということになる。よって，それ以上の精度を望むのであれば，仮数部の桁数を可変にして ε_M を小さくしなければならない。本稿では，このような仮数部の桁数が可変の浮動小数点数を用いた演算を多倍長計算と呼ぶことにする。

[1] これをケチ表現と呼ぶ。

この多倍長計算を実現するには，通常，まず桁数が可変の整数型演算が可能なライブラリを作成し，それを利用して桁数可変の浮動小数点演算を行うライブラリを構築する[2]。従って，多数の整数演算及びビット操作を必要とするため，計算時間はIEEE754 standardの浮動小数点演算を利用した場合の何倍，何十倍にも増大する。データ長が長くなっているから，必要なメモリの量もずっと増える。

それでも，科学技術計算において多倍長計算が必要とされる場合もある。以下，その具体例を示す。

[2]これとは別のアプローチも存在する。

2.1 悪条件問題に対するアプローチ (の一つ) として

数値計算の標準的なテキストでは，必ず「桁落ち」についての説明がある。これは，お互いに誤差を含んだ近接している浮動小数点数の加減算の結果，絶対値が極端に小さくなるとその分だけ相対誤差が増大する，という現象である[3]。この桁落ちが計算途中に断続して発生し，計算結果の相対誤差がマシンイプシロンよりも極めて大きくなる問題を，悪条件 (ill-conditioned) と呼ぶ。例えば n 次元の連立一次方程式 $A\mathbf{x} = \mathbf{b}$ の係数行列 A の条件数 $\mathrm{cond}(A) = \|A\|\|A^{-1}\|$ が大きい時は，それに比例して数値解 $\mathbf{x}$ に含まれる相対誤差が増大することが知られている。

[3] よく 2 次方程式の解の公式の例が取り上げられているが，これは解決策が簡単に導出できる。これればかり強調されるのはあまり適切ではないと考えている。

もっと簡単な例では，Logistic 写像に基づく数列の計算がある。例えば初期値を $x_0 = 0.7501$ とし，次のような漸化式

$$x_{n+1} := 4x_n(1 - x_n)$$

によって数列 $\{x_n\}$ を生成する。この時，$1 - x_n$ の計算において少しずつ桁落ちを起こし，x_n の相対誤差は部分的に増減はあれど次第に増大していくことが知られている。その結果，$x_{100} = 0.078817989\cdots$ を IEEE754 倍精度計算で求めると 0.515390006286616020 となってしまう [2]。

もし，計算のアルゴリズムそのままでこのような悪条件問題の解の精度を良くしようとするのであれば，多倍長計算を行うしか手はない。但し，計算時間もメモリ量も増大することを覚悟しなくてはならない。また，科学技術計算における全ての悪条件問題が多倍長計算によって解決する訳でもない。

2.2 数値計算の研究用ツールとして

連立一次方程式の係数行列 A と解 $\mathbf{x}$ を

$$A = \begin{bmatrix} 100 & 99 & \cdots & 1 \\ 99 & 99 & \cdots & 1 \\ \vdots & \vdots & & \vdots \\ 1 & 1 & \cdots & 1 \end{bmatrix}, \ \mathbf{x} = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 99 \end{bmatrix}$$

とする。この問題に対して Conjugate-Gradient 法というアルゴリズムを適用し，IEEE754 単精度 (float), 倍精度 (double)，128bit, 256bit, 512bit, 1024bit でそれぞれ計算し，残差のノルム $\|\mathbf{r}_k\|_2 = \|\mathbf{b} - A\mathbf{x}_k\|_2$ の，初期残差 $\|\mathbf{r}_0\|_2$ との比をプロットすると図 4 のようになる。

