# Python プログラミング環境における多倍長精度数値計算について

# Python プログラミング環境における多倍長精度数値計算について

On multiple precision computing on Python programming environment

幸谷智紀*
Tomonori KOUYA

Abstract: Multiple precision numerical computing is currently one of critical techniques, which can be applied to ill conditioned problems or severe cases required to be able to guarantee enough accuracy of numerical results. On the other hand, Python environment is used for AI technique such as deep-learning which uses short length floating-point arithmetic such as IEEE754-1985 half-precision (binary16) or bfloat16 proposed by Intel in order to obtain the best performance. Therefore, multiple precision floating-point arithmetic in Python are not currently gotten attention. In this paper, we introduce examples of ill conditioned problems, show some methods to evaluate effect by round-off errors, and then explain the examples with multiple precision library such as mpmath and gmpy2. Finally, we show that RDD library, which is our original multi-component multiple precision arithmetic library available on Python or C environment, can get the best high performance for 2 to 4 times longer precision arithmetic than binary64 compared with mpmath and gmpy2.

## 1. 初めに

多倍長精度数値計算は，丸め誤差に敏感な悪条件問題や，精度保証が必要となる場合において必要不可欠な現代科学技術の一つである．一方，Deep Learning に代表される AI 技術の活用のためには Python プログラミング環境が使用されることが多く，もっぱら高パフォーマンスのために半精度や bfloat16 のような短い仮数部の浮動小数点演算を多用するため，IEEE754-1985 倍精度 (binary64) より長い仮数部を持つ多倍長精度浮動小数点演算はあまり注目されていない．

本稿では多倍長精度計算が必要となる悪条件問題を紹介し，Python 環境における丸め誤差の測定法，区間演算の使用例，そして mpmath や gmpy2 といった既存の任意精度演算パッケージの使用例を紹介する．最後に，2020 年現在まだ提供されていない，我々独自のマルチコンポーネント型の多倍長精度演算ライブラリ RDD を使用し，binary64 の 2〜4 倍精度の演算が高いパフォーマンスで実行できることをベンチマークテストで示す．

## 2. 丸め誤差計測方法

有限桁の浮動小数点数に基づく数値計算では，計算過程において丸め誤差が発生する．これを減らす根本的な解決策は，浮動小数点数を使わず整数や有理数で完結するアルゴリズムに変更するか，使用する浮動小数点数の仮数部の桁数を増やしてマシンイプシロン (丸め誤差の最小単位) を小さくする他ない．

以下，桁落ちする計算の例として，ロジスティック写像

$$f(x) = 4x(1-x) \tag{1}$$

を使った，漸化式

$$x_{i+1} := f(x_i) = 4x_i(1-x_i) \tag{2}$$

によって導出される実数列 $\{x_i\}_{i=0}^{100}$ を，IEEE754 倍精度で計算するスクリプト (Listing 1) と，数値結果 (Fig.1) を示す．初期値は全て $x_0 = 0.7501$ と指定した．一見するときちんと計算できているように見えるが，この数列の計算は断続的な桁落ちが無限に発生する悪条件な計算の一例で，$x_{50}$ より先の値は全く有効桁数がなくなる．

桁落ちが発生する理由は，$0 < x_i < 1$ であれば，$1 - x_i$ のところで，必ず少しずつではあるが有効桁数が減ってくる．この写像の性質から，再び桁落ちする領域に $x_{i+1}$ が戻ってくるため，$i$ が大きくなるにつれて，精度が悪化していくことになる．

Listing 1: ロジスティック写像に基づく数列計算

```
# logistic_function.py
  x = [0.7501] # 初期値を配列の先頭値に格納

  # x[i+1] に値を追加
  for i in range(0, 100):
    x.append(4 * x[i] * (1 - x[i]))


  # x[0], x[10], ..., xを表示 [100]
  print('␣␣␣␣i,␣␣␣␣␣␣␣␣␣␣␣x[i]␣␣␣␣␣␣␣␣')
  for i in range(0, 101):
    if i % 10 == 0:
      print(f'{i:5d},␣{x[i]:25.17e}')
```

このような丸め誤差の影響が拡大する悪条件問題に対して，誤差を見積もる方法としては

1. IEEE754 丸めモード変更による計算結果の差異を利用する方法
2. 区間演算による厳格な誤差評価法
3. 多倍長精度計算を用いた誤差の正確な算出

の 3 つが考えられ，1 → 2 → 3 という順に誤差評価のためのコストが増大することが知られている．逆に，誤差評価の正確性という点では，2 は厳格ではあるが過大になるケースが多く，1 より 3 の方がより正確な誤差の見積もりが可能になる．誤差評価のコストと正確性を鑑みると，両者はトレードオフの関係にある．

2020 年 2 月 28 日 受理
*情報学部 コンピュータシステム学科

```
$ python3 logistic_function.py
    i,           x[i]
    0,   7.50099999999999989e-01
   10,   8.44495953602201199e-01
   20,   1.42939724528399537e-01
   30,   8.54296020314658677e-01
   40,   7.74995885155205677e-01
   50,   7.95128764501052410e-02
   60,   2.73187240440892098e-01
   70,   5.52530562083362264e-01
   80,   2.16255663995813446e-01
   90,   7.87467937188412348e-01
  100,   2.69706745887651977e-01
```

Fig. 1: IEEE 倍精度 (binary64) でのロジスティック写像計算

以下，これらの方法を Python スクリプトで実行してみることにする．

### 2.1 IEEE754 丸めモードを利用する方法

計算桁数を変えずに丸め誤差を検出する方法として，もっとも原始的な「職人芸」としては，例えば山下真一郎によるエコーシステムのように，末尾桁に誤差を混入させ，元の計算結果との差異を測る，というものがある．これをもう少しシステム的に確実かつ簡単に実行する方向として，IEEE754-1985 浮動小数点演算規格に定められている丸め方式 (丸めモード) を変更することで行う評価法がある [10]．現在，浮動小数点演算ユニットがハードウェアとして備わっている CPU では，デフォルトの丸めモードである RN(Round to Nearest) の他，$+\infty$ 方向への丸めを行う RP(Round to Plus infinity)，$-\infty$ 方向への丸めを行う RM(Round to Minus infinity) が備わっている．計算過程の前にこの丸めモード変更を行い，その際の最大値で相対誤差の評価を行うという簡易的な区間演算のような手法である．丸め誤差を確率変数として考えるというのは Henrich[5] から始まっているが，現在は Higham らが理論的な観点からの確率的誤差評価法を提案している [6]．

