論　説

多倍長浮動小数点クラスの開発と応用

Development of a Multiple-Precision Floating-Point Class and Its Applications

津　曲　　隆
Takashi TSUMAGARI

C++言語を用いて多倍長浮動小数点演算を処理するBfloatクラスを開発した。このクラスでは，四則演算，数学関数および入出力などの数値クラスに必要な基本機能をC++言語の多重定義を使って実現しており，自然な形式で多倍長プログラムを記述することが可能になっている。また，処理速度に関して，アセンブラで記述されている著名な多倍長計算ソフトウェアUBASICと比較実験を行った。その結果より，本クラスは有効桁数の大きいところでUBASICを凌駕する高速性能を持つことを示す。最後に，誤差評価の観点から，多倍長計算を利用したいくつかの応用例について述べる。

1．はじめに

今日の計算機のほとんどは，浮動小数点を表現するdouble型等の基本データ型がハードウェアでサポートされている。通常の利用形態では，これらのデータ型の精度（高々十数桁）で十分であることが多いが，微妙な問題になると数値的に不安定な計算に出くわすことも少なくない。基本データ型を使用する限り，そういった不安定性を避けるには何らかの新しいアルゴリズムを見いださねばならず，それは一般に一つの研究テーマになり得るほどの困難を伴う場合が多い。このような時に，多倍長計算を利用できれば，当面の問題を回避でき，本質を見失うことなく仕事や研究等を進展させることができる。本論文は，この目的に使用する多倍長計算用プログラムパッケージの検討とその開発結果に

ついての報告である。

多倍長計算の著名な例として円周率の数億桁計算というのがあるが，これは多倍長計算の中でも特異な例で，実用上は千桁程度までが重要になる。この目的で数千桁レベルの汎用多倍長プログラムがこれまでにもいくつか開発されてきた。例えば，FORTRANで記述されたBrentによるMP[1)]，SmithによるFM[2)]パッケージ，また，入手しやすいソフトとしては木田によるUBASIC[3)]，REDUCEやMATHEMATICAなどの数式処理ソフトの多倍長実数パッケージなどがある。これらのパッケージにおいて，FORTRAN系のプログラムでは全ての演算をサブルーチンの形式で呼び出す必要があるため多倍長プログラムの可読性が悪く，本格的なプログラム作成はかなり難しいという欠点がある。一方，アセンブラで記述されたUBASICは，高速処理に特徴があるが，当然ながら移植性が悪く，また既存のプログラムとのインターフェースの面でも問題がある。数式処理ソフトは，UBASIC同様のインターフェースの問題と同時に，基本的に記号処理を重点においた処理系であるため，処理速度が遅く実用性に劣る。

今回，これまでの多倍長パッケージの欠点を解決するために，使いやすさ・移植性の面を考慮した新しい浮動小数点演算パッケージを開発した。これは，C++言語[4)]を用いて作成しており，その規模は行数に換算しておおよそ4500行程度である。C++言語では演算子の多重定義が可能で，ユーザ定義クラスに対して，＋，－，＊，／などの演算子を自由に定義できるという特徴がある。この機能によって，多倍長プログラムでも通常プログラムと同程度の可読性を確保できるため，大規模プログラムの作成もさほど困難でなくなる。さらに，高水準言語であるから移植性にも優れ，また，既存のプログラムとの結合の面でも問題が生じないなどの，特に数値クラスに対して多くの優れた特徴を有している。筆者と同様な考えでC++を利用したパッケージを平山も最近発表しており[5)]，その有効性を強調している。本パッケージは，いくつかの工夫を加えることによって，平山のパッケージに比べて処理速度の面で優れた性能を有しており，本論文では，そのアルゴリズムと性能およびこれを利用した応用例について述べる。

２．多倍長浮動小数点数クラス

２．１　内部表現とデータ構造

有限精度の実数値 R は，一般に，

$$R=S\times(f_1r^{-1}+f_2r^{-2}+f_3r^{-3}+\cdots\cdots\cdots\cdots+f_nr^{-n})\times r^m \qquad (1)$$

と表せる。ここで，S：符号，m：指数部，$f_j(j=1\sim n)$：仮数部，r：基数である。ただし，仮数部は正規化してあるものとする（すなわち，$f_1>0$）。基数 r は，$r\geqq 2$を満足する任意の整数で構わないが，ここでは，便宜上，

$$r=b^k \qquad (b,k \mid b\geqq 2,\ b,k\in \boldsymbol{N}) \qquad (2)$$

と仮定する。

Bfloat クラスでは，(1)式中の諸量を付録に示す形式のクラスデータで定義している。符号 S は int 型変数に格納し，R の正負に応じて $S=\pm 1$，また，$S=0$ で $R=0$を表現する。指数部 m は long int 型（32bit）であるから，このクラスでは，およそ，

$$-2^{31}\log_{10}r \leqq \log_{10}\mid R\mid \leqq +2^{31}\log_{10}r \qquad (3)$$

の数値範囲を表現できる。仮数部 $f_j(j=1\sim n)$ は，Mtype 型の n個の一次元配列で表現し，その配列はオブジェクト生成時に動的に確保している。Mtype 型は，次節の表１の基数に対する仮数部を表現するために，long int 型を typedef したものである。配列長 n で有効精度が決まり，これを10進に換算した有効桁数 p は，

$$(n-1+1/k)\log_{10}r \ \leqq\ p\ \leqq\ n\log_{10}r$$

となり，n が大きいとき，おおよそ

$$p\fallingdotseq n\log_{10}r \qquad (4)$$

で与えられる。基数については次節で詳細に述べる。最後に，クラス定義にある ZeroPos であるが，これは，$f_{j_z}\neq 0(1\leqq j_z<n)$ かつ $f_j=0(j>j_z)$ であるとき，

$$\text{ZeroPos}=j_z+1, \qquad (5)$$

とし，もし $f_n\neq 0$であれば，

$$\text{ZeroPos}=n+1 \qquad (5)'$$

と定義する。ZeroPosは，LSD側にあるゼロ要素を除いた実質的な配列長を表す量であるが，データとしては冗長である。しかし，ZeroPosを使うと，ZeroPos≦nであるオブジェクトに対してLSD側のゼロ値も計算に含めてしまう無駄な処理を省くのに効率がよいため，あえてデータとして定義した。

2.2　基数について

基数は，多倍長計算の処理速度を決定する重要な因子の一つである。これを定めるには，その①大きさと②何進表現とするか，の二点について考えなければならない。まず，①について議論する。一般に基数が大きいほど処理速度を向上できるが，自由に大きくできるわけではない。その上限は，通常，使用する乗算アルゴリズムによって決まる。多倍長乗算を$O(n^2)$アルゴリズムで行うものとすると（理由は3章で述べる），その中間過程で

$$g=f_i\times f_j \tag{6}$$

の形式のMtype型同士の乗算が現れる。$g\leqq r^2-1$であるから，積gにはf_jの2倍以上のビット長が必要となるため，gをℓbit長のデータ型とすると，

$$r\leqq\lfloor 2^{\ell/2}\rfloor \tag{7}$$

が基数の上限となる。従来の多倍長パッケージでは[1)2)3)5)]，整数演算のみを考慮してrの大きさを定めているようである。そのため，ℓbitデータ型としては32bit整数（long int型），Mtype型はその半分のshort int型（16bit）として，$r=2^{16}$または$r=10^4$に選んだものが多い。しかしながら，最近の計算機ではほとんど標準的に浮動小数点レジスタを内蔵しており，こちらの方が整数演算レジスタよりもビット長が長いのが普通である。例えば，IEEE規格のdouble型では仮数部が53bit，また，long double型の場合には64bitある。この仮数部ビットを上記のℓbitのデータ型として利用すれば，32bitの整数演算よりも有利に

---

*1 $\lfloor X\rfloor=\max\{q \mid q\leqq X,\ q\in$整数$\}$を意味する。同様な記号として，$\lceil X\rceil=\min\{q \mid q\geqq X,\ q\in$整数$\}$も以後使用する。

なることは明らかであるから[*1]，本パッケージではgを浮動小数点型とし，実数計算によってgを求めることにした。

次に，②について考察する。数値表現は2進か10進のどちらかを基本にするのが普通であるから，(2)式において$b=2$あるいは$b=10$とすることになる。前者はメモリの使用効率がよく，また，少なくとも基数を後者よりも大きくできるという利点がある。さらには，ビット処理をうまく利用すれば，より高速のプログラムを実現することも可能である。しかしながら，そういったプログラムにはマシンやOSに依存した記述が陽に現れ，移植が非常に難しくなることが予想される。今回は，マシン依存性を極力少なくし，移植性に優れた汎用パッケージを開発することも目的としているため，本パッケージではビット処理は行わないことにした[*2]。ビット処理をしないのであれば，2進でも10進表現でもさほど大きな違いはなくなるから，プログラミングの容易さを考えて，今回は後者の10進表現を採用した。なお，計算に関しては本質的なことではないが，10進表現では，基数変換を行う必要がないため多倍長データを（ASCII形式でも）高速に入出力できるという特徴があり，入出力を頻繁に行うアプリケーションでは有利になる。