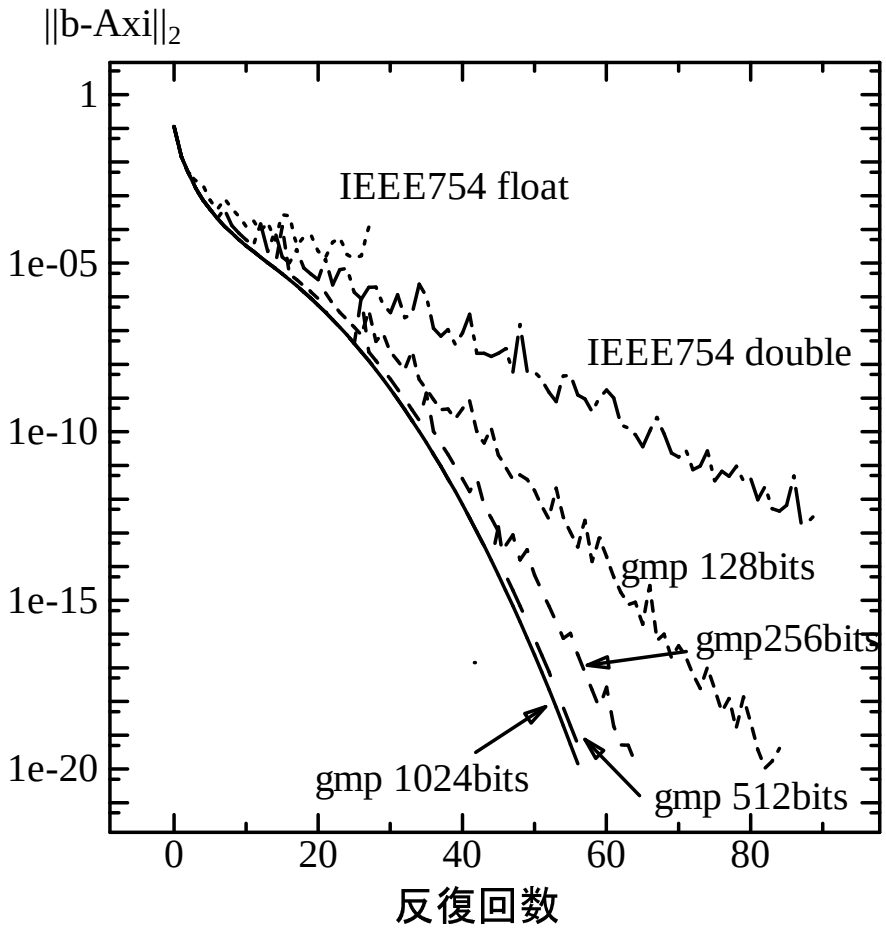


図 4: CG 法における丸め誤差の影響

通常は IEEE754 倍精度での数値実験結果のみを使用することが多いが，こうして多倍長計算も行ってみると，如何に CG 法が丸め誤差の影響に敏感であるかが明確となる。512bit 計算でも僅かだが残差のノルムが上昇しており，完全に単調収束させるにはそれ以上の精度が必要であることがわかる。

このように，数値計算のアルゴリズム，もしくは特定の問題の数値的性質を調べる際に多倍長計算を利用することは有用である。

但し，IEEE754 より演算の実行速度が著しく遅いため，単純に桁を増やすだけでアルゴリズム全体の実行速度が IEEE754 のそれよりも向上することはあまり期待できない。

残差と初期残差との比が 10^{-15} 以下になった時点で反復を停止。後述の Pentium IV マシンで実行。

	IEEE 倍精度	**GMP 4.1**			
精度 (bits)	53	128	256	512	1024
反復回数	107	63	52	48	47
ミリ秒	0.008	0.019	0.018	0.028	0.042