丸めモード変更のためには，Linux, Windows, macOS それぞれに浮動小数点演算ユニットの状態を変化させるための標準的な関数が備わっており，それを利用すればよい．Linux, macOS では `fesetround` 関数，Windows では `_controlfp_s` 関数を使用する．我々は OS によらず Python 環境で丸めモードの設定と確認ができるよう，`rmode.py` を作成し，ここで次の 2 つの関数を定義した．

- **set_rmode 関数** …`FE_NEAREST`(RN)，`FE_UPWARD`(RP)，`FE_DOWNWARD`(RM) を引数に与えて丸めモードを変更する．
- **get_rmode 関数** … この関数実行時の丸めモードを標準出力に表示する．

`rmode.py` を読み込んで，これらの関数を使い，それぞれの丸めモードでの数列 $\{x_n^{\mathrm{RN}}\}, \{x_n^{\mathrm{RP}}\}, \{x_n^{\mathrm{RM}}\}$ を求め，その差異の最大値 $x_n^{\max} := \max\{|x^{\mathrm{RN}} - x^{\mathrm{RP}}|, |x^{\mathrm{RN}} - x^{\mathrm{RP}}|, |x^{\mathrm{RM}} - x^{\mathrm{RP}}|\}$ を $x_n^{\mathrm{RN}}$ の絶対誤差の評価値として使用する．このスクリプトを Listing 2 に示す．この結果は Fig.2(上) に示したようになり，$x_{50}$ で有効桁数が失われていることが分かる．

Listing 2: ロジスティック写像 (丸め誤差評価付き)

```
# logistic_function_rmode.py
import rmode # 丸めモード変更

# デフォルトモード (RN)
rmode.print_rmode()
# --- start ---
x_rn = [0.7501] # 初期値を配列の先頭値に格納

# x[i+1] に値を追加
for i in range(0, 100):
  x_rn.append(4 * x_rn[i] * (1 - x_rn[i]))
# --- end ---

# RP モード
rmode.set_rmode(rmode.FE_UPWARD)
rmode.print_rmode()
# --- start ---
# x_rp[i+1] を計算
# --- end ---

# RM モード
rmode.set_rmode(rmode.FE_DOWNWARD)
rmode.print_rmode()
# --- start ---
# x_rm[i+1] を計算
# --- end ---

# diff_rn_rm, diff_rn_rp, diff_rp_rm
rel_diff_rn_rm = [abs((x_rn[i] - x_rm[i]) / x_rn
    [i]) for i in range(len(x_rn))]
rel_diff_rn_rp = [abs((x_rn[i] - x_rp[i]) / x_rn
    [i]) for i in range(len(x_rn))]
rel_diff_rm_rp = [abs((x_rm[i] - x_rp[i]) / x_rn
    [i]) for i in range(len(x_rn))]
max_rel_diff = [max(rel_diff_rn_rm[i],
    rel_diff_rn_rp[i], rel_diff_rm_rp[i]) for i
     in range(len(x_rn))]

# x[0], x[10], ..., xを表示 [100]
print('␣␣␣␣i,␣␣␣␣␣␣␣␣␣␣x_rm[i]␣␣␣␣␣␣␣␣,␣␣␣␣␣␣␣␣
    ␣␣␣␣x_rn[i]␣␣␣␣␣␣␣␣␣,␣␣␣␣␣␣␣␣␣␣␣x_rp[i]␣␣␣␣␣␣
    ␣␣,max_rel_diff')
for i in range(0, 101):
  if i % 10 == 0:
    print(f'{i:5d},␣{x_rm[i]:25.17e},␣{x_rn[i
        ]:25.17e},␣{x_rp[i]:25.17e},␣{
        max_rel_diff[i]:5.1e}')
```

### 2.2 区間演算

区間演算 (interval arithmetic) は，真の値が存在する区間の端点を浮動小数点数で表現し区間単位で演算し，誤差を含む区間を常に保つ厳格な演算手法である．区間 $I(a) = [\underline{a}, \overline{a}]$ の左端点 $\underline{a}$ と右端点 $\overline{a}$ を浮動小数点数として表現するためには，真値 $a$ の丸めに際してそれぞれ RM 方式，RP 方式を使用する．こうすることで，$a$ の符号に寄らず，左右の端点を有限桁の浮動小数点数として表現することができるようになる．

区間 $I(a), I(b) = [\underline{b}, \overline{b}]$ に対する四則演算は次のように実行される．ここで $\oplus$, $\ominus$, $\otimes$, $\oslash$ は通常の浮動小数点演算における加減乗算を示す．