表1　Bfloatクラスで採用した基数

gのデータ型	基数r（f_jの保存に必要なビット数）
double型（$\ell=53$bit）	10^7　（24bit）
long double型（$\ell=64$bit）	10^9　（30bit）

IEEE規格の浮動小数点型を仮定し，10進表現を採用することと(7)式を考慮

[*1] もちろん，個々のマシンに対応してアセンブラで処理を記述するならば，$\ell=64$bitで整数乗算を実行することも可能であるようだが，プログラムの汎用性を考えて，これは行わない。

[*2] 本パッケージはPC98上でボーランド社製BC++3.1を用いて開発した。ビット処理を含んでいないため，ソースコードをほとんど修正することなくGNU g++でコンパイルでき，NEC製EWS4800へも簡単に移植できた。

して決定した基数を表1に示す。(6)式の g のデータ型を double 型あるいは long double 型のいずれにするかは計算機に依存し（具体的には前者はワークステーション，後者はインテル MPU 系パソコンを想定している），それによって基数が異なる。この選択には，ヘッダファイル内のマクロを使うようにしている（付録参照）。なお，本パッケージは本質的にハードウェアに依存するのは基数値だけであるため，IEEE 規格を採用していない計算機にも容易に移植できるものと思われる。

2.3　利用可能な演算子と関数の種類

Bfloat クラスでは，数値演算に必要な基本的なメソッドである，四則演算子，比較演算子および入出力演算子を既存の基本データ型に対するそれと同じ形式で多重定義してある。多重定義のコンパイラによる処理テクニックは実に単純なものであるが，これのプログラミングに対する効果は絶大である。多重定義によって，例えば，図1のようなプログラミングが可能になり，従来と同じ感覚で多倍長演算プログラムを作成できる。ところで，Brent[1] や Smith[2] らも指摘しているように，ある種の演算では，引数が long int 型の数値で表現できるときには高速に処理できる場合がある（例えば，Bfloat と long int の乗算の場合，この計算時間は $O(n)$程度で済む)。付録のヘッダファイルでは省略しているが，本クラスでも，引数が long int 型（正確には，r 以下の数値）の時に高速処理が可能となる演算については，その処理を多重定義している。

```
Bfloat::SetPrecision(100);          // 有効桁数≒100に設定
Bfloat x, y;                        // Bfloat型オブジェクト宣言（ﾃﾞﾌｫｰﾙﾄｺﾝｽﾄﾗｸﾀ）
Bfloat z=ltobf(7);                  // ｵﾌﾞｼﾞｪｸﾄ宣言(ｺﾋﾟｰｺﾝｽﾄﾗｸﾀ),ltobfによる型変換
long int i;
                    .
                    .
                    .
cin >> y;                           // Bfloat型の標準デバイスからの入力
x += y;                             // Bfloat型同士の加算
if(x>0)                             // Bfloat型と整数型の比較
    x = z * i;                      // Bfloat型と整数型の乗算
else
    x /= i;                         // Bfloat型と整数型の除算
cout << precision << space(10)      // Bfloat型の標準デバイスへの出力（ﾏﾆﾋﾟｭﾚｰﾀ使用）
     << intpart(3) << plength(53)
     << x << '¥n';
```

図1　Bfloatクラスの利用例

多倍長数値クラスに特有な精度設定・取得関数および型変換関数，さらに，数学定数取得，符号操作，整数化，剰余，べき乗，初等超越関数などの基本的な数学関数も提供している。数学関数も，標準的なC言語ライブラリと同様な関数名で多重定義してあり，従来と同じ感覚で利用できる。数学関数の中で，sinHP系関数は象限三角関数と呼ばれ[6]，FORTRAN等の標準関数としては採用されていないが，著名な数学ライブラリNUMPAC[7]に採用されているものである。この関数は，普通の三角関数が周期2πであるのに対し，周期が整数値4と厳密に表現できるため，区間縮小を高速かつより正確に実行できる（区間縮小のために，内部では普通の三角関数も象限関数を経由している）。従って，象限三角関数を使える状況にあるならば，積極的にこれを利用した方がよい[6]。同様なことが指数関数についてもあり，区間縮小のために，e^xは一旦10^x関数に変換して計算している。

C++言語の知識があれば，付録のヘッダファイルだけで，Bfloatクラスの演算子や関数の利用方法は十分に把握できるものと思われるので，一部を除いて個々の利用方法の詳細については省略する。

2.4　オブジェクトの生成と有効桁の指定

Bfloatクラスには，デフォールトコンストラクタ（オブジェクトを生成しゼロに初期化）とコピーコンストラクタ（オブジェクトを生成し，別のオブジェクトで初期化）の二つのコンストラクタがある。生成されるオブジェクトの有効桁数の指定には，SetPrecision関数を使う。例えば，有効桁をd桁にしたい場合，

Bfloat::SetPrecision(d);

とする。これで，付録に示す静的クラスデータBfltLengthが

BfltLength ← $\lceil d / k \rceil$

にセットされ，これ以降に生成されるBfloatオブジェクトの配列長Lengthには，BfltLengthの値が使用されるようになる。なお，現在設定されている有効桁数（$=k$*BfltLength）を取得するにはGetPrecision静的関数を使用する。

２．５　型変換

既存の基本データ型に対する変換コンストラクタ（Conversion Constructor: CC）を定義しておくと，プログラムの記述が容易になるが，多倍長数クラスの場合には，CC によって厄介な問題がおきることがある。double や long double 型（まとめて REAL 型と略記する）および long int 型（INT 型と略す）の Bfloat クラスへの変換（キャスト）を対象にして，その問題点を述べる。

まず，前者の変換において，ある厳密値3.8を多倍長変数 x に代入する処理を例にしよう。ただし，Bfloat クラスには REAL 型用の CC が定義されているものとする。この処理を，

```
Bfloat x;
x=3.8
```

と記述したとする。コンパイラは，２行目右辺の定数3.8を２進内部表現に変換するわけであるが，3.8は２進では循環小数 $(11.11001100\cdots)_2$ となるため，その内部表現には丸め誤差が含まれることになる。そして，その２進表現が CC によって Bfloat 型へ暗黙のうちにキャストされ，最終的に誤差を伴う数値3.8 $\{1+o(\varepsilon)\}$ が x には代入されてしまう[*1]。ここで，ε は REAL 型に対するマシンイプシロンである。もしこの x が，それ以後の多倍長演算に使用されるようであれば，最終的に得られた結果の信頼性は極めて疑わしいものとなり，最悪の場合，誤った結果を信用してしまうことにもなりかねない。このような問題が REAL 型用の CC にはあるため，多倍長数に対してこれを定義しておくのは非常に危険である。一方，INT 型の変換でも同様な問題が生じる。INT 型用の CC が定義されていると，例えば，利用者が上の例と同じく x に3.8を代入するつもりで同じような記述をした場合，２行目は，

```
x=(Bfloat)(int)3.8
```

[*1] 例で用いた浮動小数点定数3.8は，有理数19/5で表せる。有理数定数であれば丸め誤差の問題は発生しない。有理数を扱うために，今回，$(n/d)\times 10^e$ 形式の（拡張）有理数定数クラス ExConst も開発している。ここで，n,d,e は long int 型である。付録では記述を省略しているが，この ExConst クラスとの演算も Bfloat には多重定義してある。

と解釈され，利用者には何の警告も発せられることなく，x に定数3が代入されてしまう。もちろん，REAL 型用の CC を同時に定義していれば，このような誤ったキャストを避けることはできるが，先に述べた理由でそれは定義できない。

CC はプログラムの自然な記述を可能にする優れたインターフェースであることは言うまでもない。しかしながら，少なくとも多倍長数に関しては，自然な記述形式であるがために，利用者が変換誤差あるいは誤変換などのバグを見落としやすいという問題を含む。それゆえ，利用に際して多少不便さはあるが，異なる型間の演算や代入では明示的に型変換し，できるだけバグの混入を抑制するという立場を優先して，Bfloat クラスでは CC を一切定義しなかった。これによって，明示的に変換していないところはコンパイルエラーとなり，誤変換の生じる可能性のある記述はチェックできる。変換には，REAL 型では_ftobf（あるいは_lftobf）を，INT 型については ltobf の型変換関数を使う（使用例を図1に示す）。もちろん，_ftobf を使っても先の例と同様な変換誤差の問題がなくなるわけではない。それゆえ，もし REAL 型の定数を記述する必要がある場合には ExConst クラスを利用するか，また変換時間を気にしないのであれば atobf 変換関数を使った方が安全である。ところで，Bfloat クラスには，処理の高速化のために long int+Bfloat 形式の演算子が定義してある。これにも INT 型用の CC と同様の問題を生じるが，今のところ，これに対するチェック機構は用意していない。今後，何らかの対策を行う予定でいる。

型変換関数のなかで bftobf 関数について，付録の記述だけでは理解しがたいと思われるため，ここで簡単に説明しておく。有効桁数の異なる二つの Bfloat オブジェクト x，y に対して，x を y にコピーするのに，y=x とすると実行時エラーとなる。これも一種の型変換であるから，bftobf 関数を使って，

```
x.bftobf(&y);
```