3 GNU MPについて

GNU MP(以下，GMP と略す) は 1991 年から開発が進められてきた ANSI C とアセンブラで記述された多倍長計算ライブラリで，2002 年 9 月現在の最新バージョンは 4.1 である。最初は整数型 (mpz_t) と有理数型 (mpq_t) しか存在しなかったが，Version 2 になって浮動小数点型 (mpf_t) が追加された。後で述べるが，現行の BNCpack はこの浮動小数点型を利用している。GMP を用いて先に挙げた数列を計算する C プログラムの例を以下に示す。ここでは仮数部を 2048bit としてある。

```
#include <stdio.h>
#include "gmp.h"

main()
{
    int i;
    mpf_t x[102];

    mpf_set_default_prec(2048);

    mpf_init_set_str(x[0], "0.7501", 10);
    for(i = 1; i <= 101; i++)
            mpf_init(x[i]);

    for(i = 0; i <= 100; i++)
    {
      mpf_ui_sub(x[i+1], 1UL, x[i]);
      mpf_mul(x[i+1], x[i], x[i+1]);
      mpf_mul_ui(x[i+1], x[i+1], 4UL);
      if(i%10 == 0)
       gmp_printf("%5d %58.50Fe\n", i, x[i]);
    }
}
```

演算を行う部分は全て 2 項演算として人間が翻訳し，書き直してやらねばならず，アセンブラで数値計算を行っているような気分がする。また，変数を使うにもいちいち初期化と解放を行う必要があり面倒である。

反面，変数ごとに仮数部の精度を指定できる上，精度の異なる変数同士の演算も可能である。多倍長計算よって，同じ計算を異なる精度の変数で行って精度を比較するといった使い方が出来る。

Version 4 では試験的とは言え，以前より要望の強かった C++クラスインターフェースが追加された。また，GMP とは別の Project で開発の進められてきた MPFR パッケージ [7][4]も追加され，インストールオプションの追加が必要ではあるが，C++クラスインターフェースと共に使用可能になっている。上の C プログラムをこのインターフェースを用いて書き直すと次のようになる。

```
#include <iostream.h>
#include "gmpxx.h"

main()
{
    mpf_set_default_prec(2048);

    int i;
    mpf_class x[102];

    x[0] = "0.7501";

    for(i = 0; i <= 100; i++)
    {
      x[i+1] = 4 * x[i] * (1 - x[i]);
      if(i%10 == 0)
        gmp_printf("%5d %58.50Fe\n", i, x[i].get_mpf_t());
    }
}
```

[4]MPFR は，GMP の `mpf_t` 型をベースに，IEEE754 standard と互換性を持つように改良されたライブラリである。丸めモードの変更が可能で，高速な初等関数も提供されている。

では，GMP の性能はどの程度なのだろうか？

MPFR のサイトでは，商用ソフトウェアも対象として多倍長計算のベンチマークテストを行い，その結果を公表している。それによれば，1CPU の PC において，浮動小数点数の四則演算では GMP の性能が最高のレベルにあることが分かる。BNCpack で GMP を採用したのもこの点に因るところが大きい。

以下の環境下で同様のベンチマークテストを行ってみる。

CPU Intel Pentium III 750MHz

RAM 384MB

OS Windows XP Professional

C compiler GCC 2.95

GMP Ver.4.0.1 (`./configure; make; make install` でインストール)

Software MuPAD Pro 2.0, Mathematica 4.1

ベンチマークテストに使用したプログラムは，MPFR プロジェクトで提供されているものを使用した。各演算に要する時間は，以下のように，同じ演算を多数繰り返すことで得るようになっている。

```
st = cputime(); for (i=0;i<N;i++) mpf_add(z, x, y);
printf("x+y took %1.2ems\n", (double)(cputime()-st)/N);
```

その結果を表 1 と表 2 に示す。時間は全てミリ秒である。

多倍長計算については，CPU のアーキテクチャに依存して様々なチューン方法が存在しうるため，上記の結果が他の計算機環境で通用するかは不明である。しかしながら，このベンチマークテストを行った環境下では，ほぼ GMP の計算速度が最良のものであるといえる。他にも多倍長計算が可能なライブラリは幾つか存在するが，速度の order は GMP のそれと似たり寄ったりであり，四則演算レベルで見る限り，これ以上の劇的な速度向上が見込める状況にはない。なお，GMP が安定したパフォーマンスを誇っているのは，桁数に応じて使用するアルゴリズムを変えているためであろう。詳細については GMP のマニュアルを参照されたい。また，そこで使用されている各種の四則演算アルゴリズムについては Knuth[9] が詳しい。

