精度保証付き多倍長演算の新手法

電気通信大学　松田 望

2011/12/10

概要

- 精度保証付き数値計算には、通常、区間演算を用いる。
- 区間演算の実装には、下端・上端方式と中心値・半径方式がある。

$$[9,\ 11] = \langle 10,\ 1 \rangle$$

- 実装の容易な下端・上端方式が一般的。
- 多倍長の場合、実装方法を工夫すれば、中心値・半径方式の方が、計算速度・メモリ効率の両面で有利になる。
- 実際にライブラリを作成し、中心値・半径方式の有利を確かめた。

既存の多倍長演算ライブラリ

GMP(GNU Multi-Precision Library)

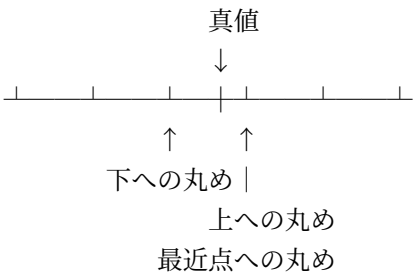
- 最も広く使われている多倍長演算ライブラリ。
- Mathematica でも使用。
- 四則演算と平方根のみ実装。
- 精度に応じてアルゴリズムを使い分ける。

MPFR

- GMP を用いて correct rounding を実現。
- 丸めの方向制御が可能。
- 豊富な数学関数を実装。

correct rounding とは

- 計算結果の真値と近似値の間に他の浮動小数点数がない。
- IEEE 754 の浮動小数点数は、四則演算と平方根について correct rounding 。



MPFI

- MPFR を用いて精度保証付き多倍長演算を実現。
- 区間の下端と上端を多倍長で保持。
- 使用メモリは MPFR の約 2 倍。
- 計算時間は MPFR の約 2 倍。

下端・上端方式での区間演算

$$0 \leq \underline{a} \leq \overline{a}$$
$$0 \leq \underline{b} \leq \overline{b}$$

のとき、

$$\begin{aligned}
[\underline{a},\ \overline{a}] + [\underline{b},\ \overline{b}] &= [\underline{a} + \underline{b},\ \overline{a} + \overline{b}] \\
[\underline{a},\ \overline{a}] - [\underline{b},\ \overline{b}] &= [\underline{a} - \overline{b},\ \overline{a} - \underline{b}] \\
[\underline{a},\ \overline{a}] \cdot [\underline{b},\ \overline{b}] &= [\underline{a}\underline{b},\ \overline{a}\overline{b}] \\
[\underline{a},\ \overline{a}] / [\underline{b},\ \overline{b}] &= [\underline{a}/\overline{b},\ \overline{a}/\underline{b}] \quad (\underline{b} > 0) \\
\sqrt{[\underline{a},\ \overline{a}]} &= \left[\sqrt{\underline{a}},\ \sqrt{\overline{a}}\right]
\end{aligned}$$

乗算・除算には、符号によって場合分けが必要。

INTLAB(INTerval LABoratory)

- MATLAB 上で精度保証付き区間演算を実現。
- 主な機能は、倍精度の精度保証付き数値計算。
- 倍精度の実数は下端・上端方式、複素数は中心値・半径方式。
- 補助的に精度保証付き多倍長実数も実装。
- 多倍長実数区間は中心値・半径方式。

INTLAB における精度保証付き多倍長演算

$$\begin{aligned} \text{中心値} \quad &: \quad \texttt{x.sign} \sum_{k=1}^{n} \{\texttt{x.mantissa}(k) \times 2^{-23k}\} \times 2^{23\texttt{x.exponent}} \\ \text{半径} \quad &: \quad \texttt{x.error.mant} \times 2^{23\texttt{x.error.exp}} \end{aligned}$$

- $\texttt{x.mantissa}(k)$, $\texttt{x.error.mant}$ は倍精度実数。
- 中心値は多倍長、半径は倍精度。
- $\texttt{x.mantissa}(k)$ の 64 ビットの内 23 ビットしか使っていない。
- 四則演算のみ実装。
- 乗算・除算は簡単だが誤差の大きい手法。