$$
\begin{aligned}
I(a) + I(b) &= [\mathrm{RM}(\underline{a} \oplus \underline{b}), \mathrm{RP}(\overline{a} \oplus \overline{b})] \\
I(a) - I(b) &= [\mathrm{RM}(\underline{a} \ominus \overline{b}), \mathrm{RP}(\overline{a} \ominus \underline{b})] \\
I(a) \times I(b) &= [\underline{a \otimes b}, \overline{a \otimes b}] \\
&\quad \text{ここで} \\
&\quad \underline{a \otimes b} = \min\{\mathrm{RM}(\underline{a} \otimes \underline{b}), \mathrm{RM}(\underline{a} \otimes \overline{b}), \\
&\qquad \mathrm{RM}(\overline{a} \otimes \underline{b}), \mathrm{RM}(\overline{a} \otimes \overline{b})\} \\
&\quad \overline{a \otimes b} = \max\{\mathrm{RP}(\underline{a} \otimes \underline{b}), \mathrm{RP}(\underline{a} \otimes \overline{b}), \\
&\qquad \mathrm{RP}(\overline{a} \otimes \underline{b}), \mathrm{RP}(\overline{a} \otimes \overline{b})\}.
\end{aligned}
\tag{3}
$$

```
$ python3 logistic_function_rmode.py
最近偶数値丸め
+Inf への丸め
-Inf への丸め
    i,           x_rm[i]        ,           x_rn[i]        ,           x_rp[i]        ,max_rel_diff
    0,   7.50099999999999989e-01,   7.50099999999999989e-01,   7.50099999999999989e-01, 0.0e+00
   10,   8.44495953602235394e-01,   8.44495953602201199e-01,   8.44495953602203309e-01, 4.0e-14
   20,   1.42939724494728609e-01,   1.42939724528399537e-01,   1.42939724526234518e-01, 2.4e-10
   30,   8.54295985559328397e-01,   8.54296020314658677e-01,   8.54296018080507924e-01, 4.1e-08
   40,   7.74953760069827080e-01,   7.74995885155205677e-01,   7.74993177338575201e-01, 5.4e-05
   50,   1.09649817246645534e-01,   7.95128764501052410e-02,   8.13185242954565651e-02, 3.8e-01
   60,   9.11998235902822794e-04,   2.73187240440892098e-01,   5.22706152743196872e-01, 1.9e+00
   (略)

$ python3 logistic_function_interval.py
    i, [       x[i].left         ,         x[i].right       ]
    0, [  7.50099999999999989e-01,   7.50099999999999989e-01]
   10, [  8.44495953582817260e-01,   8.44495953621622331e-01]
   20, [  1.42919379590904727e-01,   1.42960069692795372e-01]
   30, [ -1.50087739433940874e+03,   8.13706409017080773e+02]
   40, [                     -inf,                       inf]
   (略)

$ python3 logistic_function_mpmath.py
   i,                      x[i]                    , reldiff_x[i]
   0,  0.7501 ,  4.2e-42
  10,  0.844495953602217447537148702561541372668 7 ,  6.3e-39
  20,  0.142939724512307655284281757231306288305 1 ,  3.7e-35
  30,  0.854296003704421891666131184258892843388 5 ,  6.3e-33
  40,  0.774975753118201241280223461270756292212 8 ,  8.4e-30
  50,  0.093375332197703029055189663345836976725 87 ,  5.0e-26
  60,  0.408220168290878131871019584781918317890 9 ,  2.0e-23
  70,  0.071511999705058574901435477093910453812 37 ,  6.1e-20
  80,  0.463253302900775719152417290949214181004 5 ,  1.9e-17
  90,  0.001334405012087528183908816303744773552959 ,  4.8e-13
 100,  0.078817989366641240348640554153741897446 01 ,  6.2e-11
```

Fig. 2: ロジスティック写像に基づく数列計算: 丸め方式変更を用いた有効桁判定方法 (上)，区間演算結果 (中)，mpmath による多倍長精度計算結果 (下)

逆数は $1/I(b) = [1 \oslash \overline{b}, 1 \oslash \underline{b}]$ となるが，この場合は $0 \notin I(b)$ という前提が必要である．この時，除法は逆数を用いて $I(a)/I(b) = I(a) \times (1/I(b))$ として求める．

Python には後述する mpmath パッケージに区間演算 iv の機能があるが，ここでは IEEE754 倍精度の丸めモード変更だけで実装した独自の `interval.py`(Listing 3) を使用する．Python には演算子の定義ができるクラスが用意されており，コンストラクタ (`__init__`)，加算 (`__add__`) といった固定された関数を定義するだけで，独自のデータ型に対するクラスライブラリが簡単に記述できる．但し，この機能は後述するように，記述的には楽ができるが，あまり実効効率が良くない．

Listing 3: 区間演算ライブラリ (一部)

```
# interval.py
import rmode
import math

# R.T.Kneusel, "Numbers and Computers", Springer, 2015.
# Interval -> [left, right]
class Interval:

  # 開始前のデフォルト丸めモード
  #default_rmode = rmode.FE_TONEAREST
  default_rmode = rmode.get_rmode()

  # コンストラクタ
  def __init__(self, left, right = None):
    if right == None: self.left = left; self.right = left
    else: self.left = left; self.right = right

  # + : 加算
  def __add__(self, y):
    rmode.set_rmode(rmode.FE_DOWNWARD)
    left = self.left + y.left
    rmode.set_rmode(rmode.FE_UPWARD)
    right = self.right + y.right
    rmode.set_rmode(Interval.default_rmode)
```

```
    return Interval(left, right)
```

ここでは Knuesel[13] の Interval クラスを用いて区間演算をロジスティック写像計算に使用する (Listing 4)．実行結果は Fig.2(中) に示す．普通に計算するより区間幅が急激に大きくなり，$x_{30}$ で真値の位置が特定できないほど区間が広がり，$x_{40}$ では区間の両端点がオーバーフローして無限大に発散してしまっている．Listing 2 では $x_{40}$ ではまだ 10 進 4 桁は有効桁数が残っており，明らかに精度が悪化している．このように，単純に区間演算を使うと誤差評価としては過大になりがちで，悪条件問題ではこのような区間の爆発現象が起きることが知られている．反面，区間内には必ず真値を含むことは理論的に保障されているので，コンピュータによる計算結果の検証方法としてはよく利用されている．

Listing 4: ロジスティック写像 (区間演算)

```
# logistic_function_interval.py
from interval import * # 手製区間演算ライブラリ

x = [Interval(0.7501)]

# x[i+1] に値を追加
const4 = Interval(float(4)) # const4 := 4
const1 = Interval(float(1)) # const1 := 1
for i in range(0, 100):
  x.append(const4 * x[i] * (const1 - x[i]))

# x[0], x[10], ..., xを表示 [100]
print('␣␣␣␣i,␣[␣␣␣␣␣␣␣x[i].left␣␣␣␣␣␣␣␣␣,␣␣␣␣␣␣␣
    ␣␣x[i].right␣␣␣␣␣␣␣]')
for i in range(0, 101):
  if i % 10 == 0:
    print(f'{i:5d},␣[{x[i].left:25.17e},␣{x[i].
        right:25.17e}]')
```

### 2.3 多倍長精度計算

IEEE754 倍精度を使う限り，仮数部は 2 進 53 bits 固定でしか計算できず，10 進数換算で 15 桁程度の精度しか求められない．そこで，仮数部を可変に伸ばした多倍長浮動小数点数をソフトウェア的に定義し，より長い仮数部桁の浮動小数点数で計算した結果を真値の代わりに利用し，短い桁による計算結果の誤差を正確に見積もることを考える．