表 1: IEEE754 浮動小数点数の演算速度

	IEEE 倍精度
$x+y$	0.000010
$x-y$	0.000011
$x \times y$	0.000009
x/y	0.000053
$\sqrt{x}$	0.00157

表 2: 多倍長計算ソフトウェアの比較

	MuPAD Pro	Mathematica	GMP
10 進 100 桁			
$x+y$	0.0060	0.0054	0.00040
$x-y$	0.0070	0.0092	0.00050
$x \times y$	0.011	0.011	0.0022
x/y	0.012	0.073	0.0041
$\sqrt{x}$	0.036	0.097	0.0076
10 進 1000 桁			
$x+y$	0.010	0.0090	0.002
$x-y$	0.020	0.0161	0.001
$x \times y$	0.27	0.21	0.079
x/y	0.32	0.63	0.13
$\sqrt{x}$	0.29	0.90	0.085
10 進 10000 桁			
$x+y$	0.10	0.040	0.01
$x-y$	0.10	0.07	0.01
$x \times y$	25	5.2	2.62
x/y	26	25	5.53
$\sqrt{x}$	93	29	3.52

以下の環境下で同様のベンチマークテストを行ってみる。

CPU Intel Pentium IV 2.2GHz

RAM 512MB

OS RedHat Linux 7.3

C compiler GCC 2.96

GMP & MPFR Ver.3.1.1, Ver.4.1 (`./configure; make; make install` でインストール)

GMP 3.1.1, 4.1, MPFR 間での比較

	GMP 3.1.1	GMP 4.1	GMP 4.1& MPFR
10 進 100 桁			
$x+y$	0.00023	0.00012	0.00020
$x-y$	0.00023	0.00012	0.00036
$x \times y$	0.00184	0.00057	0.00060
x/y	0.00297	0.00168	0.00187
$\sqrt{x}$	0.00692	0.00264	0.00231
10 進 1000 桁			
$x+y$	0.00080	0.00045	0.00057
$x-y$	0.00079	0.00043	0.00097
$x \times y$	0.0815	0.0242	0.0222
x/y	0.145	0.0442	0.0426
$\sqrt{x}$	0.195	0.0324	0.0316
10 進 10000 桁			
$x+y$	0.0059	0.0034	0.0040
$x-y$	0.0058	0.0033	0.0070
$x \times y$	2.80	0.859	0.865
x/y	5.67	1.83	1.83
$\sqrt{x}$	9.94	1.22	1.20

4 BNCpackについて

BNCpackは，Basic Numerical Computation PACKageの略称である。この名から分かる通り，十把一絡げの数値計算のうすぅい教科書に書いてある程度のアルゴリズムを実行できる環境を作りたい，というのが本来の目的であった。が，途中で欲が出，ついでなら多倍長化してしまえとGMPがVersion 2の時にそれを採用し，原型となる基本線型計算の関数群を作り上げた。幸いGMPの上位互換性に助けられ，大幅な仕様変更なしで現在のGMPでも利用可能である。

言語は何でも良かったのだが，Fortranユーザが激減しているし，そもそもGMPがVersion 3まではANSI Cのインターフェースしか提供していなかったこともあって，C(not C++)で記述することになった。