と明示的に変換しなければならない。なお，当然であるが，x.ZeroPos>y.Length のケースでは x の仮数部末尾の情報が失われる。bftobf 関数の戻り値は，情報が失われた場合は0，それ以外は1となるようにしてある。

２．６　入出力

Bfloat クラスでも，抽出（>>）と挿入（<<）演算子を使って，既存のデータ型と全く同様な形式で入出力ストリームを利用できる。抽出演算子にはBfloat クラス専用の６個の出力オプションがある。各オプションを表２にまとめておく。設定には SetOutForm 関数かまたはマニピュレータを使用する（付録参照）。前者はオプションの規定値を変更するもので，後者はその場限りの一時的なオプションの変更の際に使用する。マニピュレータの設定の方が規定値より優先される。オブジェクト x に対して，その有効桁を表示し，整数部３桁，小数部50桁および空白を10桁毎に挿入して標準デバイス（cout）に出力する処理をマニピュレータで行うには図１の例のようにする。

表２　出力オプション

オプション	データ型	意　味
precision	Switch 型*	有効桁数の表示(ON)/ 非表示(OFF)の切り換え．（初期値 =OFF）
truncate	Switch 型*	仮数部 LSD 側ゼロ要素の表示(ON)/ 非表示(OFF)の切り換え． ゼロ要素がある場合のみ有効．（初期値 =OFF）
form	Switch 型*	他のオプションに従って表示形式を整える(ON)/ 無視して 高速に出力(OFF)の切り替え．（初期値 =ON）
space	int	指定した桁数毎に空白を挿入．空白を入れたくない場合は０を 指定する．（初期値 =5）
intpart	int	整数部の表示桁数を指定する．（初期値 =0）
plength	int	出力する桁数の制限値を指定する．（初期値＝10000）

*Switch 型（付録参照）において，"NODEF" とすると前回の設定が変更されない．

この他に，付録には示さなかったが，ファイルデータの高速入出力の目的で，Bfloat オブジェクトのバイナリ入出力を行う ifstreambin および ofstreambin クラスも同時に開発した。このクラスを利用すれば，例えば，テンポラリデータなどをディスクへ一時的に保存しておきたいような場合，抽出・挿入演算子よりも高速な入出力処理が可能である。

3．多倍長乗除算アルゴリズム

多倍長数では，加算や減算は乗算に比べてその処理時間を無視でき，また，除算はNewton法によって乗算に帰着できるから，乗算が最も重要な演算となり，これによってパッケージの性能が大きく左右される。乗算を実行する代表的な方法としては，古典的算法（以下，C法と略す），再帰アルゴリズム（R法），FFTアルゴリズム（F法）などがあり，それらがKnuthのテキスト[8]に詳述してある。C法の計算量は$O(n^2)$であり，R法は2のベキに従う算法のとき$O(n^{\log 3})$となる。また，F法は，乗算が畳み込み演算であることを利用したもので，計算量は$O(n \log n)$である。ただし，この方法は計算誤差を分離する必要があるため，他の2つの方法に比べ基数rをかなり小さくしなければならない（Gtype型がlong double型のとき$r=10^7$程度）。

C法は，基本データ型に対する乗算1回，加算2回およびキャリー処理が約$n^2/2$回繰り返され，他に比べてアルゴリズム的には劣っているものの，シンプルな算法ゆえ有効桁数が小さいところで最も速いことが知られている。さらに，次のような工夫をすると処理時間をもっと短縮でき，適用範囲を拡大することができる。本パッケージで採用している10進を基本にした基数を採用した場合，C法の算法の中でキャリー処理に案外と時間がかかる。今，(1)式の形式の多倍長浮動小数点数a, b, cを考え，それぞれの仮数部のj番目の項をa_j, b_j, c_jとする。このとき，積$c=a\times b$のj番目の項c_jは，

$$\tilde{c}_j=a_1b_j+a_2b_{j-1}+\cdots\cdots+a_jb_1 \qquad (j=1,2\cdots,n) \tag{8}$$

の値と下位からのキャリーとによって決まる。ここで，2．2節で述べたデータ型Gtypeの仮数部がℓ bitであれば，$\Delta j=\lfloor(2^\ell-1)/(r-1)^2\rfloor$個のMtype型同士の積の和を一つのGtype変数に格納できることに注目する。このことを利用すれば，(8)式の右辺をΔj毎に$w_j=\lceil j/\Delta j\rceil$個の部分和に分けて計算して，それぞれをGtype型作業変数に記憶しておけば，Gtypeがlong double型（$r=10^9$）のとき$\Delta j=18$，また，double型（$r=10^7$）では$\Delta j=90$であるから，最終的に，キャリー処理を高々（$2n+w_1+w_2+\cdots+w_n$）回程度にまで減らすことできる。ただし，乗算回数は同じであるため，アルゴリズム的にはKnuthの記述と

同等である。Knuthの記述と区別するために，この手法をIC法，Knuthの方をC法と呼ぶことにする。IC法とC法，双方をインプリメントして計算時間を測定した結果，nが小さいうちはオーバーヘッドの影響でさほど違いはなかったが，仮数部配列長n=10付近で3/4，n～20で1/2，最終的には1/4以下まで計算時間を短縮できることがわかった。本法は，基数が上限値よりも少し小さいことが本質的に重要であり，もし基数が上限値に設定されているとC法でしか対応できない。本パッケージでは，この意味もあって，上限値よりもわずかに小さい基数（表1）を採用している。IC法は，計算量を定数倍低減するだけであるが，測定結果からわかるように乗算時間を短縮するのにかなりの効果がある。新規性には乏しいものの本法の実用的効果について明記された文献は見当たらないため，ここで，その重要性をあえて強調しておきたい。

最近，Zuras[9]によってCとアセンブラを併用したプログラムを用いて上記3つのアルゴリズムの性能が詳細に検討されている。それによると，n=23まで$O(n^2)$アルゴリズムが，n=24～384ではR法が最も優れているようである。ただし彼は，基本的な処理をアセンブラで記述しており，また，$O(n^2)$アルゴリズムにKnuthと同様なC法を使用しているようであるから，我々のパッケージにはその結果をそのまま適用できない。そこで，Zurasと同様にIC，R，F法をインプリメントし性能比較実験を行った。その結果を図2に示す。なお，測定にはPC−9801BX2（IntelDX2™ODP付）を用い，IC法およびR法ではr=10^9，F法（基数2のFFTを使用）ではr=10^7とした。これからわかるように，有効桁3500桁（n～400程度に相当する）辺りでR法が，有効桁5000辺り（F法でn～700，他の2つはn～550）でようやくF法もIC法を逆転する。この傾向はEWS4800による測定でも同様であった。古典的算法が極めて優秀なのは，キャリー処理を低減したIC法を採用したからである。もしC法を用いれば，Zurasの結果と同様，250桁（n～30）付近でR法の方が高速になることを確かめている。科学技術計算で使用するのはほとんど1000桁程度までであるから，通常の利用形態ではIC法だけインプリメントしておけば十分であり，基本的に本パッケージもこれを採用している。しかしながら，超高精度計算についての利用も考慮して，F法のコードをマクロ定義の変更だけで追加できるように