Python では mpmath パッケージ [8] が提供されており，これを使うことで多数桁方式による任意長の仮数部を持つ浮動小数点演算が実行できる．デフォルトでは Python の整数型を用いた実装が使用されるが，後述する gmpy2 パッケージ [7] をインストールすると，自動的に GNU MP(GMP)[14]，MPFR[12]，MPC[3] といった高速な任意精度多倍長ライブラリが使用される．これは mpmath パッケージを読み込んだ後，下記のような Python コードで `libm.BACKEND` 属性を調べることで判別できる．

```
import mpmath
print('mpmath.libmp.BACKEND = ', mpmath.libmp
.BACKEND)
```

Python の整数演算を利用している場合は `python` と表示され，gmpy2 をインストールして居る場合は `gmpy` と表示される．

mpmath で 10 進 40 桁相当の浮動小数点数を用いてロジスティック写像を計算し，その相対誤差を 10 進 80 桁相当の計算結果を比較することで求めるスクリプトを Listing 5 に示す．また，その計算結果を Fig.2(下) に示す．$x_{100}$ も 10 進 10 桁程度の有効桁数を保っていることが分かる．また，これを比較することで，丸めモード変更による相対誤差の評価が正しいことも見て取れる．

Listing 5: ロジスティック写像 (mpmath 利用)

```
import mpmath

# 10 進 40 桁計算
mpmath.mp.dps = 40 # 仮数部の進桁数 10
x = [mpmath.mp.mpf('0.7501')]
for i in range(0, 100):
  x.append(4 * x[i] * (1 - x[i]))

# 10 進 80 桁計算
mpmath.mp.dps = 80
xl = [mpmath.mp.mpf('0.7501')]
for i in range(0, 100):
  xl.append(4 * xl[i] * (1 - xl[i]))

reldiff_x = [mpmath.fabs((xl[i] - x[i]) / xl[i])
     for i in range(len(x))]

print('␣␣␣␣i,␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣x[i]␣␣␣␣␣␣␣␣␣
    ␣␣␣␣␣␣␣␣␣␣,␣reldiff_x[i]')
for i in range(0, 101):
  if i % 10 == 0:
    print(f'{i:5d},␣', mpmath.nstr(x[i], 40), ',
        ␣', mpmath.nstr(reldiff_x[i], 2))
```

なお，gmpy2 は単独で GMP, MPFR, MPC の多倍長計算の利用が可能である．実際，ロジスティック写像の計算を 128 bits の仮数部で行い，256 bits の計算結果を真値の代わりに使用することで相対誤差の評価を正確に行うスクリプトは Listing 6 のようになる．計算結果は mpmath と同様になるので省略するが，書式制御が使えるので，mpmath より表示結果の見栄えは良くなる．

Listing 6: ロジスティック写像 (gmpy2 使用)

```
import gmpy2

# 128 bits
gmpy2.get_context().precision = 128
x = [gmpy2.mpfr('0.7501')]
for i in range(0, 100):
  x.append(4 * x[i] * (1 - x[i]))

# 256 bits
gmpy2.get_context().precision = 256
xl = [gmpy2.mpfr('0.7501')]
for i in range(0, 100):
  xl.append(4 * xl[i] * (1 - xl[i]))

reldiff_x = [gmpy2.reldiff(xl[i], x[i]) for i in
    range(len(x))]

print('␣␣␣␣i,␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣x[i]␣␣␣␣␣␣␣
    ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣')
for i in range(0, 101):
  if i % 10 == 0:
    print(f'{i:5d},␣{x[i]:50.40e},␣{reldiff_x[i
        ]:5.1e}')
```

mpmath にしろ gmpy2 にしろ，仮数部の桁を増やして計算するので，計算結果の精度は格段に向上する反面，ソフトウェア的に演算を実装する必要があるため，ハードウェアの 1 命令で実行できる IEEE754 倍精度計算に比べて計算コストは格段に増える．そのため，止むを得ない場合以外は，なるべく多倍長精度計算は利用せずに済む計算アルゴリズムを使用することが求められる．

とはいえ，既存のアルゴリズムをそのまま使って悪条件問題を計算コストをかけて乗り切るという解決法は，安直ではあるが手軽な手法ではある．特に，Python の環境では演算子もそのまま利用でき，C++のように煩雑なテンプレートを意

識せず使えるのは魅力的である．

## 3. 多倍長精度浮動小数点演算の実装方式

コンピュータの性能を最大限発揮させるプログラムを記述するためには，機械語にごく近いアセンブラ言語を使うのが最も性能を発揮させやすいが，プログラムのメンテナンス性を高めるためには，せめて C もしくは C++のようなコンパイラ言語を使用することが望ましい．多倍長精度計算のように計算コストを要し，なおかつ複雑な処理を必要とする処理はなおさらである．

2020 年現在，最も広く使用され高速とされている多倍長精度ライブラリは，GMP, MPFR，QD[1] である．このうち前者 2 つは多数桁方式と呼ばれる方式で実装されており，CPU アーキテクチャに適したアセンブラルーチンに基づく任意長自然数カーネルライブラリを土台として構築されているものである．後者の QD はマルチコンポーネント方式と呼ばれ，IEEE754 倍精度 (binary64) 等既存のハードウェアサポートのある浮動小数点数の配列として多倍長精度の浮動小数点数を表現し，無誤差変換技法と呼ばれる精密な誤差評価付き計算を組み合わせて四則演算を実現する．

これらのライブラリの詳細と使用方法については拙著 [16] にまとめたので，詳細はそちらに譲り，ここではその概要を示すにとどめる．

### 3.1 多数桁方式

浮動小数点数の仮数部を任意桁数に設定できるように拡張する多倍長浮動小数点数の実装方式を多数桁 (multi-digits) 方式と呼ぶ．拡張方法としては整数演算ベースのものと既存浮動小数点数の仮数部を整数として扱う方式が考えられるが，GMP に備わっている整数演算ベースの任意長自然数 (mpn) カーネルライブラリが，営々20 年以上に渡って T.Granlaud を中心とする開発チームがサポートし続け，高速性を維持していることから，多数桁方式と言えばこの GMP に同梱されている mpf，もしくは mpn カーネルを利用する MPFR ライブラリが代表的なものと言える．固定精度の実装としては GCC に同梱されている float128 ライブラリがあり，IEEE754-1985 の 4 倍精度 (binary128) を忠実に実装したものとして有名であるが，ソースコードには GMP のソースコードが流用されており，一種の派生ライブラリとも言える．