現在提供されているのは，次の関数群である。但し「自分の研究用」に作成したために，テストが不十分で，buggy な部分も多々あると思われる。

1. GMP に存在しない初等関数 (Ver.0.3 より MPFR の初等関数が利用可能)
2. 複素数演算 (C++の Complex テンプレートとは全く別の実装系として提供。GSL[8] とも互換性なし)
3. 基本線型計算 (ベクトル，正方密行列の和・差・内積 (積)・スカラー倍など)
4. 連立一次方程式 (直接法，反復法，CG 法)
5. 行列の固有値・固有ベクトル計算 (現状ではべき乗法・逆べき乗法のみ)
6. 非線型方程式 (Newton 法，準 Newton 法, Regula-Falsi 法)
7. 代数方程式 (DKA 法)
8. 数値積分 (台形則，Romberg 積分)
9. 常微分方程式の初期値問題 (陽的・陰的 Runge-Kutta 法)

BNCpack の特徴の一つに，GMP の浮動小数点型と同様，変数毎に桁数を可変に出来る点にある。

なるべく関数の引数を簡素にするため，よく使用すると思われるデータ型は構造体を用いて typdef してある。現在の所，以下のデータ型が利用出来る。

複素数 `MPFCmplx`

実係数多項式 `MPFPoly`

配列 `MPFArray`, `CMPFArray`(複素数配列)

スタック `MPFStack`

ベクトル `MPFVector`

実正方密行列 `MPFMatrix`

以上のデータ型は全て，デフォルトの桁数とは別に設定が可能である。例えば複素数型は

```
typedef struct{
    unsigned long int prec;
    mpf_t re;
    mpf_t im;
} mpfcmplx;

typedef mpfcmplx *MPFCmplx;
```

と宣言してある。このうち `prec` は複素数の精度 (bit 数) を保持している。よって，BNCpack で独自に宣言した変数においても，GMP の `mpf_t` 型と同様，一つのプログラム中に異なる桁数のデータが混在できるようになっているのである。

以下，BNCpack を使ったプログラムの例を示す。

4.1 複素数演算

複素数の和を計算するプログラムの一部を以下に示す。

```
#ifdef USE_GMP /* GMP 使用 */

  /* 変数の宣言 */
  MPFCmplx mpfca, mpfcb, mpfcc;

  /* 変数の初期化 */

  /* Default: 128bit */
  set_bnc_default_prec(128);

  /* Default の精度で初期化 */
  mpfca = init_mpfcmplx();
  mpfcb = init_mpfcmplx();

  /* 256bit で初期化 */
  mpfcc = init2_mpfcmplx(256);

  /* mpfa := 1 + 2i */
  /* mpfb := 3 + 4i */
  /* mpfc := rand() + rand() i */
  set_real_mpfcmplx_ui(mpfca, 1);
  set_image_mpfcmplx_ui(mpfca, 2);
  set_real_mpfcmplx_ui(mpfcb, 3);
  set_image_mpfcmplx_ui(mpfcb, 4);
  set_real_mpfcmplx_ui(mpfcc, rand());
  set_image_mpfcmplx_ui(mpfcc, rand());

  /* mpfcc := mpfca + mpfcb */
  add_mpfcmplx(mpfcc, mpfca, mpfcb);

  /* 出力 */
  printf("mpfca + mpfcb = "); print_mpfcmplx(mpfcc);

  /* 変数領域の解放 */
  free_mpfcmplx(mpfca);
  free_mpfcmplx(mpfcb);
  free_mpfcmplx(mpfcc);

#endif
```

