数学定数に対する 2 進 BBP 型公式の計算における除算について

高橋 大介 [1]
[1] 筑波大学計算科学研究センター
e-mail : daisuke@cs.tsukuba.ac.jp

1　はじめに

π に対する Bailey–Borwein–Plouffe（BBP）公式 [1] は，前のビットをすべて計算することなく，π の特定のビットを計算できることが知られている．これまでに数学定数に対する BBP 型公式がいくつか提案されている．BBP 型公式の計算で最も時間を要するのはべき剰余であるが，級数の項の除算も無視できない時間を要する．この除算は exact division[2] を用いることで効率的に計算できることが知られているが，exact division を 2 進数で計算する場合，除数は奇数でなければならない．

本論文では，数学定数に対する 2 進 BBP 型公式の計算における除算に exact division を用いる方法を提案する．

2　BBP 型公式

BBP 公式 [1] は以下の式で表される．

$$\pi = \sum_{k=0}^{\infty} \frac{1}{16^k} \left(\frac{4}{8k+1} - \frac{2}{8k+4} - \frac{1}{8k+5} - \frac{1}{8k+6} \right) \tag{1}$$

16 進数で $n+1$ 桁目から始まる π の数桁を計算することを考える．これは $\{16^n \pi\}$ の計算と等価であることに注意する．ここで，$\{\cdot\}$ は小数部を示す．式 (1) から以下の式が得られる．

$$\{16^n \pi\} = \{4\{16^n S(1)\} - 2\{16^n S(4)\} - \{16^n S(5)\} - \{16^n S(6)\}\} \tag{2}$$

$$S(j) = \sum_{k=0}^{\infty} \frac{1}{16^k (8k+j)} \tag{3}$$

$$\begin{aligned} \{16^n S(j)\} &= \left\{ \left\{ \sum_{k=0}^{n} \frac{16^{n-k}}{8k+j} \right\} + \sum_{k=n+1}^{\infty} \frac{16^{n-k}}{8k+j} \right\} \\ &= \left\{ \left\{ \sum_{k=0}^{n} \frac{16^{n-k} \bmod (8k+j)}{8k+j} \right\} + \sum_{k=n+1}^{\infty} \frac{16^{n-k}}{8k+j} \right\} \end{aligned} \tag{4}$$

$\bmod(8k+j)$ が式 (4) の 1 番目の総和の分子に現れるのは，小数部だけを計算すればよいからである．式 (4) の 1 番目の総和の分子 $16^{n-k} \bmod (8k+j)$ は，バイナリ法を用いて効率的に計算することができる．式 (4) の 2 番目の総和の分子では，16 の指数が負になっているので，残りの項が浮動小数点演算の計算機イプシロンより小さくなるまで計算すればよいことになる．最終結果を 16 進数に変換することにより，$n+1$ 桁目から始まる π の数桁が得られる．

3　提案手法

Exact division を 2 進数で計算する場合，除数は奇数でなければならない．しかし，式 (4) の 1 番目の総和において分数の分母 $8k+j$ は，$j = 4, 6$ では偶数になるため，exact division を直接使うことはできない．その場合，2 進 BBP 型公式のべき剰余に Montgomery 乗算を適用する方法 [3] を用いることで，分数の分母を奇数にすることができる．1 ワードの被除数を 1 ワードの除数で割った m ワードの商の小数部の計算を固定小数点演算で行うアルゴリズムを Algorithm 1 に示す．

Algorithm 1 1 ワードの被除数を 1 ワードの除数で割った m ワードの商の小数部の計算

Input: x, N, r, μ such that $0 \le x < N$, $0 < N < \beta$, $\gcd(\beta, N) = 1$,
$r = (\beta^m \cdot x) \bmod N$, $\mu = N^{-1} \bmod \beta$

Output: $Q = \lfloor (\beta^m \cdot x)/N \rfloor$

1: **if** $r = 0$ **then**
2: **return** 0
3: $q_0 \leftarrow (-r \cdot \mu) \bmod \beta$
4: **for** j **from** 1 **to** $m - 1$ **do**
5: $q_j \leftarrow [\{(\beta - 1) - \lfloor (q_{j-1} \cdot N)/\beta \rfloor\} \cdot \mu] \bmod \beta$
6: **return** $Q = \sum_{i=0}^{m-1} q_i \beta^i$.