なぜ半径が低精度でも良いのか

- 中心値に対して半径が小さければ、半径は低精度で十分。
- 半径が大きい区間は、多倍長で表す意味がない。

$X = [-0.2,\ 0.2]$ のとき、

$$Y = (X+1)(X-1)(X-3)$$

の真の解は、

$$Y \subseteq [2.688,\ 3.079201435679]$$

である。しかし、これを通常の機械区間演算で計算すると、結果は

$$Y \subseteq [1.792,\ 4.608]$$

となり、半径が大きく拡大する。一方、式を

$$Y = X^3 - 3X^2 - X + 3$$

と展開してから計算すると、結果は

$$Y \subseteq [2.656,\ 3.224]$$

となり、半径の拡大を抑えることができる。

開発したライブラリ

概要

- C++ のライブラリである。
- 多倍長実数クラス LongFloat を提供する。
- 精度保証付き多倍長区間クラス LongInterval を提供する。
- 区間値を整数型の中心値・半径方式で保持する。
- 中心値は多倍長精度、半径は低精度で保持する。
- 多倍長演算の精度を任意に設定できる。
- 多倍長クラスはプログラム上、通常の「数」とほぼ同じように扱える。
- 点の演算について correct rounding である。

クラス構造

多倍長実数クラス LongFloat

```
class LongFloat{
  int sign;              // 符号
  int size;              // 仮数部配列のサイズ
  uint32 *mantissa;      // 仮数部配列へのポインタ
  int exponent;          // 指数部
};
```

- uint32 は 32 ビット符号なし整数。

$$\mathtt{x} = \mathtt{x.sign} \sum_{k=0}^{\mathtt{x.size}-1} (\mathtt{x.mantissa}[k] \times 2^{-32k}) \times 2^{32\mathtt{x.exponent}}$$

精度保証付き多倍長区間クラス LongInterval

```
class LongInterval{
  LongFloat center;      // 中心値
  Radius radius;         // 半径
};
```

$$\mathtt{x} = \langle \mathtt{x.center},\ \mathtt{x.radius} \rangle$$

区間半径クラス Radius

```
class Radius{
  uint32 mantissa;       // 仮数部
  int exponent;          // 指数部
};
```

$$\mathtt{x} = \mathtt{x.mantissa} \times 2^{\mathtt{x.exponent}}$$

- Radius の演算は、全て上への丸めで行われる。
- LongInterval の演算で発生した誤差は、全て radius に取り込む。

中心値・半径方式での区間演算

$$
\begin{aligned}
\langle c_1,\ r_1\rangle + \langle c_2,\ r_2\rangle &= \langle c_1 + c_2,\ r_1 + r_2\rangle \\
\langle c_1,\ r_1\rangle - \langle c_2,\ r_2\rangle &= \langle c_1 - c_2,\ r_1 + r_2\rangle
\end{aligned}
$$

乗算 (INTLAB)

$$\langle c_1,\ r_1\rangle \cdot \langle c_2,\ r_2\rangle \ \subseteq\ \langle c_1c_2,\ |c_1|r_2 + |c_2|r_1 + r_1r_2\rangle$$

乗算 (新手法)

$c_1 \geq 0,\ c_2 \geq 0$ のとき、$c_1 \geq r_1,\ c_2 \geq r_2$ ならば、

$$\langle c_1,\ r_1\rangle \cdot \langle c_2,\ r_2\rangle \ =\ \langle c_1c_2 + r_1r_2,\ c_1r_2 + c_2r_1\rangle$$

$c_1r_2 \geq c_2r_1$ ならば、

$$\langle c_1,\ r_1\rangle \cdot \langle c_2,\ r_2\rangle \ =\ \langle c_1c_2 + c_2r_1,\ c_1r_2 + r_1r_2\rangle$$

$c_1r_2 < c_2r_1$ ならば、

$$\langle c_1,\ r_1\rangle \cdot \langle c_2,\ r_2\rangle \ =\ \langle c_1c_2 + c_1r_2,\ c_2r_1 + r_1r_2\rangle$$