4.2 線型計算

二つの行列 mpfmat_a, mpfmat_b の積を計算する例を以下に示す。

```
#ifdef USE_GMP
  /* 変数の宣言 */
  MPFMatrix mpfmat, mpfmat_a, mpfmat_b;
  mpf_t tmp, sqrt2;
  long int dim, row_dim, col_dim, i, j;

  /* Default: 128bit */
  set_bnc_default_prec(128);

  /* 変数の初期化 */
  mpfmat   = init_mpfmatrix(row_dim, col_dim);
  mpfmat_a = init_mpfmatrix(row_dim, col_dim);
  mpfmat_b = init_mpfmatrix(row_dim, col_dim);
  mpf_init(tmp); mpf_init(sqrt2);
  mpf_sqrt_ui(sqrt2, 2UL);

  /* mpfmat[i][j] := sqrt(2) * (i+1) / (j+1) */
  for(i = 0; i < row_dim; i++)
  {
    for(j = 0; j < col_dim; j++)
    {
       mpf_mul_ui(tmp, sqrt2, i+1);
       mpf_div_ui(tmp, tmp, j+1);
       set_mpfmatrix_ij(mpfmat, i, j, tmp);
    }
  }

  /* mpfmat_a := mpfmat */
  subst_mpfmatrix(mpfmat_a, mpf_mat);

  /* mpfmat_b := mpfmat * mpfmat_a */
  mul_mpfmatrix(mpfmat_b, mpfmat, mpfmat_a);

  /* 出力 */
  printf("mpfmat_b:\n");
  print_mpfmatrix(mpfmat_b);

  /* 変数領域の解放 */
  mpf_clear(tmp); mpf_clear(sqrt2);
  free_mpfmatrix(mpfmat);
  free_mpfmatrix(mpfmat_a);
  free_mpfmatrix(mpfmat_b);
#endif
```

4.3 連立一次方程式

連立一次方程式の係数行列を LU 分解して解くプログラムを以下に示す。変数の宣言部は略してある。

```
#ifdef USE_GMP

  /* Default: 256bit */
  set_bnc_default_prec(256);

  /* 初期化 */
  mpf_init(reps); mpf_init(aeps);
  mpfa = init_mpfmatrix(DIM, DIM);
  mpfb = init_mpfvector(DIM);
  mpfx = init_mpfvector(DIM);
  mpfans = init_mpfvector(DIM);

  /* 係数行列 A，定数ベクトル b，真の解 x を得る */
  get_mpfproblem(mpfa, mpfb, mpfans);

  /* 係数行列 A を標準出力に表示 */
  print_mpfmatrix(mpfa);

  /* LU 分解と後退代入 */
  ret_mpf = MPFLUdecomp(mpfa);
  ret_mpf = SolveMPFLS(mpfx, mpfa, mpfb);

  /* 結果表示 */
  for(i = 0; i < DIM; i++)
  {
    printf("%5ld ", i);
    mpf_out_str(stdout, 10, 0, \
    get_mpfvector_i(mpfx, i));
    printf(" ");
    mpf_out_str(stdout, 10, 0, \
    get_mpfvector_i(mpfans, i));
    printf("\n");
  }

  /* 変数領域の解放 */
  mpf_clear(reps); mpf_clear(aeps);
  free_mpfmatrix(mpfa);
  free_mpfvector(mpfb);
  free_mpfvector(mpfx);
  free_mpfvector(mpfans);

#endif
```

5 BNCpackの欠点

現状のBNCpackは様々な欠点が存在する。作成者としては欠点しかないように見えたりするが，大まかに箇条書きしてみると次のようになろうか。

1. 全てANSI Cで記述してあるので，プログラムの開発が煩雑になる。同じくANSI Cのインターフェースした提供していないGSL[8]にも共通して言える。ユーザを増やすにはC++クラスインターフェースの提供は必須事項であろう。既存のtemplateに組み込み可能な形にして提供する方法もあるかも知れない。

2. 関数の種類が少ない。但し現状より手を広げる予定はない。

3. 多倍長線型計算が遅い。この場合，遅いというのは定評のある線型計算ライブラリのBLASやLAPACK[10]のルーチンと比較しての話である。これらについては多倍長化するという予定が以前より漏れ聞こえて来るので，リリースされればそちらへ乗り換えることも検討する必要がある。

4. そもそも多倍長計算を使いこなすノウハウがまた整っていない。多倍長計算を前提として，過去の議論を再検討していく必要もあると思われる。

以下，これらの欠点について解説する。

5.1 プログラムの作成が煩雑になる

現在の所，BNCpack はC のインターフェースしか持っていない。そのため変数の初期化・解放は関数を呼び出す必要があるし，演算部分も GMP 同様，全て 2 項演算に落として関数を呼び出す手順を考えなければならない。