表 1. π の 16 進 10^{10} 桁目計算の実行時間（秒）

	従来手法	提案手法
Intel Core i3-8121U	2675.36	2543.13
Intel Xeon Gold 6230	464.41	378.61

式 (1) の BBP 公式を用いて π を計算する際に，exact division を用いた提案手法と exact division を用いない従来手法の実行時間を比較した．表 1 は，Intel Core i3-8121U と Intel Xeon Gold 6230 で π の 16 進 10^{10} 桁目を計算するのに必要な実行時間を示している．提案手法は従来手法に比べて Intel Core i3-8121U で約 1.05 倍, Intel Xeon Gold 6230 で約 1.23 倍高速であることが分かる．提案手法が従来手法よりも高速である理由としては，提案手法が整数除算命令の代わりに exact division を用いているためであると考えられる．また，Intel Core i3-8121U では，Intel Xeon Gold 6230 よりも提案手法と従来手法の性能差が小さくなっている．これは Intel Core i3-8121U が整数除算命令に要するサイクル数が少なくなっているためである．

4 まとめ

本論文では，数学定数に対する 2 進 BBP 型公式の計算における除算に exact division を用いる方法を提案した．提案する手法は，2 進 BBP 型公式の計算において除数が偶数である場合でも exact division を用いることを可能にした．性能評価の結果，提案手法が性能向上に有効であることを示した．

謝辞 本研究は，JSPS 科研費 22K12045 の支援によって行われた．

参考文献

[1] D. Bailey, P. Borwein, and S. Plouffe, “On the rapid computation of various polylogarithmic constants,” *Math. Comput.*, vol. 66, pp. 903–913, 1997.

[2] T. Jebelean, “An algorithm for exact division,” *J. Symb. Comput.*, vol. 15, pp. 169–180, 1993.

[3] D. Takahashi, “On the use of Montgomery multiplication in the computation of binary BBP-type formulas for mathematical constants,” *Ramanujan J.*, vol. 59, pp. 211–219, 2022.

Python 環境における多倍長精度基本線形計算モジュールの実装と性能評価

幸谷 智紀
静岡理工科大学
e-mail : kouya.tomonori@sist.ac.jp

1　初めに

多種多様な Python[1] 環境においては可変長精度の高性能計算ニーズが高い．にもかかわらず高速な多倍長精度演算環境の提供はごく限られている．我々は開発中の BNCmatmul[4] を Python 環境で実行できるよう，DLL 化とモジュール化を行い，BNCamPy パッケージを構築し，ほぼ C/C++ と同程度の性能を Python 環境に持ち込むことに成功している．本講演ではフリーで高速な多倍長精度数値計算環境の実装と性能評価の概要を示す．

2　BNCamPy の概要

我々が構築したソフトウェア構成を図 1 に示す．BNCmatmul を Python 環境で使用するため，マルチコンポーネント型多倍長精度 (DD(2 進 106bits 精度), TD(159bits), QD(212bits)) 演算を RDD.c(実基本演算), RCDD.c(複素基本演算) を介して Python から呼び出せるようにし，AVX2 を用いた高速基本線形計算機能も BNCamPy から使用できるようにしてある．

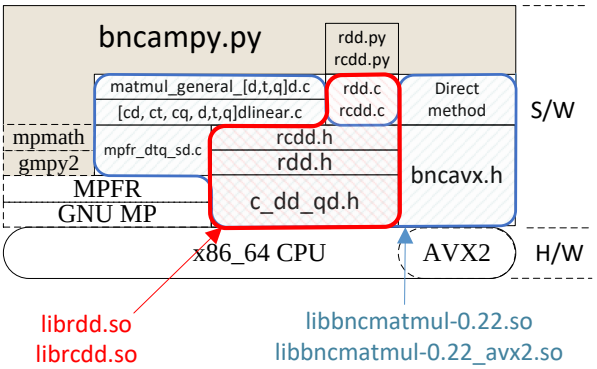

図 1. Python 環境における BNCmatmul のソフトウェア階層

Python 環境における多倍長精度数値計算ライブラリとしては mpmath[3] が著名であり，信頼性も高い．また，MPFR や MPC といった既存の高速な任意精度浮動小数点ライブラリを Python 環境下で使用可能にする gmpy2[2] の利便性と高速性は捨てがたい．従って，これら既存の Python パッケージと連携して使えるようにしておくことが望ましい．今のところテスト的実装であるが，互いにコンバートできることを確認している．

