中心値・半径方式による精度保証付き多倍長区間演算ライブラリの開発

松田 望

2016 年 3 月

電気通信大学 電気通信学研究科
博士 (工学) の学位申請論文

中心値・半径方式による精度保証付き多倍長区間演算ライブラリの開発

論文審査委員会
主査　山本野人　教授
委員　仲谷栄伸　教授
委員　緒方秀教　教授
委員　山本有作　教授
委員　小山大介　助教
委員　渡部善隆　准教授 (九州大学)

Development of interval multiple precision arithmetic library with center-radius form

MATSUDA Nozomu

Abstract

We discuss methods of multiple-precision arithmetic with verified numerics and development of a library for those methods.

Rounding errors in calculation of floating point arithmetic are usually small within a single operation but may be getting so large during successive operations and may give a serious influence to the results. In order to reduce the influence of rounding errors, multiple-precision arithmetic is a strong countermeasure widely used. Indeed the more precision we use, the more accuracy we obtain. However multiple-precision arithmetic by itself does not tell us whether the precision is too less nor too much. To verify that the results have expected accuracy and to avoid consuming too much computational resources with too long digits, we need some additional method.

Combination of multiple-precision and verified numerics is a solution. Verified numerics based on interval arithmetic tells us which digits are accurate or contain errors. Interval arithmetic operates interval numbers instead of floating point numbers and returns intervals including the results of the operations. To express the influence of rounding errors, it employs so called machine intervals whose inferior and superior bounds are represented by floating point numbers together with control of rounding directions.

In implementation of machine interval arithmetic, it is usual way to keep two floating point numbers in the memory for the inferior and the superior bounds of an interval. Basic operations of intervals are constructed by operations for inferior bounds using rounding mode toward lower direction and operations for superior bounds with rounding mode toward upper direction. Therefore at least twice computational cost is necessary as compared with usual floating point operations. We call this expression the inf-sup form of intervals.

There is another way to install the machine interval arithmetic, so called the center-radius form. It uses two floating point numbers which indicate the center and the radius to keep an interval in the memory. Since operations of intervals are more complicated than the inf-sup form, there is no advantage so far as using double precision as usual. However in case of multiple-precision arithmetic, there is a possibility to get advantage of the center-radius form. If we use a floating point number of long digits for the center and short digits for the radius, we can save computational cost both for cpu time and memory. As is sure that there is an offset between such effects and complicateness of operations, we have to investigate actual implimentation of the center-radius form and compare it with the inf-sup form.

In the present paper, we develop technic of interval operations in the center-radius form, describe an implimentational structure of multiple-precision interval numbers, and construct a library for interval multiple-precision arithmetic with the

center-radius form. In conclusion we have shown an advantage of the center-radius form against the inf-sup form by numerical experiments using our library actually implimented.

中心値・半径方式による精度保証付き多倍長区間演算ライブラリの開発

松田 望

概 要

本論文は、精度保証付き数値計算法に基づく多倍長演算ライブラリの研究および実装を含む開発について論ずるものである。

計算機で扱える浮動小数点数は有限桁であるため、計算の過程で値を丸める必要がある。このとき発生するのが、丸め誤差である。一回の演算で発生する丸め誤差は小さく、計算結果にはほとんど影響を与えない。しかし、丸め誤差を含む計算結果を引き続く計算に用いると、誤差は伝播・拡大する。このため、最終的な計算結果が非常に大きな誤差を含むことがある。

丸め誤差の累積を防ぐ汎用的な手段として、多倍長演算がある。浮動小数点数の仮数部のビット数 (桁数) を大きくすれば、発生する丸め誤差は相対的に小さくなり、累積する誤差も小さくなる。どんなに丸め誤差が累積しやすい問題でも、演算の桁数を十分大きくすれば、理論上、誤差を小さく抑えることができる。