GMP 及び MPFR のソフトウェア階層図を Fig.3 に示す．どちらも純粋な C プログラムで実装されており，mpn カーネルの主要部分については，C コードを各 CPU アーキテクチャに依存したアセンブラコードに置き換えて使用できる．x86_64 アーキテクチャの場合，アセンブラコードと C コードで速度比較すると，6〜7 倍もの速度差があることが判明している．

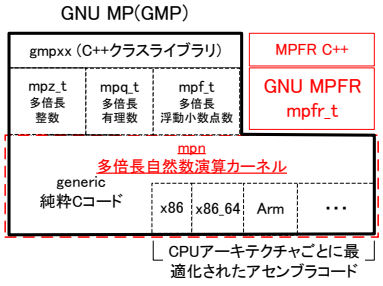


Fig. 3: GNU MP, MPFR のソフトウェア階層

MPFR は mpn カーネルとセットで動作する，GMP に同梱されている mpf より IEEE754-1985 規格に忠実に拡張された浮動小数点演算 mpfr を実装したもので，mpf と同様，仮数部を実行中にも動的に変化できるよう分離した複雑な構造体になっている (Fig.4)．mpf にはない初等関数や特殊関数，丸めモードをサポートしており，GMP のマニュアルでも mpf ではなく mpfr を使うことを推奨している．MPC はこの MPFR をベースに構築された任意精度複素数演算ライブラリである．

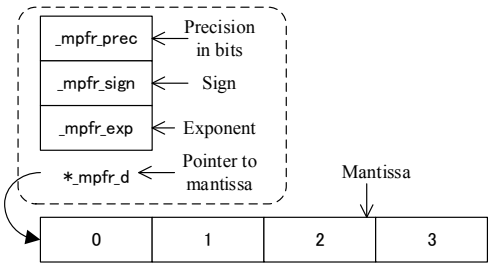


Fig. 4: mpfr_t データ型の構造体

Python の mpmath パッケージの多倍長精度浮動小数点演算 mpf はこの GMP をベースして実装されており，通常は Python の整数演算をベースにして動作するが，MPFR や MPC を直接呼び出してオーバーヘッドを極力減らした gmpy2 パッケージをインストールすると，GMP の mpn カーネルを利用するように変更される．mpmath マニュアルでも，パフォーマンス向上が見込めることから gmpy2 の利用を推奨している．

### 3.2 マルチコンポーネント方式

GMP の mpn カーネルは技術的に優れたものであるが，こと科学技術計算に関しては，数千桁もの計算が必要になることはさほどなく，倍精度を数倍超える程度，数百桁オーダーの計算ができれば十分なケースが多い．この場合，MPFR はいささかオーバースペックで，Fig.4 のような複雑なデータ構造は計算速度向上のためにはボトルネックになる．また，ハードウェアサポートのある binary32, binary64 は高速で，SIMD 命令のサポートもあり，整数演算よりはこちらを利用した方が計算速度的に有利になり得る．

Dekker[2] は，既存の浮動小数点演算においても，丸め誤差を含む加算，減算，乗算の計算結果と，誤差を分離して正確な等式が成り立つ無誤差変換 (error-free transformation) 技法を利用することで，既存浮動小数点数を 2 つ組み合わせ，例えば binary64 二つで 2 倍の仮数部長の Double-double(DD) 精度演算が実現できることを示した．同様の考え方で binary64 を 4 つ分，Quadruple-double(QD) 精度演算も実現でき，これらを C++のクラスライブラリとして実装したものを Bailey らが QD ライブラリ [1] として公開している．また，近年，DD と QD の中間となる 3 つの既存浮動小数点数を用いる 3 倍精度演算 [4] も提唱されており，binary64 を用いれば Triple-double(TD) 精度演算が実現できることが判明している．後述する我々の RDD ライブラリでは，これらの DD, TD, QD 演算を Fig.5 に示すような単純なデータ構造で実装している．

既存浮動小数点数を複数組み合わせて仮数部だけでなく，指数部も符号部も含めて一つのコンポーネントとして利用して正確な多倍長精度浮動小数点数を表現することから，多数桁方式とは異なるマルチコンポーネント (multi-component) 方式と呼ぶ．もちろん，マルチコンポーネント方式でも QD 以上の任

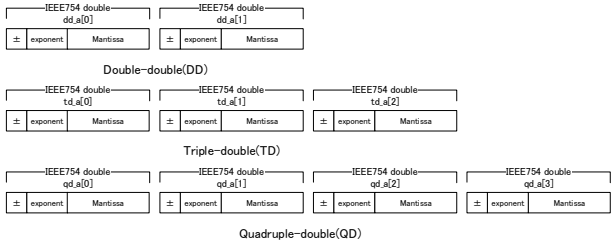


Fig. 5: DD, TD, QD データ型

意精度を実現することができ，実装系としては CAMPARY[9] が存在しているが，QD 以上の精度の場合，高速な mpn カーネル依存の MPFR より高速にすることは，現状では難しいようである．

逆に QD 以下の精度であれば，MPFR 212 bits 計算よりも四則演算，特に加減乗算は高速になる．実際，演算単位の速度比較を行うと，Fig.6 に示す通り，DD 精度では MPFR 106 bits 計算より圧倒的に高速であり，QD 演算では除算を除いて MPFR 212 bits 計算より高速になる．我々の実装で比較すると，TD 演算も MPFR 159bits 計算で高速であることが判明している．

以上示してきたように，仮数部長 212 bits 以下の計算ではマルチコンポーネント方式が，多数桁方式に比べて優位であるが，Python では高速なマルチコンポーネント方式の多倍長精度ライブラリが実装されていない．DD 精度相当の実装があるにはあるが，高速性を意識しているとは思えないものなので，実用的に gmpy2 よりも高速な Python ライブラリは，2020 年 2 月末現在，広く公開されているものは存在していないと言える．

## 4. RDD ライブラリの実装とベンチマークテスト

我々は MPFR や QD をベースとした多倍長精度行列乗算ライブラリ [15] を作成し公開している．現在，さらなる高速化，特に DD,QD 演算を高速化し，QD ライブラリには存在しない TD 演算の機能を付加した C ベースの RDD ライブラリを用いた新しいバージョンを開発中で，定評のある C++ ベースの MPLAPACK[11] より高速な行列乗算が可能であることを示している [17]．