3　LU 分解のベンチマークテスト

下記の EPYC 環境下で実 LU 分解，複素 LU 分解のベンチマークテストを行った．

EPYC　: AMD EPYC 9354P 3.75GHz 32 cores, Ubuntu 20.04.6 LTS, Intel Compiler version 2021.10.0, MPLAPACK 2.0.1, GNU MP 6.2.1, MPFR 4.1.0, MPC 1.2.1, Pytho 3.8.10

使用した n 次元連立一次方程式は $A\mathbf{x} = \mathbf{b}$ とした．ここで $A \in \mathbb{R}^{n\times n}$, $\mathbb{C}^{n\times n}$, $\mathbf{x} \in \mathbb{R}^n$, $\mathbb{C}^n$,

$\mathbf{b} \in \mathbb{R}^n, \mathbb{C}^n$ は下記のように与えたものを使用している．

A　$d_i := 10^{-26(i-1)/n}$ として実対角行列 $D = \mathrm{diag}[d_1 \cdots d_n]$，乱数行列 $R_{\mathrm{re}}, R_{\mathrm{im}} \in \mathbb{R}^{n\times n}$ を生成し，$A := R_{\mathrm{re}} D R_{\mathrm{re}}^{-1}$(実行列), $A := R_{\mathrm{re}} D R_{\mathrm{re}}^{-1} + \mathrm{i} R_{\mathrm{im}} D R_{\mathrm{im}}^{-1}$(複素行列) として mpmath を用いて DD, TD, QD 精度相当の精度で計算を行った．

$\mathbf{x}, \mathbf{b}$　真の解は $\mathbf{x} = [0\ 1\ \cdots\ n-1]^T$(実ベクトル), $\mathbf{x} = (1+\mathrm{i})[0\ 1\ \cdots\ n-1]^T$(複素ベクトル) とし，mpmath を用いて $\mathbf{b} := A\mathbf{x}$ を求めて使用した．

ベンチマークテスト結果を表 1 に示す．DD, TD，QD 精度それぞれに対し，高速化なし，AVX2 による高速化あり，同精度の mpmath の直接法の計算時間と，得られた近似解の最大相対誤差を記してある．

表 1. 直接法の計算時間と最大相対誤差
EPYC: $n = 1000$

	Real		Complex	
Prec. Direct Methods	Comp.Time(s)	Max Relerr	Comp.Time(s)	Max Relerr
DD	1.17	2.8E-02	4.7	9.3E-14
DD(AVX2)	0.34	2.5E-02	1.38	1.2E-13
MP(106bits)	1844.5	2.8E-02	3594.5	5.7E-15
TD	10.2	4.1E-18	51.0	1.8E-30
TD(AVX2)	10.5	6.2E-19	39.1	2.8E-30
MP(159bits)	1877.0	1.8E-18	3801.1	1.3E-30
QD	39.7	4.0E-35	145.1	3.7E-47
QD(AVX2)	12.6	5.3E-35	46.6	4.2E-46
MP(212bits)	1894.8	5.5E-35	3812.1	7.6E-47

DD 精度では mpmath の直接法 (`lu_solve`) の方が多少精度が良いが，TD, QD 精度では我々の実装と mpmath とはほぼ同じ精度の近似解を得られている．計算時間においては AVX2 による高速化は Python 環境下でも実 LU 分解で約 0.97〜3.4 倍，複素 LU 分解で約 1.8〜3.8 倍に達している．同じ精度における mpmath との比較では，実 LU 分解で約 150〜5371 倍，複素 LU 分解で約 262〜6028 倍，我々の実装のほうが高速であることが分かる．

4　今後の課題

今後の課題としては，に示すように，複素線形計算をサポートした BNCmatmul Version 0.22 をリリースし，今回ここで示した Python 多倍長精度計算機能を広く利用可能とすることが挙げられる．