点の除算

- a/b を求めるには、 b の逆数の近似値 $\bar{b}$ を求め、 $a\bar{b}$ を計算する。
- $\bar{b}$ を求めるには、以下の反復計算を用いる。

$$\bar{b}_{k+1} = (2 - b\bar{b}_k)\bar{b}_k$$

- 最初に b を一度、倍精度数に変換し、その逆数を求める。この値を反復計算の初期値 $\bar{b}_0$ とする。
- b の値が倍精度数で表せる範囲を超えている場合、工夫が必要になる。
- $\bar{b}$, $a\bar{b}$ の計算では、それぞれ誤差が発生する。
- a/b を correct rounding で計算するためには、一時的に多倍長演算の精度を上げる必要がある。

区間の除算

$\langle c,\ r \rangle$ の逆数は、

$$\frac{1}{\langle c,\ r \rangle} = \frac{\langle c,\ r \rangle}{c^2 - r^2}$$

と表せるので、

$$\frac{1}{c^2 - r^2}$$

の精度保証ができれば、除算の精度保証ができる。

初めに、

$$w = c^2 - r^2$$

を計算し、その近似値を z 、誤差を ε_1 とする。

$$z = w + \varepsilon_1$$

次に、

$$\bar{z}_{k+1} = (2 - z\bar{z}_k)\bar{z}_k$$

の反復計算によって、$1/z$ の近似値 $\bar{z}$ を求める。

ここで発生する誤差を以下の ε_2 とする。

$$z\bar{z} = 1 + \varepsilon_2$$

以上より、$\bar{z}$ の $1/w$ に対する誤差 δ は以下の式で表わされる。

$$\begin{aligned}
\delta &= \frac{1}{w} - \bar{z} \\
&= \frac{1}{z - \varepsilon_1} - \frac{1 + \varepsilon_2}{z} \\
&= \frac{\varepsilon_1 - (z - \varepsilon_1)\varepsilon_2}{z(z - \varepsilon_1)}
\end{aligned}$$

実際の計算では、誤差は

$$\begin{aligned} \varepsilon_1 &\in [-e_1,\, e_1], \quad e_1 > 0 \\ \varepsilon_2 &\in [-e_2,\, e_2], \quad e_2 > 0 \end{aligned}$$

のように区間包囲されるので、δ は以下の式で区間包囲される。

$$\delta \in \frac{[-e_1,\, e_1] - (z - [-e_1,\, e_1])[-e_2,\, e_2]}{z(z - [-e_1,\, e_1])}$$

δ の計算に除算が必要になるが、ここではそれほど高い精度は必要ないので、倍精度での精度保証付き計算を行えば十分である。

点の平方根

- $\sqrt{x}$ を求めるには、 $1/\sqrt{x}$ の近似値 y を求め、 xy を計算する。
- y を求めるには、以下の反復計算を用いる。

$$y_{k+1} = \frac{1}{2}(3 - xy_k^2)y_k$$

- $\sqrt{x}$ ではなく $1/\sqrt{x}$ を求めるのは、計算過程の除算を減らすためである。
- y, xy の計算では、それぞれ誤差が発生する。
- $\sqrt{x}$ を correct rounding で計算するためには、一時的に多倍長演算の精度を上げる必要がある。

区間の平方根

区間の平方根の計算には、以下の式を用いる。

$$\begin{aligned}\sqrt{\langle c,\ r\rangle} &\subseteq \left\langle \sqrt{c},\ \sqrt{c}-\sqrt{c-r} \right\rangle \\ &= \left\langle \sqrt{c},\ \frac{r}{\sqrt{c}+\sqrt{c-r}} \right\rangle\end{aligned}$$

- 中心値の計算で発生した誤差は、半径に加える。
- 半径の式を変形するのは、桁落ちを防ぐためである。
- 半径の計算には、それほど高い精度は必要ないので、倍精度での精度保証付き計算を行えば十分である。

数値例

漸化式

$$a_{n+2} = \frac{34}{11}a_{n+1} - \frac{3}{11}a_n$$
$$a_0 = 1, \quad a_1 = \frac{1}{11}$$