RDD ライブラリのソフトウェア階層を Fig.7 に示す．最下層の `c_dd_qd.h` は，QD ライブラリの C インターフェース互換の DD 演算，QD 演算用のヘッダファイルであるが，QD ライブラリのものは C++クラスライブラリをベースに C 用の API を提供するだけであるのに対し，我々の方はこのヘッダファイル単独で DD, TD, QD 演算が C コンパイラだけで実現できるものになっている．rdd.h はこの `c_dd_qd.h` を土台にして，MPFR 関数と同じ引数順の，`r{dtq}d_`というプレフィックスの関数を提供するヘッダファイルであり，インライン関数として提供しているので，C プログラムでインクルードして使用すると，関数呼び出しのオーバーヘッドを極力減らすことができる．

### 4.1 RDD.py の構造

今回我々はこの RDD ライブラリを Python から呼び出せるよう，RDD パッケージを作成した．`rdd.h` のインライン関数を API 化するために `rdd.c` を用意し，`librdd.so` という DLL を作成して Python から呼び出し,`{dtq}d_float` という Python クラスから RDD ライブラリの演算機能を利用できるようにしたものである．ここではこれを RDD.py と呼ぶことにする．

Listing 7 に，RDD.py の冒頭と `dd_float`(DD 精度演算クラス) の抜粋を示す．ここではコンストラクタ (`__init__`)，`mpfr_set_dd` 関数 (DD 精度変数から MPFR 変数への変換)，文字列への変換 (`__str__`)，加算 (`__add__`) の定義を行っている．

Listing 7: RDD.py:マルチコンポーネント型多倍長精度演算パッケージ

```
# rdd.py

# RDD library
librdd = ct.cdll.LoadLibrary('../librdd.so');

# dd_float
class dd_float(ct.Structure):
  _fields_ = [('val', ct.c_double * 2)]

  # constructor
  def __init__(self, x0 = None, x1 = None):
    if x0 != None: self.val[0] = x0
    else: self.val[0] = 0.0
    if x1 != None: self.val[1] = x1
    else: self.val[1] = 0.0

  # mpfr_set_dd
  def mpfr_set_dd(self, prec = None):
    old_prec = gmpy2.get_context().precision
    if prec != None:
      gmpy2.get_context().precision = prec
    r = gmpy2.mpfr('0.0')
    mpfr_val = [gmpy2.mpfr('0.0'), gmpy2.mpfr('
        0.0')]
    mpfr_val[0] = gmpy2.mpfr(self.val[0])
    mpfr_val[1] = gmpy2.mpfr(self.val[1])

    r = mpfr_val[0] + mpfr_val[1]

    gmpy2.get_context().precision = old_prec

    return r

# print
  def __str__(self):
    tmp = self.mpfr_set_dd(53 * 2)
    return str(tmp)

  # addition
  def __add__(self, y):
    ret = dd_float()
    librdd.rdd_add(
      ct.byref(ret.val),
      ct.byref(self.val),
      ct.byref(y.val)
    )
    return ret
```

前述したように Python のクラス機能を利用した作りになっており，RDD.py の中では DD 精度だけでなく，TD 精度 (`td_float`)，QD 精度 (`qd_float`) も同様のやり方でクラス化してある．

### 4.2 C ベースの行列乗算機能と Python API 化

RDD.py でクラス化したマルチコンポーネント方式の多倍長精度演算のフォーマンスを調べるため，行列乗算によるベンチマークテストを行う．比較のため，RDD ライブラリに単純行列乗算アルゴリズムを用いた C の関数として`{dtq}d_matmul_simple` を作成した．Listing 8 に DD 精度の単純行列乗算の C コードを示す．RDD ライブラリ中の乗算 (`rdd_mul`) と加算 (`rdd_add`) 関数を利用している．

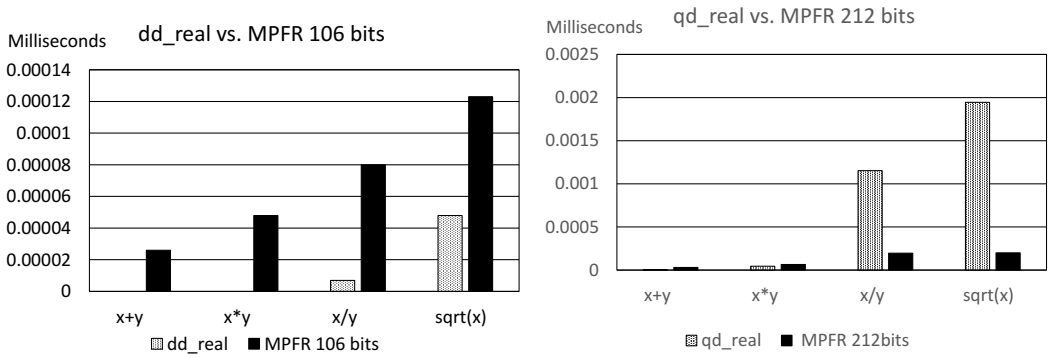


Fig. 6: 四則演算の速度比較: QD vs. MPFR

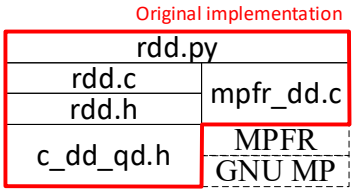


Fig. 7: RDD ライブラリのソフトウェア階層

Listing 8: DD 精度行列乗算関数

```
void dd_matmul_simple(double *ret, int
    ret_row_dim, int ret_col_dim, double *a,
    int a_row_dim, int a_col_dim, double *b,
    int b_row_dim, int b_col_dim)
{
  int i, j, k;
  int ij_index, ik_index, kj_index;
  double ab[DDSIZE];

  for(i = 0; i < ret_row_dim; i++)
  {
    for(j = 0; j < ret_col_dim; j++)
    {
      ij_index = (i * ret_col_dim + j) * DDSIZE;
      rdd_set0(&ret[ij_index]);

      for(k = 0; k < a_col_dim; k++)
      {
        ik_index = (i * a_col_dim + k) * DDSIZE;
        kj_index = (k * b_col_dim + j) * DDSIZE;
        rdd_mul(ab, &a[ik_index], &b[kj_index]);
              // ab := a[i,k] * b[k,j]
        rdd_add(&ret[ij_index], &ret[ij_index],
            ab); // ret[i,j] += ab
      }
    }
  }
}
```