謝辞　本研究は科学技術研究費 20K11843 および 20K11843 の助成を利用して行われた．

参考文献

[1] Python Software Foundation.
[2] Case Van Horsen. General multi-precision arithmetic for python 2.6+/3+ (gmp, mpir, mpfr, mpc). `https://github.com/aleaxit/gmpy`.
[3] Fredrik Johansson, et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic*. `http://mpmath.org/`.
[4] Tomonori Kouya. BNCmatmul. `https://github.com/tkouya/bncmatmul`.

gmpxx_mkII.h の紹介:GNU 多倍長精度ライブラリのための C++ ラッパーの効率性と使いやすさを向上

中田 真秀 [1], 中里 直人 [2], 河野 郁也 [3]
[1] 理化学研究所 開拓研究本部, [2] 会津大学 コンピュータ理工学部, [3] 静岡理工大学 情報学部
e-mail : maho@riken.jp

1　はじめに

本研究では、GNU 多倍長精度ライブラリ (GMP) [1] 向けの新たなヘッダーオンリー C++ ラッパー、gmpxx_mkII.h を開発したのでそれを紹介する [2]。この約 5000 行のラッパーは、GMP に標準添付されている gmpxx.h と高いソースレベルでの互換性を保ちつつ、コードの可読性とクラス設計の改善を主目的とした。

gmpxx.h は複雑なマクロやテンプレートを多用したジェネリックプログラミングで構成されており、様々なプラットフォームで動作し、高いパフォーマンスを実現している。しかし、ユーザー作成のプログラムにおけるコンパイルエラーの解釈とデバッグの困難さ、C++ の標準的な使用方法の一部が適用できないこと、そして gmpxx.h 自体の可読性の低さなどの問題があった。gmpxx_mkII.h は、GMP の mpf_t, mpq_t, mpz_t を直接クラスとしてカプセル化する従来的なアプローチを採用し、これらの問題を解決した。

2　テンプレート化された関数への演算式の直接渡しが可能に

gmpxx.h の既知の制限として、テンプレート化された関数への演算式の直接渡しができないという問題があったが、gmpxx_mkII.h ではこれが可能となった [3]。

例えば、gmpxx.h では a+b のような演算式を std::min などのテンプレート化された関数の引数として std::min(a+b, a) のように直接使用できず、std::min(mpf_class(a+b), a) のような形式を強いられていた。これは a+b の型が gmpxx.h の内部表現であり、C++ のコンパイラが自動的に mpf_class に変換できないためである。さらに、この制限に関連するコンパイルエラーは一見して理解しがたいものであった。

このような表現は MPLAPACK [4] でも頻繁に使用されており、メンテナンスコストの大幅な増加を招いていた。

3　移行と非互換性について

gmpxx_mkII.h と gmpxx.h は高いソースレベルの互換性を維持するよう設計した。既存の GMP ベースのプログラムは、ヘッダファイルの変更のみで (も) 容易に移行可能となった。ただ、オーソドックスなクラス化では実装できない機能が二つあった。これらはサポートしないこととした。(i) 左変数のみ参照した関数、これは右辺未指定での左変数の精度を参照したランダム数生成で使われている。(ii) 精度の遅延評価、これは例えば、mpf_class f(1.0/3.0, 512) とすると、f には 512bit の精度での 1/3 が入る。これらを踏まえ GMP の品質保証プログラムに修正を加えたもの、および新たに作成したテストについてそれらに合格するようにした。

4　初等関数や超越関数の実装

MPFR から独立する目的のため、対数、指数、三角関数などの広く使用される初等関数や超越関数を新たに実装した。対数、指数関数、atan には幾何算術平均、三角関数はテイラー展開をアルゴリズムとして用いた。

5　新設モードについて

gmpxx_mkII.h は互換モード、mkII モード (デフォルト)、mkIISR モードの三種類を提供する。define を適宜行うことで制御可能である。互換モードは namespace なし、初等関数などの拡張なし、かつ上セクションで述べた非互換な部分のみサポートしないモードとした。mkII モードは互換モードに namespace 導入、初等関数などの拡張あり、のモードとした。mkIISR モードは mkII モードにさらに精度固定を仮定する。つまり、プログラムの開始時に精度を指定し、以降は変更しないことを前提とすることで、不要な多倍長数のメモリ割り当てを削減し、計算速度の向上を図るものである。mkII, mkIISR モードはバイナリ互換をもつ。