はしている。

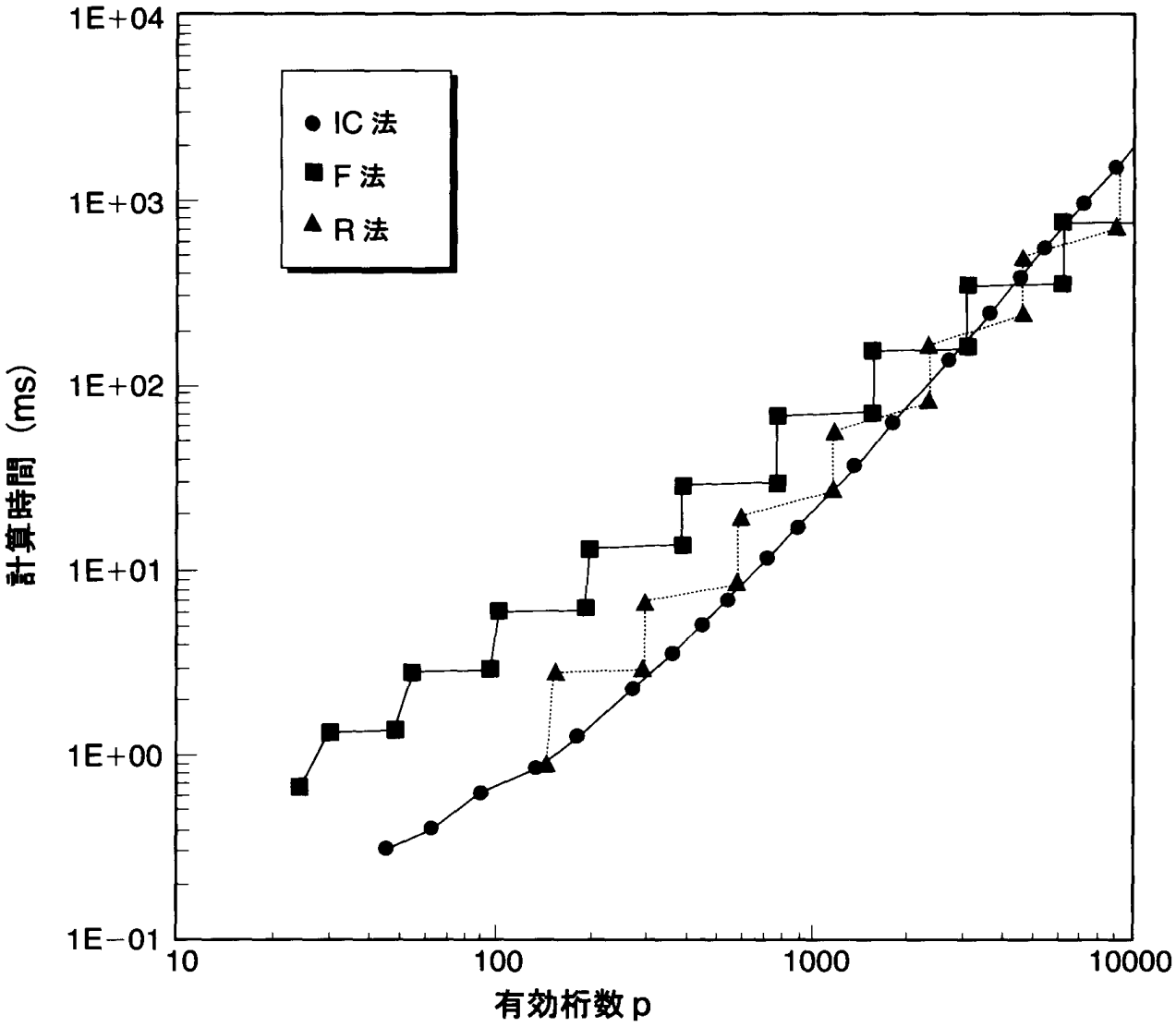

図２　乗算時間の計測結果

除算 *a*/*b* は，Newton 法によりまず逆数1/*b* を求め，最後に *a**(1/*b*)として求めている。一般に，逆数計算には時間がかかるため（理論的には乗算の３倍程度，*n* が小さいときはオーバーヘッドの影響でそれ以上になる），効率のよいプログラムを作成するには，できるだけ除算を含まないような工夫が必要である。例えば，繰り返しループ中で変化しない変数による除算があれば，まず，ループ外でその変数の逆数値を一度計算しておき，ループ内では逆数値との乗算の形式にしておいた方がよい。

４．初等超越関数の計算アルゴリズム

数学関数の開発には，一松，Smith および Brent らの文献を参考にした[1)2)10)~12)]。これらの中で特に，指数関数・対数関数・三角関数について，Bfloat クラスで使用した計算アルゴリズムを本章で述べる。一般に，初等超越関数の多倍長計算には，それらのベキ級数を使用する。現在のところ，ベキ級数を効率よく計算するには Smith の提案した Concurrent 級数（CS）アルゴリズム[1)11)]が最も優れているものと思われるため，本パッケージでもこれを全面的に採用した。

４．１　指数関数

指数関数は引数の定数倍で基底変換を容易に実行できるから，ここでは10^xのみ考察する。これを求めるには，まず，引数 x を整数部 x_i と小数部 x_f に分離し，そして

$$10^x=10^{x_i}\cdot 10^{x_f}=10^{x_i}\cdot e^y \qquad (\because\ y=x_f\log 10,\ 0<y<\log 10) \tag{9}$$

の関係を利用するのが通常のやり方である。(9)式による計算時間のほとんどは，最後の e^y を CS アルゴリズムで求めるのに費やされる。そこで，これに次のような改良を加えた。まず，任意の λ に対して，

$$z=x_f\log_\lambda 10 \tag{10}$$

とおき，z を整数部 z_i と小数部 z_f に分離する。このとき，10^x に対して，

$$10^x=10^{x_i}\cdot \lambda^{z_i}\cdot e^Y \qquad (\because\ Y=z_f\log\lambda,\ 0<Y<\log\lambda) \tag{11}$$

が成り立つから，もし $\lambda\to 1+0$ に設定すれば $Y\to +0$ となり e^Y のベキ級数計算を高速化できる。ただし，その代わりに $z_i\to\infty$ となって λ^{z_i} の計算量が増加するため，結局，適当な λ のときに全体の計算量が最小になる。(11)式の計算量を数値的に調べた結果，λ ～51/50=1.02辺りでほぼ最小になることがわかった。ただし，この評価実験では，λ^{z_i} の計算量をできるだけ低減するために，単純な数値で表現できる λ だけを対象にした。(10)式の改良によって，最適値 λ = 1.02のときに(9)式よりも約15～25% ほど計算時間を短縮することができた。

ところで，e^x のベキ級数計算では，

$$e^x=(e^\omega)^{2^m} \qquad \because\ \omega=x2^{-m} \tag{12}$$

の関係を利用して引数を小さくした後，e^{ω}を計算し，最後にそれをm回２乗して元の値を求めた方が効率がよい。このときも，e^{ω}と２乗計算全体の計算量を極小にする最適なmが存在する。最適値mは引数xに依存し，その評価にはベキ級数の計算量を決定する，

$$\omega^j/j! \leqq 10^{-p} \qquad (j \in \boldsymbol{N}) \tag{13}$$

を満たす最小のj(=j_mとする）を知る必要がある。ここで，pは有効桁数である。有効桁が大きくなると，jを逐一変化してj_mを調べてから最適値mを探索するような方法では明らかに能率が悪くなる。そこで，(13)式の不等号を等号に，jをyに置き換え，さらに階乗をStarlingの公式で近似すれば，(13)式から，

$$y(\log\omega+1)-(y+1/2)\log y+p\ \log 10-\log(2\pi)^{1/2}=0 \tag{14}$$

の形式の関係を導けるから，これをy（$y \in \boldsymbol{R}$とする）に関してNewton法で解けば，わずか数回の反復でj_mの近似値yを得ることができる。このとき，j_mは，$j_m=\lceil y \rceil$で与えられる。ここで，この計算には多倍長計算は必要ないことを注意しておく。本パッケージでは，ベキ級数を計算する場合には全てこの方法を用いている。

４．２　対数関数

対数関数$\log x$は，その近似値L=_ftobf(log(bftof(x)))と補正量

$$\delta x=(e^L/x-1)/(e^L/x+1) \tag{15}$$

とを用いれば，

$$\log x=L-\log\{(1+\ \delta x)/(1-\ \delta x)\} \tag{16}$$

と表せる。δxはマシンイプシロン程度になるから，(16)式の右辺第２項をベキ展開した級数は急速に収束することが期待される。これは平山のパッケージ[5]に採用されている手法である。Smithは対数関数を$e^y-x=0$の解としてNewton法で解いているが，それと(16)式とを比較したところ，(16)式はNewton法よりもx=2のとき3/4程度（x=10では1/2程度であった）の時間で済むことがわかった。

4.3　三角関数

正弦関数 $\sin x$ の算法は以下の通りである。引数 x は，象限三角関数を経由して区間 $[0, \pi/4]$ に還元されているものとする。さらに，$\sin x$ ではなく，引数を縮小した $\sin(x\alpha^{-m})$ のベキ級数を求め，
最後に，

$$\sin(3x)=\sin x(3-4\sin^2 x) \tag{17}$$

$$\sin(5x)=\sin x\{5-20\sin^2 x+16(\sin^2 x)^2\} \tag{18}$$

などの関係を m 回使用して元の値 $\sin x$ を求めるという手順で計算している。ここで，$m \geqq 0$，α は $\alpha=3,5,7,\cdots$ のいずれかとする。m や α には全体の計算量を最小にする最適値があり，さらにそれらは引数により変化する。プログラムでは動的にその最適値を決定している。ただし，指数関数の場合と異なり，値の復元に使用する(17)(18)式の計算量が大きいため（それぞれ，乗算の計算量の約1.5倍，2倍程度になる[*1]），三角関数では引数縮小の効果は小さかった。特に，$\alpha=7$ になると（この場合の復元式は乗算の約3倍の計算量），通常の有効桁範囲（＜1万桁）ではほとんど効果がないことが判明したため，$\alpha \geqq 7$ についてはプログラムでは考慮していない。

余弦関数も正弦関数と同様に計算している。ただし，$x=0$ 近傍での情報落ちを考慮して，

$$\cos_m(x)=\cos(x)-1 \tag{19}$$

のベキ級数を計算し，最後に $1+\cos_m(x)$ とした。$\cos_m$ 関数に対しても(17)式等と同じ計算量を持つ復元式を構成できる。正接関数は，正弦と余弦関数両者を求めて定義通りに処理している。

逆正接関数は，そのままではベキ級数の収束が非常に遅く使いものにならないため，対数関数の場合と同様に，近似値 T=_ftobf(atan(bftof(x)))を利用し，