もっとも，プログラム例さえふんだんに揃えておけば，それを改変するだけでユーザの実行したい計算は出来るようになる。最初は面倒でも，経験を積むにつれて望みのものがすぐに作れるようになるだろう。また，C++クラスインターフェースによってプログラムを書く作業自体は省力化できても，実行時間が劇的に向上することは望めない。むしろ，全ての実行過程を記述しておいた方が，計算時間の見通しがついてよい場合もあるだろう。C++への移行は数値計算用に STL がきちんと整ってからでも遅くはないと考えている。

5.2 関数の種類がまだ少ない

数値計算全般を扱うライブラリとして見た場合，最近リリースされたGSL(GNU Scientific Library)[8]と比較するまでもなく，現行のBNCpackの関数は少なすぎる。

まず，Bessel関数等の特殊関数は自分で使う予定がないため全く揃えていない。線型計算ついても，行列の固有値・固有ベクトル計算はごく一部が用意されているだけである。また，密行列を扱う関数しか揃っていない。全体的に見て，数値計算のごく基盤的な関数が一応用意されているだけなので，科学技術計算用としてはより上位層の関数群も必要になるだろう。

線型計算については，IEEE754 standardの範囲ではLAPACK(+BLAS)を使うのが常識になってきており，多倍長計算をサポートするという噂が時折聞こえてくる。そうなってしまえば，線型計算の大部分はLAPACKを利用すればいいか，と安易な考えが捨てきれず，種類を増やすことに躊躇しているのが現状である。

5.3 線型計算が遅い

線型計算，特に行列演算に関しては素朴な作りになっているため，次元数が大きくなると計算時間が長くなる傾向がある。例として，先に挙げた行列の積を計算するプログラムの実行時間 (積を計算する部分のみ) を Mathematica と比較してみる。計算機環境は前述の通りである。なお Mathematica の計測スクリプトは

```
n := 50;
digits := 100;
a := Table[N[Sqrt[2]*i/j, digits], {i, 1, n}, {j, 1, n}];
b := Table[N[Sqrt[2]*i/j, digits], {i, 1, n}, {j, 1, n}];
time := Timing[a . b;];
Print[time];
Clear[a, b, time, n, digits];
```

とした。

結果は次の通りである。

10 次元			
	50 桁	100 桁	200 桁
Mathematica	0.03	0.04	0.05
BNCpack	0.00	0.01	0.01

50 次元			
	50 桁	100 桁	200 桁
Mathematica	1.87	2.17	3.4
BNCpack	0.21	0.36	0.98

100 次元			
	50 桁	100 桁	200 桁
Mathematica	12.7	15.2	24.0
BNCpack	2.12	3.57	8.19

200 次元			
	50 桁	100 桁	200 桁
Mathematica	87.8	110.7	178.6
BNCpack	17.0	27.8	64.5

Mathematica より高速であることは前に挙げた表 2 より明らかである。問題は次元数の大きさのほぼ 3 乗に比例して計算時間が増大している所で，これはごく素朴なアルゴリズムに基づいて計算しているためである。また，Mathematica の場合，同じ次元数において，桁数を増やしても計算時間の伸びが少ないことに気づく。実際，10 次元の場合に桁数さらに増やして計算してみると

10 次元			
	1000 桁	10000 桁	100000 桁
Mathematica	0.25	5.79	148.2
BNCpack	0.08	2.82	97.9
Math./BNC	3.1	2.1	1.5

となり，段々と差が縮まってくることが分かる。

他の関数についてはまだベンチマークテストを行ってはいないが，線型計算は多くの数値計算において基盤的な役割を果たすため，これらを利用した多くの関数は同様の傾向を持つと予想される。もっと高速に線型計算を実行するための方策を考える必要がある。