6　ベンチマーク一例

AMD Ryzen 3970X (32 cores, 2.2GHz 固定)、精度 512bit、1 億次元のベクトル mpf_t の内積を計算した。シングルコアでの結果は、C 版 7.16 秒、標準添付版 10.3 秒、互換モード 7.64 秒、精度固定モード 7.09 秒であった。OpenMP 版では、C 版 0.334 秒、標準添付版 0.332 秒、互換モード 0.319 秒、精度固定モード 0.317 秒程度であった。これらの結果から、新ラッパーが標準添付版と比較して良いパフォーマンスを示すことが確認された。CPU や条件を変更しても mkIISR モードは標準の gmpxx.h と比較して概して高いパフォーマンスを発揮した。詳細については当日発表する。

7　入手方法、対応 OS

gmpxx_mkII.h はhttps://github.com/nakatamaho/gmpxx_mkIIで 2 条項 BSD ライセンスとして近日公開予定である。64bit の (x86_64 や arm64) の Linux と macOS で利用可能である。

8　展望

gmpxx_mkII.h は非常にオーソドックスな作りとなっている、そして品質保証プログラムもついているため、拡張性に優れている。例えば他の多倍長精度クラスをサポートすることも容易であろう。生成されるコードも素直であるためパフォーマンスボトルネックの特定も容易であり、今後 MPLAPACK [4] などに組み込む予定である。

謝辞　本研究は JSPS 科研費 JP23K11133 の助成を受けた。

参考文献

[1] Gorbjön Granlund and the GMP development team, GNU MP: The GNU Multiple Precision Arithmetic Library, 2012-2023.

[2] Maho Nakata, Yet another GMP C++ wrapper for High-Precision Calculations, 2024.

[3] Gorbjörn Granlund, C++ Interface Limitations, https://gmplib.org/manual/C_002b_002b-Interface-Limitations, 2012-2023.

[4] Maho Nakata, MLAPACK: a multi-precision linear algebra package, https://github.com/nakatamaho/mplapack, 2008-2022.

非正規化合成浮動小数点数を用いた計算方法の実装方法について

今村 俊幸 [1], 尾崎 克久 [1,2]
[1] 理化学研究所, [2] 芝浦工業大学
e-mail : imamura.toshiyuki@riken.jp

1　概要

単精度, 倍精度を和合成した Double-double(DD) や Quadruple-double(DD) は, qd ライブラリが広く知られている. C++ class ならびに fortran module により安価に高性能を実現利用可能である. 尾崎・今村は正規化手続きを省略した複数語フォーマットについて, その誤差解析と安定化技術についてその技術蓄積を進めてきた [1]. 正規化の有無, 内部演算アルゴリズムさらには合成基底 (種類/個数) を制御し, プログラム上で自然に記述可能な C++ クラス `mX_Real` を開発し, その実応用を目指して現在公開中である [2].

2　Pair からその拡張疑似複数語演算

倍精度を基準としてその Double Word 演算 (DW) をパッケージ化したものが DD に相当する. $a = a_0 + a_1 (u|a_0| \geq |a_1|), b = b_0 + b_1 (u|b_0| \geq |b_1|)$ のように, a, b を 2 語和かつ括弧内正規化条件を課した表現であり, TwoSum, TwoProductFMA 等を組み合わせて四則演算が実現される. DW の加算 `DW_add` の疑似コードは以下のようになる.

```
1:  function [c0,c1] = DW_add(a0,a1,b0,b1)
2:    [s,e] = TwoSum(a0,b0)
3:    e = e + a1 + b1
4:    [c0,c1] = FastTwoSum(s,e)
```

4 行目 FastTwoSum が正規化操作に対応し, 正規化を省略したものが Pair 演算 (PA) の加算 `PA_add` になる. これらは減算, 乗算, 徐算, 平方根にも適用される (ただし、加減乗算は EFT の意味で定義されるが, その他の演算は近似方法によって各種実現方法が存在する). これらの拡張として, 3 語和で構成する Triple Word(TW), 4 語 Quadruple Word(QW) についても最後に行う正規化を省略した疑似 Triple Word(QTW), 疑似 Quadruple Word(QQW) が提案できる. DW と PA は演算量の観点から PA に 1.5∼3 倍程度の優位性があるが, 正規化されないため複数語による表現能力が 1 語近くまで下がる弱点がある. 特に, 徐算は影響を受けやすく正規化を適切なタイミングで行う必要がある. また, Sloppy 版は劣精度高速版であるが, 精度を上げた Accurate 版 (IEEE 精度相当版など) も利用価値が高い.