[*1] 計算量の見積もりには，$O(n^2)$ アルゴリズムを使って計算する場合，2乗演算は乗算の約半分の手間で済むという事実を使っている。Bfloat クラスには，このことを利用した2乗演算専用関数 square も提供している。

$$\tan^{-1}x=T+\tan^{-1}\left(\frac{x-\tan T}{1+x\tan T}\right) \tag{20}$$

の関係を使って計算した。右辺第2項のベキ級数の収束は極めて速い。上式は，Smithが採用したNewton法よりも高速であることを確かめている。(20)式も平山のパッケージ[5]で採用されている。

5．性能評価

有効桁数p(10進）をパラメータにして，開発したパッケージの乗算や数学関数等の処理時間を測定した結果を表3に示す。比較のために，筆者の手元にあるREDUCE（Ver 3.2，PC版）[13]とUBASIC（Ver 8.7，32bitコード使用）[3]による結果も示しておく。REDUCEは数式処理で著名なソフトウェアであるが，今回は，これに添付されている多倍長実数演算パッケージbigfloatを使用した。一方，UBASICは，整数論への応用を指向して開発された言語で[14]，98やDOS/V等のパソコン専用のアセンブラで記述され，通常のBASICに準じた仕様を持っている。これには2600桁までの固定小数点実数がサポートされており，処理速度の計測にはこれを使った。これらと浮動小数点方式である本パッケージとを単純には比較できないが，できるだけ計算条件が同一になるように，一部を除いて結果の値に整数部が生じないような引数を選んで測定した。計測に用いた計算機はPC−9801BX2（486SX−25MHz+ODP−50MHz）である。ただし，本パッケージについてはNEC製EWS4800/330（R4400PC−67MHz）による結果も示してある。

表3より，REDUCEは計算時間が他の二つに比べ極端に長く，数値計算には不向きであることがわかる。次に，UBASICと本パッケージとを比べると，有効桁の小さいところでは前者が非常に速いが，有効桁が大きくなると特に数学関数ではそれが逆転している。この理由は，UBASICがアセンブラで記述してあることによる。すなわち，UBASICはビット処理を恐らく多用しているで

あろうから，オーバーヘッドが少なく，有効桁の小さいところではそれが強く影響している。一方，有効桁が大きいところでは，UBASICはインプリメントの容易な単純なアルゴリズムを採用せざるを得ないから（アセンブラで記述する以上当然であろう），採用しているアルゴリズムの差が逆転の原因になっているものと思われる。この結果より，本パッケージは有効桁の小さいところではオーバーヘッドが大きいが，それ以外では，アセンブラで記述されているUBASICと比べても遜色がなく，むしろ優秀な処理能力を持つことがわかった。さらに，本パッケージは移植性の良さも一つの特徴である。それゆえ，もし高速処理が必要であれば，よりCPUパワーの強力なマシンへ移植し，それで実行すればよい。この点に関しては現在パソコンのMS−DOS上でしか動作していないUBASICよりも明らかに優れている。最後に，平山のパッケージ[5]についても比較したところ，同一条件にて，本パッケージの方が数倍速いことを確認した。

表3　各種演算の処理時間の実測結果

演算††	EWDUCE†（単位=sec）			UBACIC†（単位=msec）			本パッケージ††† （単位 =msec）		
	p=63	p=270	p=1233	p=63	p=270	p=1233	p=63	p=270	p=1233
$x*y$	0.02	0.28	5.4	0.102	1.14	23.0	0.31(0.06)	2.00(0.35)	28.3(4.7)
$1/x$	0.08	0.92	19.4	0.16	1.7	32.2	1.47(0.32)	6.97(1.94)	116.7(14.6)
$\sqrt{x}$	0.11	1.42	27.5	0.57	4.5	88.0	2.31(0.53)	12.25(3.06)	200.5(26.8)
e^x	1.7	63.0	4229.0	1.98	72.3	5010.0	8.05(1.60)	52.8 (8.68)	922.5(155.6)
$\log x$	0.52	19.0	1484.0	2.6	121.0	10440.0	15.36(3.11)	90.9 (16.11)	1230.0(200.5)
$\sin x$	0.43	16.5	961.0	1.4	46.5	3040.0	6.73(1.37)	48.5 (7.87)	857.8(139.8)
$\tan^{-1}x$	0.76	28.0	1859.0	1.3	55.0	4640.0	21.9 (4.47)	137.7 (24.34)	2289.0(377.3)

† REDUCEでは，有効桁数pはprecision命令で指定した値である。UBASICでは，$p \fallingdotseq 4.82w$となるように小数部のワード数wをpoint命令で設定している。1ワードは約10進4.82桁である。

†† 引数は，乗算x=7/13,y=9/14，逆数x=13/7，対数関数x=20/13，それ以外はx=7/13とした。

††† 括弧内の数値はEWS4800での計測値である。

6．応用例

6．1　簡単な多倍長プログラミング例

再帰処理によって階乗計算を実行する簡単な多倍長プログラムの例を図３に示す。図３(a) が通常の倍精度データ型を使ったプログラム例で，(b) はそれを約50桁の多倍長計算用に書き直したものである。下線部が変更箇所を示しており，演算子の多重定義のおかげでわずかな変更だけで済む。なお，実行例 (c) に示す123456! などは通常のデータ型では計算できないが，Bfloat クラスでは問題なく計算でき，巨大な数値を扱うような場合にも本クラスは有効である。

通常，本パッケージを利用すれば図３のように多倍長プログラムへの変更は比較的容易であるが，通常プログラム中に計算精度に関する記述があると修正にはかなりの注意を要する。さらに，計算精度が陰的に表現されているプログラムだと字面だけ見て変更することは不可能で，計算原理そのものの理解が必要になるため，ソースコードに陰的記述を含む既存のライブラリプログラム等の多倍長化には多大の労力を要する。

```
 1:
 2: #include <iostream.h>
 3: double fact(long n) {
 4:         if(n==0)
 5:                 return 1;
 6:         else
 7:                 return n*fact(n-1);
 8: }
 9: void main(void) {
10:
11:        double y;  long n;
12:        cout << "n = "; cin >> n;
13:        y=fact(n);
14:        cout << n << "!=" << y << '¥n';
15: }
```

(a)通常精度（倍精度）計算プログラム

```
#include "Bfloat.h"
#include <iostream.h>
Bfloat fact(long n) {
        if(n==0)
                return ltobf(1);
        else
                return n*fact(n-1);
}
void main(void) {
        Bfloat::SetPrecision(50);
        Bfloat y;  long n;
        cout << "n = "; cin >> n;
        y=fact(n);
        cout << n << " !=" << y << '¥n';
}
```

(b) 多倍長計算プログラム

```
123456!= 0.26040 69904 92913 78729 51393 05609 26568 81827 32704 09503 018577e574965
```

(c)多倍長プログラム(b)の一実行例（計算時間:EWS4800にて4.3s）

図３　再帰手続きによる階乗 $n!$ 計算プログラム

6.2　誤差評価への応用

一般に数値計算の理論誤差解析，特に実用上要求される定量性のある理論誤差を導出するのは非数学系の研究者にとってはかなりの困難を伴う。そのため，多くの場合は数値実験によって直接的に誤差評価を行なっているのが実状である。しかし，これでは複数の誤差要因が絡んでいるような場合に最終誤差を支配している要因を特定するのが非常に難しい。本節では，この実験的誤差評価の持つ難点を解消する一つの方法として，本パッケージを適用した例を述べる。例題としては，筆者が扱っている表面電荷法による数値電界計算の誤差評価問題を考える。本題に入る前に，まず，回転対称場に対する表面電荷法の原理を簡単に述べておく。電磁気学の基本定理によれば，任意点のポテンシャル$\phi(r,z)$は，境界上の表面電荷密度をσ，ラプラス方程式の Green 関数をgとすると，

$$\phi(r,z)=\int_0^1 \sigma(u)g(r,z;u)du \tag{21}$$

のように表せる。ここでuは境界上の規格化座標を表す。(21)式において，境界ポテンシャルϕを既知，境界上のσを未知数と考えれば，最終的に電界の境界値問題は，核関数gに関する第1種の Fredholm 型積分方程式に帰着できる。里らは，この積分方程式に対して，区間［0,1］で定義されたチェビシェフ多項式

$$T_j(u)=\cos\{j\cos^{-1}(1-u)\} = \sum_{k=0}^{j} c_{jk}u^k \tag{22}$$

$$\because \quad c_{jk}=(-1)^k 4^k \frac{j}{j+k}\binom{j+k}{j-k} \tag{23}$$

を用いて，$\sigma(u)$を

$$\sigma(u)=\sum_{j=1}^{n}\sigma_j T_{j-1}(u) \tag{24}$$

と$n-1$次多項式近似し，(22)(24)式を利用することにより電界の高速解法に成功している[15]。そのアルゴリズムでは，

$$[I_j]=[c_{jk}][J_k] \tag{25}$$

$$\because\ [c_{jk}]:n\times n\ \text{下三角行列},\qquad J_k=\int_0^1 u^{k-1}g(u)du$$

の行列計算を行ってI_jを求め，そのI_jを係数行列，σ_jを未知数とした連立一次方程式を解くという手順をとる。この手法では，nを大きくするほど高速化の効果が顕著になるが，(23)式の係数c_{jk}にjと共に指数関数的に増大する項があるため，nが大きくなると(25)式右辺の行列計算で激しい桁落ちが発生して，この影響でnが制限される。また，nが大きくなると連立方程式の係数行列が特異に近づき解の精度が低下するため，これでもnは制限される。他の誤差要因はアルゴリズムの工夫により取り除けるため[16)17)]，量子化誤差を除くと，これら二つの要因が混合された誤差が最終解には反映されるようになる。