RDD ライブラリに記述した C 関数を Python から呼び出すため，Listing 9 のように同名の Python 関数を RDD.py に付加して使用する．同様に TD 精度用の `td_matmul_simple` 関数，QD 精度用の `qd_matmul_simple` 関数も RDD.py に記述して使用できるようにした．

Listing 9: 行列乗算関数呼び出し (Python)

```
# dd_matmul_simple defined in rdd.c
def dd_matmul_simple(mat_c, c_row_dim, c_col_dim
    , mat_a, a_row_dim, a_col_dim, mat_b,
    b_row_dim, b_col_dim):
  librdd.dd_matmul_simple(
    mat_c, c_row_dim, c_col_dim,
    mat_a, a_row_dim, a_col_dim,
    mat_b, b_row_dim, b_col_dim
  )
```

### 4.3 正方行列乗算ベンチマークテスト

実装した RDD.py の性能を確認するため，実正方行列 $A = \sqrt{2}[i+j-1]_{i,j=1}^{n}$，$B = \sqrt{3}[\max(i,j)]_{i,j=1}^{n}$ を用いて，行列乗算 $C := AB$ の計算に要する時間を計測する．使用した Python スクリプト (`bench_matmul.py`) の DD 精度行列乗算ベンチマーク部分の抜粋を Listing 10 に示す．

Listing 10: 正方行列乗算ベンチマークスクリプト

```
# bench_matmul_dd.py

import ctypes as ct
import rdd
import gmpy2
import time

# gmpy2
gmpy2.get_context().precision = 212 # in bits
print(gmpy2.get_context())

mpfr_sqrt2 = gmpy2.sqrt(gmpy2.mpfr(2))
mpfr_sqrt3 = gmpy2.sqrt(gmpy2.mpfr(3))

# RDD library
librdd = ct.cdll.LoadLibrary('../librdd.so');

# start
librdd.rdd_start();

# 自作行列乗算
def xd_mymatmul(mat_a, row_dim_mat_a,
    col_dim_mat_a, mat_b, row_dim_mat_b,
    col_dim_mat_b, xd_zero):
  row_dim , mid_dim = row_dim_mat_a,
      col_dim_mat_a
  mid_dim_b, col_dim = row_dim_mat_b,
      col_dim_mat_b

  zero = xd_zero

  if mid_dim != mid_dim_b:
    print('A\'s␣col_dim␣=␣', mid_dim, ',␣B\'s␣
        row_dim␣=␣', mid_dim_b, '␣are␣mismatched
        !.')
    return [zero]

  mat_c = [zero] * (row_dim * col_dim)

  for i in range(0, row_dim):
```

```
    for j in range(0, col_dim):
      ij_index = i * col_dim + j
      mat_c[ij_index] = zero
      for k in range(0, mid_dim):
        ik_index = i * mid_dim + k
        kj_index = k * mid_dim + j
        mat_c[ij_index] += mat_a[ik_index] *
            mat_b[kj_index]

  return mat_c

# main loop
for sq_dim in [32, 64, 128, 256]:

  # dimension
  row_dim = sq_dim
  mid_dim = sq_dim
  col_dim = sq_dim

  dd_zero = rdd.dd_float(0.0, 0.0)

  # 行列データ型 DD
  # A := sqrt(2) * [(i + j + 1)]
  dd_mat_a = [dd_zero] * (row_dim * mid_dim)

  ptr_dd_mat_a = (ct.c_double * (row_dim *
      mid_dim * 2))()

  for i in range(row_dim):
    for j in range(mid_dim):
      ij_index = i * mid_dim + j
      dd_mat_a[ij_index] = rdd.mpfr_get_dd(
          mpfr_sqrt2 * (i + j + 1))

      ptr_dd_mat_a[(ij_index) * 2] = dd_mat_a[
          ij_index].val[0]
      ptr_dd_mat_a[(ij_index) * 2 + 1] =
          dd_mat_a[ij_index].val[1]

  # B := sqrt(3) * [max(i + 1, j + 1)]
  dd_mat_b = [dd_zero] * (mid_dim * col_dim)

  ptr_dd_mat_b = (ct.c_double * (mid_dim *
      col_dim * 2))()
  ptr_dd_mat_c = (ct.c_double * (row_dim *
      col_dim * 2))()

  for i in range(mid_dim):
    for j in range(col_dim):
      ij_index = i * col_dim + j
      if i > j:
        dd_mat_b[ij_index] = rdd.mpfr_get_dd(
            mpfr_sqrt3 * (i + 1))
      else:
        dd_mat_b[ij_index] = rdd.mpfr_get_dd(
            mpfr_sqrt3 * (j + 1))

      ptr_dd_mat_b[(ij_index) * 2] = dd_mat_b[
          ij_index].val[0]
      ptr_dd_mat_b[(ij_index) * 2 + 1] =
          dd_mat_b[ij_index].val[1]

  # dd
  start_time = time.time()
  dd_mat_c = xd_mymatmul(dd_mat_a, row_dim,
      mid_dim, dd_mat_b, mid_dim, col_dim,
      dd_zero)
  end_time = time.time()
  dd_matmul_time = end_time - start_time

  # ptr_dd
  start_time = time.time()
  rdd.dd_matmul_simple(ptr_dd_mat_c, row_dim,
      col_dim, ptr_dd_mat_a, row_dim, mid_dim,
      ptr_dd_mat_b, mid_dim, col_dim)
  end_time = time.time()
  ptr_dd_matmul_time = end_time - start_time

  print('dim␣=␣', sq_dim, f',␣␣␣␣␣dd:␣{
      dd_matmul_time:5.3f}')
  print('dim␣=␣', sq_dim, f',␣ptr_dd:␣{
      ptr_dd_matmul_time:5.3f}')

  # delete
  del dd_mat_a, dd_mat_b, dd_mat_c
  del ptr_dd_mat_a, ptr_dd_mat_b, ptr_dd_mat_c

librdd.rdd_end();
```