の一般解は、

$$a_n = \frac{1}{11^n}$$

である。

しかし、この漸化式は丸め誤差が累積しやすく、普通に計算すると結果が発散する。

n	真値 a_n	倍精度	多倍長区間 (100 桁)		相対誤差
			中心値	半径	半径 / 中心値
5	6.21×10^{-6}	6.21×10^{-6}	6.21×10^{-6}	3.18×10^{-106}	5.13×10^{-101}
10	3.86×10^{-11}	3.87×10^{-11}	3.86×10^{-11}	1.03×10^{-103}	2.67×10^{-93}
15	2.39×10^{-16}	4.69×10^{-11}	2.39×10^{-16}	3.33×10^{-101}	1.39×10^{-85}
20	1.49×10^{-21}	1.14×10^{-8}	1.49×10^{-21}	1.08×10^{-98}	7.26×10^{-78}
25	9.23×10^{-27}	2.77×10^{-6}	9.23×10^{-27}	3.49×10^{-96}	3.78×10^{-70}
30	5.73×10^{-32}	6.72×10^{-4}	5.73×10^{-32}	1.13×10^{-93}	1.97×10^{-62}

速度比較

10 進 100 桁

	加算	減算	乗算	除算	平方根
GMP	6.79e-8	7.38e-8	2.45e-7	5.99e-7	9.58e-7
MPFR	8.09e-8	6.19e-8	2.33e-7	5.58e-7	9.22e-7
MPFI	1.82e-7	2.82e-7	4.77e-7	1.37e-6	1.85e-6
MPFR / GMP	1.19	8.39e-1	9.51e-1	9.32e-1	9.62e-1
MPFI / MPFR	2.25	4.56	2.05	2.46	2.01
LongFloat	2.48e-7	2.99e-7	1.93e-6	1.46e-5	3.41e-5
LongInterval	2.88e-7	3.46e-7	3.13e-6	3.31e-5	3.70e-5
LongInterval / LongFloat	1.16	1.16	1.62	2.27	1.09

10 進 1000 桁

	加算	減算	乗算	除算	平方根
GMP	1.62e-7	1.67e-7	8.76e-6	1.39e-5	1.42e-5
MPFR	2.33e-7	1.68e-7	7.54e-6	1.76e-5	1.40e-5
MPFI	4.79e-7	5.49e-7	1.52e-5	3.56e-5	2.80e-5
MPFR / GMP	1.44	1.01	8.61e-1	1.27	9.86e-1
MPFI / MPFR	2.06	3.27	2.02	2.02	2.00
LongFloat	1.21e-6	1.59e-6	1.16e-4	6.08e-4	2.90e-3
LongInterval	1.19e-6	1.65e-6	1.23e-4	1.11e-3	2.94e-3
LongInterval / LongFloat	9.83e-1	1.04	1.06	1.83	1.01

10 進 10000 桁

	加算	減算	乗算	除算	平方根
GMP	1.02e-6	1.02e-6	2.61e-4	5.44e-4	4.97e-4
MPFR	1.74e-6	1.26e-6	2.42e-4	6.67e-4	4.97e-4
MPFI	3.62e-6	3.39e-6	4.86e-4	1.34e-3	9.98e-4
MPFR / GMP	1.71	1.24	9.27e-1	1.23	1.00
MPFI / MPFR	2.08	2.69	2.01	2.01	2.01
LongFloat	1.06e-5	1.44e-5	1.12e-2	5.63e-2	3.82e-1
LongInterval	1.04e-5	1.48e-5	1.13e-2	1.02e-1	3.82e-1
LongInterval / LongFloat	9.81e-1	1.03	1.01	1.81	1.00

- 下端・上端方式では、区間演算には点演算の約 2 倍の時間がかかる。
- 中心値・半径方式では、区間の加減乗算にかかる時間は、点演算とほぼ同じ。
- 時間のかかる区間の除算も、点演算の約 1.8 倍の時間で可能。
- 新ライブラリの計算速度自体は、既存のライブラリに劣る。
- 既存のライブラリに新手法を組み込むことによって、改善が期待される。

今後の課題

- 新ライブラリで用いた手法を、既存のライブラリに組み込む。
- 反復計算による除算は効率が悪い。筆算にすべきか。
- 各種数学関数についても、中心値・半径方式での効率的な手法を考案する。
- 複素数に対応する。