3　mX_Real

mX_Real は, PA, DW, QTW, TW, QQW, QW を統一的に扱う C++ クラスである.

```
1:  using namespace mX_real;
2:  template < class T, T_mX(T) > T calc_pi() {
3:    auto y = T::zero(), f = T::one(); int s1 = 1, s2 = 2, s3 = 3;
4:    while ( f != T::zero() ) {
5:      auto z = T::zero(); T::base_T h = 1.0;
6:      for(int i=0; i<9; i++) {
7:        z += (T::one()/s1 + T::two()/s2 + T::one()/s3)*h;
```



```
 8:        s1 += 4; s2 += 4; s3 += 4; h *= (-0.25);
 9:      } y += z * f; f *= h;
10:    }
11:    return y;
12:  }
13:  void calc(){
14:    std::cout << calc_pi<tX_real::tx_real<double,Algorithm::Sloppy>>();
15:  }
```

calc_pi はテンプレート引数 T(<>内) に合成型数のデータ型を渡すことができ, 例では double を基底とする 3 語長で内部に Sloppy アルゴリズムを使う TW を指定している. Algorithm::Quasi と指定すれば非正規化版 QTW に切替わる. dX_real::dx_real, tX_real::tx_real, qX_real::qx_real がそれぞれ 2,3,4 語の複合型であり, それぞれ 2 つのテンプレート引数<Type,Algrithm>をもち, Type に double もしくは float, Algorithm に Algorithm::Quasi(非正規化), Algorithm::Sloppy, Algorithm::Accurate が指定可能. 同一基底であれば任意の合成型数間での変換 (自明な型変換やコピーコンストラクト) が可能である. 例えば, auto x = tX_real::tx_real<double,Algorithm::Quasi>(1.0+sqrt(qX_real::qx_real<>(1.0)/5)); は 4 語 (quad-ddouble) 精度で 1.0+1/5 の平方根を定め, 3 語 (triple-double) 精度で丸め x に初期化するものである. この際, 初期化の計算は Algorithm::Accurate(デフォルト) で実施されるが、コンストラクタの時点で変数 x は Algorithm::Quasi 属性を持ち, 以降の計算は Pair 演算 (非正規化合成数) として取り扱われる (異アルゴリズム属性の変数間の演算は低精度アルゴリズムに合わせて計算される).

mXX_Real は qd ライブラリ同様四則演算をサポートし, C++ の文脈で合成数型を数としてあるがままの数式を記述できる. 浮動小数点数として必要な比較演算ならびに, max, min, abs, sqrt のみ利用可能であり, その他初等関数は別パッケージで対応予定である. 現状 level1, level2 BLAS は記述性 · 性能両面で, ある程度実用に供する. 一方, 行列積は性能面で課題があり, 内部データの適切な並替え (AOSOA 等) 技術導入が必須である.

4 まとめ

合成浮動小数点数型の利点はより安価に利用可能な低精度演算器を活用し, 高精度演算を実現する点にある. AI 普及により FP32 よりも低精度な演算器が CPU や GPU に搭載される中で, 本技術活用はプロセッサ設計や消費電力削減の観点からも有望である. 非正規化版アルゴリズムの安定化技術の探求, 計算機科学分野との強固な連携, 利用ソフトウェアの充実と高性能化を精力的に進める.

謝辞 本研究は, 文部科学省「次世代計算基盤に係る調査研究」事業ならびに, JSPS 科研費 (23H03410, 23K28100) の助成を受けたものです.

参考文献

[1] 尾崎 克久, 今村 俊幸, Pair Arithmetic の拡張とその使い方について, IPSJ 研究報告 HPC, 2023-HPC-192(19), pp. 1–8, 2023

[2] mX_Real: R-CCS github repositry, https://github.com/RIKEN-RCCS/mX_real (2024 年 8 月 2 日最終アクセス)