比較対象は

**xd_mymatmul** ⋯ binary64(Double)，DD, TD, QD, mpmath(mpf), gmpy2(mpfr) の演算子を用いた単純行列乗算関数

**{dtq}d_matmul_simple** ⋯ C による行列乗算実装に基づく単純行列乗算関数

の二つである．計算環境は Intel Core i7-9700K (3.6GHz), 16 GB RAM, Ubuntu 18.04.2 x86_64, GCC 7.3.0, MPFR 4.0.2[12]/GMP 6.1.2[14], QD 2.3.22[1] である．Python は 3.6.9 を使用し，mpmath は gmpy2 をバックエンドの自然数カーネルとして使用している．

計算時間の一覧を Table 1 に示す．各行列サイズ ($n$) に対し，上段は `xd_mymatmul` の計算時間，下段の “C” 行は C 実装行列乗算利用時 (`{dtq}d_matmul_simple`) の計算時間を示す。

Table 1: 正方行列乗算の計算時間 (秒)

| | | | | | 212 bits | |
|---|---|---|---|---|---|---|
| $n$ | Double | DD | TD | QD | mpmath | gmpy |
| 32 | 0.004 | 0.067 | 0.079 | 0.092 | 0.053 | 0.013 |
| | C | 0.001 | 0.006 | 0.011 | | |
| 64 | 0.032 | 0.536 | 0.635 | 0.731 | 0.422 | 0.107 |
| | C | 0.007 | 0.047 | 0.084 | | |
| 128 | 0.270 | 4.258 | 5.068 | 5.798 | 3.406 | 0.858 |
| | C | 0.063 | 0.377 | 0.669 | | |
| 256 | 2.214 | 34.82 | 41.90 | 49.87 | 28.052 | 7.135 |
| | C | 0.561 | 3.034 | 5.331 | | |

倍精度 (binary64) が最も高速であるのは当然であるが，実際には最適化された Intel Math Kernel や OpenBLAS では更に高速になる．それでも，DD, TD, QD, mpmath, gmpy2, いずれの多倍長精度計算結果より高速であることがわかる．多倍長精度計算の中では gmpy2 の高速性が際立つが，C 実装に基づく実装であれば DD, TD, QD 精度の方が全て高速になってることもわかる．

反面，演算子を定義した Python クラスに基づく行列乗算は非常に低速で，DD, TD, QD 精度は同じくクラスを使用している mpmath より低速である．これらの固定精度マルチコンポーネント型演算は，演算そのものは比較的軽く，関数呼び出しのオーバーヘッドの方がそれに比して大きくなる．演算子は行列要素ごとに呼び出されるので，このオーバーヘッドが累積して計算時間を伸ばしている．gmpy2 では演算子は呼び出しをベースの C ライブラリから行っており，この点，Python スクリプトのオーバーヘッドによる影響を回避できているものと思われる．

## 5. まとめと今後の展開

現在のコンピュータ上で，IEEE754-1985 倍精度 (binary64) を超える精度の浮動小数点演算を行うためには，GMP の高速な mpn カーネルに基づいた多数桁方式の MPFR ライブラリか，マルチコンポーネント方式の固定精度ライブラリ QD を用いることが，高速性と信頼性を確保するためには望ましい．

2020 年 2 月現在の Python 環境では多数桁方式の gmpy2 が最も高性能であり，QD に相当する強力なマルチコンポーネント方多倍長精度演算パッケージは存在していない．今回我々はマルチコンポーネント方式の C ライブラリである RDD に基づく RDD.py を Python パッケージとして作成し，その結果を正方行列乗算ベンチマークで示した．結果として，ネイティブの行列乗算を C ライブラリとして作成し，それを Python から呼び出せるようにすることで，Python クラス化した物より高速に実行できることが示された．また，QD 以下の計算精度では，やはりマルチコンポーネント方式の方が優位であることが，Python 上でも判明した．

今後の課題としては，RDD ライブラリを組み込んだ BNCmatmul ライブラリの新版を公開し，しかるのちに RDD ライブラリを分離させ，単独の Python パッケージとして利用できるように環境を整えることがあげられる．合わせて RDD ライブラリの高速化も，C の SIMD 関数を利用するなどして図っていきたい．

### 参考文献


1) D.H. Bailey. QD. `http://crd.lbl.gov/~dhbailey/mpdist/`.
2) T. J. Dekker. A floating-point technique for extending the available precision. *Numerische Mathematik*, Vol. 18, No. 3, pp. 224–242, Jun 1971.
3) Andreas Enge, Philippe Théveny, and Paul Zimmermann. MPC. `http://www.multiprecision.org/mpc/`.
4) N. Fabiano, J-. M. Muller, and J. Picot. Algorithms for triple-word arithmetic. *IEEE Trans. on Computers*, Vol. 68, pp. 1573–1583, 2019.
5) ヘンリッチ, 清水留三郎・小林光夫訳. 計算機による常微分方程式の解法 I, II. サイエンス社, 1973.
6) Nicholas J. Higham and Theo. Mary. A new approach to probabilistic rounding error analysis. *SIAM Journal on Scientific Computing*, Vol. 41, No. 5, pp. A2815–A2835, 2019.
7) Case Van Horsen. General multi-precision arithmetic for python 2.6+/3+ (gmp, mpir, mpfr, mpc). `https://github.com/aleaxit/gmpy`.
8) Fredrik Johansson, et al. *mpmath: a Python library for arbitrary-precision floating-point arithmetic (version 0.18)*, December 2013. `http://mpmath.org/`.
9) M. Jolders, J-. M. Muller, V. Popescu, and W.Tucker. Campary: Cuda mutiple precision arithmetic library and applications. *5th ICMS*, 2016.
10) 幸谷智紀, 永坂秀子. IEEE754 規格を利用した丸め誤差の測定法について. 日本応用数理学会論文誌, Vol. 7, No. 1, pp. 79 – 89, 1997.
11) MPLAPACK/MPBLAS. Multiple precision arithmetic LAPACK and BLAS. `http://mplapack.sourceforge.net/`.
12) MPFR Project. The MPFR library. `https://www.mpfr.org/`.
13) R.T.Kneusel. *Numbers and Computers*. Springer, 2015.
14) T.Granlaud and GMP development team. The GNU Multiple Precision arithmetic library. `https://gmplib.org/`.
15) T.Kouya. BNCmatmul. `http://na-inet.jp/na/bnc/bncmatmul-0.2.tar.bz2`.
16) 幸谷智紀. 多倍長精度数値計算. 森北出版, 2019.
17) 幸谷智紀. 3 倍精度行列乗算の性能評価. 第 173 回 HPC 研究会, 2020.