5.4　多倍長計算を使いこなすノウハウの欠如

歴史を降り返ってみると，ハードウェアの高機能化に合わせて，サポートされる浮動小数点数の有効桁数は増えてきている。ユーザにとっては，計算速度があまり変化しないのであれば，精度は高いに越したことはない。そこで，多倍長計算が利用できるようになってくると，次のような疑問が浮かんでくる。

1. 浮動小数点数の有効桁数に鋭敏なアルゴリズムは，多倍長計算を利用することで，現在最良とされている頑健なアルゴリズムよりも速度面で凌駕する可能性があるのではないか？

2. 悪条件問題において，浮動小数点数の精度が数値解の精度に与える影響はどの程度のものか？特に最良の精度との関係を図示するとどのようなものになるのか？

これらを理論的に追求するのは困難であると思われるが，数値実験を網羅的に行うことである程度の見積もりを得ることは可能であろう。但し前者については，前節のベンチマークテストの結果から，現在の所あまり期待できないと言ってよい。大雑把に図示すれば図 2 のようになろうか。

従って現状では悪条件問題において，必要な精度を得るために多倍長計算を使用する，ということが，多倍長計算の一番の存在意義となろう (図 3)。それも，問題自体の誤差が任意に調整できる，限定された状況においてのみである。多倍長計算を適用するにあたっては，図 1 において，入力数値の変換誤差，計算途中に発生する丸め誤差，数値を出力する際に発生する出力（変換）誤差にのみ，効果があることをしっかり認識しておく必要がある。さもなければ，多大な誤差を含む数値を使って，無意味に計算機資源を費やすことになるだろう。

6　結論と今後の課題

日常的に使用する数値計算は，CPU で直接処理できる範囲の，固定された桁数の浮動小数点演算を用いて高速に行われるべきである。この原則が変更されることは当面ないだろう。従って，多倍長計算は日常的な数値計算を補完する役割を担うことになる。

現在，多倍長計算を利用するには，Mathematica, MuPAD のような数式処理ソフトを使うか，本稿で取り上げた GMP のようなライブラリを自分のプログラムとリンクして呼び出すことになる。両者を比四則演算レベルで比較すると，GMP が最も高速な多倍長計算環境を提供していることも明らかになった。よって，多倍長計算の機能に絞ってみれば，GMP という土台の上に数値計算用のライブラリを構築することが望ましい。BNCpack はその一例である。

が，BNCpack を実用に供するにはまだまだ改良すべき点が数多く存在する。bug 潰しは当然だがもっと根元的な所に根ざす弱点としては

1. 関数の種類が少ない。
2. 線型計算に改良の余地がある。
3. 全て ANSI C で記述してあるので，プログラムの開発が煩雑になる。
4. そもそも多倍長計算を使いこなすノウハウがまた整っていない。

が挙げられる。

今後の課題はこれらの欠点を解消することで，特に最後については過去の議論をもう一度再検討する必要も出てくる。

だが，BNCpack はあくまで一個人が自分の研究に開発したものであって，かなり独りよがりなものである。全く別のアプローチを取った科学技術計算用の多倍長計算ライブラリが free かつ open なものとして登場することを願って止まない。

参考文献

[1] 永坂秀子, 計算機と数値解析, 朝倉書店, 1980.

[2] 「ソフトウェアとしての数値計算」, http://www.pas-net.jp/nasoft/

[3] 森正武, 数値解析, 共立出版, 1973.

[4] 伊理正夫・藤野和建, 数値計算の常識, 共立出版, 1985.

[5] BNCpack, http://member.nifty.ne.jp/tkouya/na/bnc/

[6] GNU MP, http://swox.com/gmp/

[7] The MPFR Library, http://www.mpfr.org/

[8] The GNU Scientific Library, http://sources.redhat.com/gsl/

[9] D.E.Knuth/中川圭介・訳,「準数値算法/算術演算」, サイエンス社,1986.

[10] LAPACK, http://www.netlib.org/lapack/

[11] MuPAD, http://www.mupad.de/

[12] Wolfram Research, http://www.wolfram.com/