それでは，(25)式右辺の行列計算と係数行列の近特異性のどちらが最終誤差に最も影響し，高速化を阻害している本質的な因子はどちらであるか，という問題について考えてみる。このためには，片方の要因を多倍長計算によって除去してしまうのが単純である。まず，通常の倍精度計算プログラムで計算した時の電荷密度の誤差の振る舞いを調べ，そして，次に(25)式右辺の行列計算部分だけを十分な精度で多倍長計算して同様な計算を行った（変更はプログラムを数行修正するだけで済む）。半径９㎜の二つの導体球を中心間距離20㎜において，それぞれに１と０の電位を与え，1V 導体球上の近似次数ｎを変化させて電荷密度の最大相対誤差をプロットしたのが図４である。n=20までは量子化誤差の特徴である指数関数的な誤差の減少が認められるが，n>20で急速に誤差が増大し，この傾向は(25)式右辺を倍精度型・多倍長型のどちらでもほとんど変わっていない。このことは，最終解の誤差には第二の要因の方が支配的

であることを示唆しており，最終的に，里らのアルゴリズムの高速化を阻害する原因は係数行列の近特異性にあると結論できる。

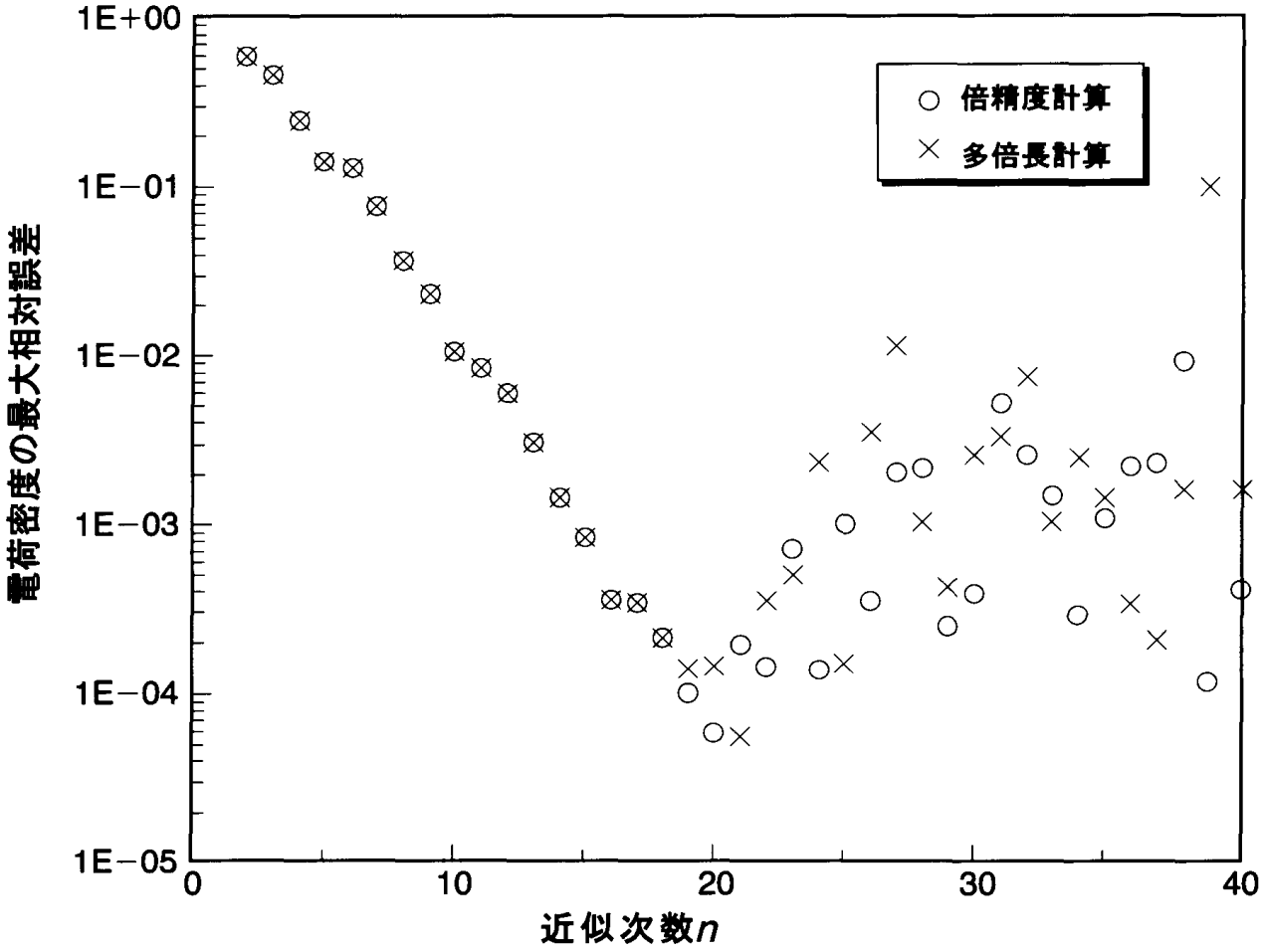

図４　倍精度・多倍長型で計算したときの電荷密度誤差の挙動

異なる誤差要因が複雑に絡んでいるような問題に対して，多倍長計算を利用すれば問題の分析が容易になることを一つの例を使って説明した。こういった分析を行うには，通常のプログラムを基本にして，それを少し変更するだけで対処できるのが望ましい。例えば，通常プログラムに大幅な変更が必要であると，その労力も大変であるが，気付かないうちに測定条件が変わってしまっているといった危険性もでてくる。本パッケージの多重定義の機能は，このような点で非常に優れたものであるといえる。

6.3　関数値の精度保証計算への応用

本節では，前節の誤差要因の除去の考え方をさらに推し進めた応用例について述べる。数値シミュレーションシステム等の開発段階において，既存の手法では正しく計算できない数学関数が含まれていると，その関数の数値誤差の影響でプログラム全体が正しく動作しないという状況に陥ることがある。このような場合に，とりあえず既存のアルゴリズムのままでも関数値が十分な精度を確保できる簡単な手段があれば，システムの開発期間をかなり短縮できるはずである。また，当然ながら，これは前節で述べた実験的な誤差評価にも役立つ。

今回，関数値の精度を自動的に保証するツールを多倍長クラスを使って実現した。ただし，対象にした関数は，戻り値が一つの普通の意味の関数のみである。その実現には特別なことは何も行っておらず，ただ，「有効桁 p_1 で計算した結果 R_1 と有効桁 $p_2(>p_1)$ で計算した結果 R_2 とが上位 n 桁一致すれば，R_2 は約 $n+(p_2-p_1)$ 桁正しい」[18] という基本的な事実を利用しているだけである。今回は，任意形式の関数に対応できるように，基本処理を担当する REfreeBase 基本抽象クラスだけ開発した。任意の関数の計算には，適切な派生クラスを利用者が作成して対応する方式とした。REfreeBase クラスには，REelimination, SetTargetPrecision, REFresult 関数等が公開メンバとして，また FuncValue が純粋仮想関数として定義してある。REelimination が最も重要な関数で，これに派生クラスのポインタを渡せば，ポリモーフィズム機構により適切な派生クラスが選択され，その派生クラスで指定された対象関数（派生クラスの FuncValue 関数内でコールされる関数がこれに相当する）の値が自動的に目標精度で計算される。REelimination 関数は，有効桁長を可変にしながら FuncValue 関数をコールし，その結果が目標精度を満足したかどうかを上記の単純な原理を利用して確認しつつ処理を行っている。そして，REelimination 自体は，もし有効桁長が探索上限値（規定値は約１万桁）以前に目標精度を達成すれば１，そうでなければ０を戻す。REelimination 関数が計算に成功すれば，その結果は Bfloat 型オブジェクトの形式で保持される。その抽出には REFresult 関数を使用する。また，SetTargetPrecision 関数で任意の目標精度を

設定できる。

int 型と double 型の２つ引数を持ち，戻り値 double 型の関数に対する派生クラスの定義の一例を図５に示す。メソッドとしてコンストラクタと自動精度保証計算用関数 Calc が定義してある。コンストラクタで対象関数を指定し，それと同時に，関数値の目標精度（有効桁数）を16桁に設定している。この目標精度は，対象関数の戻り値の型である double 型の精度に合わせた。Calc 関数は，引数をクラスデータに代入した後（このデータは FuncValue 関数で使われる），REelimination 関数によって目標精度の関数値を計算する。そして，最後に，REFresult 関数で取り出した Bfloat 型の関数値を bftof 関数で倍精度データ型へ変換して，それを Calc 関数の戻り値としている。

```
typedef Bfloat (*TFunPtr)(int,double);
class REfree : public REfreeBase {
private:
     TFunPtr pfun;  int arg1;  double arg2;  // クラスデータ(関数ポインタと引数)
     Bfloat FuncValue(void)                  // 基本クラスからの対象関数呼び出し用.この定義は必須.
       { return (*pfun)(arg1,arg2); }
public:
     REfree(TFunPtr Name) : REfreeBase() {   // コンストラクタ
         pfun=Name;                          // 対象関数の指定
         SetTargetPrecision(16);             // 目標有効桁(倍精度データ型)の設定
     }
      double Calc(int a1, double a2) {       // 倍精度データ型用精度保証計算実行関数
         arg1=a1; arg2=a2;                   // 引数のセット
         if( !REelimination( this ) )        // 自動精度保証計算
           cerr << "探索上限に達した¥n";
         return bftof( REFresult(*this) );   // 関数値の抽出と倍精度型への変換
     }
};
```

図５　double (*)(int,double)型関数に対する派生クラスの定義例

```
double func(int,double);
void main(void) {
     double y,x;  int n;
               .
               .
               .
     y=func(n, x);
               .
               .
               .
}
double func(int n, double x){
               .
               .
               .
}
```

(a)通常精度プログラム

```
Bfloat func(int,double);  // 対象関数の戻り値を
                             Bfloat型へ
void main(void) {
  double y,x;  int n;
  REfree AF(func);        //①対象関数をfuncとする
                .
                .
                .
  y=AF.Calc(n,x);         //②自動精度保証計算の実行
                .
                .
                .
}
Bfloat func(int n, double arg) {
    Bfloat x=_ftobf(arg);      // 実数引数の多倍長化
                .
                .              // 多倍長プログラム化
                .
}
```

(b)関数 func に対する精度保証型計算プログラム

図６　関数値の自動精度保証型計算プログラム化の一例

図６(b)は，図５の派生クラスを利用して，図６(a)の通常プログラム中の func 関数の精度を保証するように変更したプログラムである。まず，①で対象関数 func が REfree 派生クラスオブジェクトへ登録され，②で16桁の精度が保証された関数値が y に代入される。通常プログラム中の func 関数は全て②のように修正すればよい。ただし，func 関数およびそれから呼び出される関数については全て多倍長化しておかなければならない。本ツールを使用すれば，派生クラスの定義と対象とする関数ルーチンの多倍長化だけで，通常プログラム自体はほとんど修正することなく特定の関数の精度が保証されたプログラムへ移行できる。適用範囲が戻り値一つの関数だけに限定されてはいるもののプログラム開発や誤差評価等において非常に有効なツールになるものと思われる。

７．あとがき

多倍長浮動小数点クラス Bfloat を開発し，そのアルゴリズム，性能および応用例について詳細に述べた。今後は，これを利用した応用分野についてさらに

検討を進めていく予定である。多倍長計算は，本質的に重要な場合もあるが，その多くは誤差評価等の場面で使用する補助的な道具としての利用が圧倒的であろう。道具の利用は簡単であることが望ましく，本パッケージを用いればそれがある程度は満足される。しかし，６．３節で述べたように特定の関数値の精度を保証するような問題では，対象の関数を変えながら最終解の誤差の性質を調べるケースも多くあると思われ，こうなると，補助的な道具という感覚があるがために，変更作業を非常に面倒に感じるようになる。この障壁を低くするには，やはり変換を自動化する能率の良いプリプロセッサの開発が不可欠であり，現在，これについての検討も進めている。

本論文では，原始的な手法を使って最終結果を指定した精度で自動的に計算するREfreeBaseクラスについても述べた。このクラスは，上でも述べたように数値計算用の補助的ツールとしては十分に意義のあるものと考えている。しかし，計算機の普及に伴い数値計算を専門としないユーザもプログラムや表計算ソフトウェアなどで数値（実数値）計算を行う機会が増加している状況を考慮すると，今後，REfreeBaseクラスのような精度保証の考え方をより本質的なものとして位置づけていくべきであろう。全てのソフトウェアに対してそれが必要だとは思われないが，一般のユーザが計算精度で悩んだり，あるいは誤った結果を得ることがないように，少なくともユーザからの要求があれば（できればオプションスイッチをオンにするというような簡単な操作だけで），そういった処理を効率よくかつ自動的に実行する機構を何らかの形で今後実現していくべきであると思われる。

謝辞

多倍長パッケージを送付いただき，その使用をご許可くださった神奈川工科大学平山弘博士に感謝いたします。

参考文献

1）R. P. Brent: “A Fortran Multiple-Precision Arithmetic Package”, ACM Trans. Math. Software., vol. 4, no. 1(1978) pp. 57-70.
2）D. M. Smith: “A FORTRAN Package for Floating-Point Multiple-Precision Arithmetic”, ibid, vol. 17, no. 2(1991) pp. 273-283.
3）木田祐司：“多倍長計算用BASIC UBASIC86”，日本評論社（1994).
4）M. A. Ellis and B. Stroustrup: “The Annotated C++ Reference Manual”, AT&T Bell Lab., Murray Hill, New Jersey (1990).
5）平山　弘：“C++言語による高精度計算プログラムパッケージの作成”，日本応用数理学会平成6年度年回講演予稿集，pp. 252-253.
6）二宮市三：“科学技術計算への二つの提案”，応用数理，vol. 2, no. 1(1992) pp. 2-8.
7）二宮市三・秦野やす世：“数学ライブラリNUMPAC”, 情報処理, vol. 26, no. 9 (1985) pp. 1033-1042.
8）D. E. Knuth: “The Art of Computer Programing Vol. 2/Seminumerical Algorithms”, Addison-Wesley, Reading, Mass. (1981). 邦訳：中川圭介訳，“準数値算法”，サイエンス社（1991).
9）D. Zuras: “More on Squaring and Multiplying Large Integers”, IEEE Trans. Computers, vol. 43, no. 8 (1994) pp. 899-908.
10）一松　信：“初等関数の数値計算”，教育出版，(1990).
11）D. M. Smith: “Efficient Multiple-Precision Evaluation of Elementary Functions”, Math. Computation, vol. 52, no. 185 (1989) pp. 131-134.
12）R. P. Brent: “Fast Multiple-Precision Evaluation of Elementary Functions”, J. ACM, vol. 23, no. 2 (1976) pp. 242-251.
13）A. C. Hearn: “REDUCEユーザーズマニュアル”，戸島熙訳，マグロウヒル出版 (1991).
14）木田祐司・牧野潔夫：“UBASICによるコンピュータ整数論”，日本評論社（1994).
15）里　周二・青柳浩邦・澄川俊雄・本多正己：“高速表面電荷法”，電気学会論文誌A，vol. 101. no. 9 (1981) pp. 455-462.
16）内川嘉樹・大江俊美・後藤圭司：“表面電荷法の改良”，電気学会論文誌，vol. 101, no. 5 (1981) pp. 263-270.
17）津曲　隆：“第2種円環関数の数値計算に適した新しい公式について”，日本応用数理学会論文誌，vol. 2, no. 3 (1992) pp. 177-191.
18）例えば，伊理正夫・藤野和建：“数値計算の常識”，共立出版（1985) pp. 20-25.

付録. Bfloat クラスヘッダファイル（一部）[*1]

```
  ////////////////////////////  積 g のデータ型の設定（(6)式参照）  ////////////////////////////////
            #define Gtype_Ldouble 1  // long double型（80bit）を使用できる処理系     1
                                     //              〃            できない処理系   0
  ////////////////////////////////////// 設 定 終 了 ////////////////////////////////////////////////
typedef long int  Mtype;                                        // 仮数部配列のデータ型
#if Gtype_Ldouble
        typedef long double Gtype;                              // g のﾃﾞｰﾀ型（表1,(6)式参照）
        const int RadixP=9;                                     // 基数 r=10^k の定義. k=RadixP
#else
        typedef double Gtype;
        const int RadixP=7;
#endif

class Bfloat {
public:
        enum Switch { ON, OFF, NODEF };
private:
        int         Sgn;                                        // 符号 S
        long int    Expo;                                       // 指数部 m
        Mtype*      Manti;                                      // 仮数部配列の先頭ポインタ
        int         Length;                                     // オブジェクトの仮数部配列長 n
        int         ZeroPos;
        static int BfltLength;                                  // Bfloatクラスの仮数部配列長
                    .
                    .
                    .
public:
// ★ コンストラクタ・デストラクタ
        Bfloat (void);                                          // デフォールトコンストラクタ
        Bfloat ( const Bfloat& );                               // コピーコンストラクタ
        ~Bfloat (void) { delete[] Manti; }                      // デストラクタ
// ★ 精度設定・取得
        static void SetPrecision (int);                         // Bfloatｸﾗｽの有効桁数設定
        static int  GetPrecision (void);                        // Bfloatｸﾗｽの有効桁数取得
        friend int  GetPrecision (const Bfloat& x);             // Bfloatｵﾌﾞｼﾞｪｸﾄの有効桁数取得
// ★ 代入演算子
        Bfloat& operator=( const Bfloat&  );                    // y=x
// ★ 単項演算
        Bfloat operator+(void) const;                           // +x
        Bfloat operator-(void) const;                           // −x
// ★ 二項演算（Bfloat・long間演算は省略した）
        Bfloat& operator+=(const Bfloat&);                      // y+=x
```

[*1] long·Bfloat 間処理や FFT 等の記述については省略している。コメント中，x と y は Bfloat オブジェクト，n は整数値を表す引数とする。

```
        Bfloat& operator-=(const Bfloat&);                  // y-=x
        Bfloat& operator*=(const Bfloat&);                  // y*=x
        Bfloat& operator/=(const Bfloat&);                  // y/=x
        friend Bfloat operator+(const Bfloat&, const Bfloat&);   // x+y
        friend Bfloat operator-(const Bfloat&, const Bfloat&);   // x-y
        friend Bfloat operator*(const Bfloat&, const Bfloat&);   // x*y
        friend Bfloat operator/(const Bfloat&, const Bfloat&);   // x/y
// ★ 比較演算子（Bfloat・long間比較演算子は省略した）
        friend Boolean operator< (const Bfloat&, const Bfloat&);  // x<y
        friend Boolean operator<=(const Bfloat&, const Bfloat&);  // x≦y
        friend Boolean operator> (const Bfloat&, const Bfloat&);  // x>y
        friend Boolean operator>=(const Bfloat&, const Bfloat&);  // x≧y
        friend Boolean operator==(const Bfloat&, const Bfloat&);  // x=y
        friend Boolean operator!=(const Bfloat&, const Bfloat&);  // x≠y
// ★ 入出力演算子
        friend ostream& operator<<(ostream&, const Bfloat&);      // ストリーム出力演算子
        friend istream& operator>>(istream&, Bfloat&);            // ストリーム入力演算子
        // グローバルＩ／Ｏパラメータ設定
        static void SetOutForm(Switch precision=OFF,              // 有効桁数表示/非表示
                               Switch truncate=ON,                // 末尾のゼロ値表示/非表示
                               Switch form=ON,                    // ﾌｫｰﾏｯﾄ/ｱﾝﾌｫｰﾏｯﾄ出力
                               int    space=5,                    // 指定桁毎の空白挿入
                               int    intpart=0,                  // 整数部表示桁数
                               int    plength=10000);             // 出力桁数制限値
        // ローカルＩ／Ｏパラメータ（マニピュレータ）
        friend ostream& precision(ostream&);                      // 有効桁数表示ON
        friend ostream& noprecision(ostream&);                    //       〃     OFF
        friend ostream& trunc0(ostream&);                         // 末尾ゼロ値の非出力
        friend ostream& notrunc0(ostream&);                       //      〃      出力
        friend ostream& form(ostream&);                           // フォーマット出力
        friend ostream& unform(ostream&);                         // アンフォーマット出力
        friend ostream& space  (ostream&,int);                    // 指定桁毎の空白挿入. space(n).
        friend ostream& intpart(ostream&,int);                    // 整数部表示桁数.     intpart(n).
        friend ostream& plength(ostream&,int);                    // 出力桁数制限.       plength(n).
// ★ 組込関数
        // ☆ 型変換関数
        int bftobf(Bfloat*) const;                                // Bfloat→Bfloat(異精度ｵﾌﾞｼﾞｪｸﾄ間)
        friend Bfloat atobf(const char*  );                       // string  →  Bfloat
        friend Bfloat ltobf(long);                                // long    →  Bfloat
        friend long   bftol(const Bfloat&);                       // Bfloat  →  long
        friend double bftof(const Bfloat&);                       // Bfloat  →  double
        friend Bfloat _ftobf(double);                             // double  →  Bfloat
#if Gtype_Ldouble
        friend long double bftolf(const Bfloat&);                 // Bfloat  →  long double
```

```
        friend Bfloat _lftobf(long double);                    // long double  →  Bfloat
#endif
        // ☆ 数学定数取得関数
        friend Bfloat m_pi  (int inv=1);                       // 円周率π        (inv<0で逆数)
        friend Bfloat m_pi_4(int inv=1);                       // π/4             (    〃     )
        friend Bfloat m_e   (int inv=1);                       // 自然対数の底e   (    〃     )
        friend Bfloat m_ln10(int inv=1);                       // 自然対数ln(10)  (    〃     )
        friend Bfloat m_ln2 (int inv=1);                       // 自然対数ln(2)   (    〃     )
        // ☆ 符号・指数・仮数部参照
        friend int    sign(const Bfloat&);                     // ｵﾌﾞｼﾞｪｸﾄの符号取得
        friend long   exponent(const Bfloat&);                 //      〃      指数部取得
        Mtype operator[](int) const;                           // ｵﾌﾞｼﾞｪｸﾄの i 番目の仮数部取得
        // ☆ 符号関係
        friend int    changesign(Bfloat*);                     // 符号変換(+ ⟷ -)
        friend int    copysign(Bfloat*,const Bfloat&);         // 符号コピー(x=y.Sgn*|x|)
        friend Bfloat fabs(const Bfloat&);                     // 絶対値
        // ☆ 逆数と２乗
        friend Bfloat inverse(const Bfloat&);                  // x^-1
        friend Bfloat square(const Bfloat&);                   // x^2
        // ☆ 整数化関数
        friend Bfloat ceil (const Bfloat&);                    // 切り上げ
        friend Bfloat floor(const Bfloat&);                    // 切り下げ
        friend Bfloat round(const Bfloat&);                    // 四捨五入
        friend Bfloat modf (const Bfloat&, Bfloat*);           // 整数部・小数部分離
        // ☆ 剰余関数
        friend Bfloat fmod(const Bfloat&,const Bfloat&);       // x=k·y+f を満たす f を返す*1.
        friend Bfloat fmod(const Bfloat&,long);                // x=k·n+f       〃
        // ☆ べき乗関数
        friend Bfloat pow(const Bfloat&,long);                 // x^n
        friend Bfloat pow(const Bfloat&,const Bfloat&);        // x^y
        // ☆ 平方根
        friend Bfloat sqrt (const Bfloat&);                    // 平方根
        friend Bfloat isqrt(const Bfloat&);                    // 平方根の逆数
        // ☆ 指数関数
        friend Bfloat exp(const Bfloat&);                      // e^x
        friend Bfloat exp2(const Bfloat&);                     // 2^x
        friend Bfloat exp10(const Bfloat&);                    // 10^x
        friend Bfloat exp_1m  (const Bfloat&);                 // exp_1m(x)=e^x-1
        friend Bfloat exp2_1m (const Bfloat&);                 // exp2_1m(x)=2^x-1
        friend Bfloat exp10_1m(const Bfloat&);                 // exp10_1m(x)=10^x-1
        // ☆ 対数関数
        friend Bfloat log  (const Bfloat&);                    // log_e(x).  自然対数
        friend Bfloat log2 (const Bfloat&);                    // log_2(x)
        friend Bfloat log10(const Bfloat&);                    // log_10(x).  常用対数
```

*1 k は整数，f は$0 \leqq f < y$ を満足する Bfloat オブジェクトとする。

```
    friend Bfloat log_1p  (const Bfloat&);                  // log_1p(x)=log_e(1+x)
    friend Bfloat log2_1p (const Bfloat&);                  // log2_1p(x)=log_2(1+x)
    friend Bfloat log10_1p(const Bfloat&);                  // log10_1p(x)=log_10(1+x)
    // ☆ 双曲線関数
    friend Bfloat sinh (const Bfloat&);                     // sinh(x)
    friend Bfloat cosh (const Bfloat&);                     // cosh(x)
    friend Bfloat tanh (const Bfloat&);                     // tanh(x)
    friend Bfloat atanh(const Bfloat&);                     // tanh^-1(x)
    // ☆ 三角関数
    friend Bfloat sin  (const Bfloat&);                     // sin(x)
    friend Bfloat cos  (const Bfloat&);                     // cos(x)
    friend Bfloat tan  (const Bfloat&);                     // tan(x)
    friend Bfloat atan (const Bfloat&);                     // tan^-1(x)
    friend Bfloat atan2(const Bfloat&,const Bfloat&);       // tan^-1(x/y)
    // ☆ 象限三角関数 f_HP(x)=f(π/2 x)
    friend Bfloat sinHP  (const Bfloat&);                   // sin_HP(x)
    friend Bfloat cosHP  (const Bfloat&);                   // cos_HP(x)
    friend Bfloat tanHP  (const Bfloat&);                   // tan_HP(x)
    friend Bfloat atanHP (const Bfloat&);                   // tan_HP^-1(x)
    friend Bfloat atanHP2(const Bfloat&,const Bfloat&);     // tan_HP^-1(x/y)
};
```