このように多倍長演算は、丸め誤差の累積を抑えるのに有効な手段であるが、それだけでは、計算結果がどれだけ改善したのかは分からない。その効果を検証するには、何らかの検算が必要である。また、桁数の大きな多倍長演算には多くの記憶領域が必要であり、計算時間もかかる。そこで、求めたい計算結果の精度に対して、演算の桁数を最小限に抑えることが望ましい。例えば、 10 進数 100 桁程度で計算すれば十分な問題を、 10 進数 1000 桁で計算するのは、計算機資源の無駄である。このような無駄を削減するには、累積した丸め誤差の大きさを知る必要がある。

浮動小数点数演算の丸め誤差を見積る手法として、精度保証付き数値計算がある。精度保証付き数値計算は、一般に機械区間演算によって実装される。区間演算とは、通常の数値の代わりに幅を持った区間同士の演算を行い、計算結果も区間で表す算法である。機械区間演算とは、計算機上の浮動小数点数を用いて実装された区間演算のことである。機械区間演算では、 CPU に組み込まれた丸めの方向制御命令などを用いて、発生しうる丸め誤差を包含する区間を計算結果とする。この精度保証付き数値計算を、多倍長演算と組み合わせ、丸め誤差の大きさを把握することが考えられる。

機械区間演算の実装では、区間の下端と上端を保持する方法が一般的である。下端・上端方式の機械区間演算の基本は、丸めの方向を変えて同じ計算を二回繰り返すことである。これによって、点 (半径を持たない通常の数) の演算の二倍の計算量で精度保証が行える。別の区間表現として、区間の中心値と半径を保持する方法がある。機械区間演算を多倍長に拡張した場合、実装方法を工夫することによって、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と考えられる。ここでの工夫とは、区間の中心値を多倍長で、半径を低精度で保持することである。半径の表現を低精度で済ませることによって、計算量を点の演算の二倍未満に抑えることができる。半径が低精度であることに起因する誤差もあるが、多倍長区間演算の場合、その影響はほとんど問題にならない。

このような実装方法を用いれば、中心値・半径方式の精度保証付き多倍長区間演算は、原理的には下端・上端方式より速くなるはずである。しかし、下端・上端方式

より実装が複雑になるというデメリットもあり、理論上の優位性が実現できるかどうかは、不確定であった。精度保証付き計算法の既存ライブラリである INTLAB は、このアイデアに基づいて実装された精度保証付き多倍長区間演算の機能を持つ。しかし、 INTLAB の多倍長区間演算機能は、ライブラリ内部で使用するために作られた簡易的なものであり、そのままでは実用的とは言えず、中心値・半径方式の優位性の検証に用いることはできない。本論文では、INTLAB のアイデアを発展させた新しい精度保証付き多倍長演算ライブラリ LILIB の開発を行い、これを実行することで、中心値・半径方式の優位性を検証している。LILIB には、データ構造の無駄の解消、乗算・除算のより誤差が小さいアルゴリズムの実装、平方根の精度保証の実装、厳密な区間の大小比較の実装などの、 INTLAB にはない特徴がある。

LILIB を用いた数値実験による検証の結果として以下のものを得ている。中心値の桁数が小さいときは、通常の点の演算の二倍以上の時間がかかる区間演算結果もあったが、中心値の桁数が大きければ、四則演算と平方根の計算について、点の演算の二倍未満の計算時間を達成できた。特に加減乗算に限れば、点の演算に数

既存の精度保証付き多倍長区間演算ライブラリとの比較では、現時点の LILIB の計算速度は、下端・上端方式を採用する MPFI よりも劣っている。これは、区間演算の土台になっている、点の演算速度の差によるものである。MPFI の点の演算の土台になっている GNU Multi-Precision Library(GMP) は、高速な多倍長演算アルゴリズムの選択と、アセンブラによる各種計算機への最適化により、非常に高速である。GMP の開発はチームプロジェクトであるので、現在の我々の研究態勢では、GMP の速度に追いつく多倍長演算プログラムを独自に開発することは困難であった。しかし、本論文の成果によって、精度保証付き多倍長区間演算に中心値・半径方式を用いることで下端・上端方式より高速化できることは確認できている。そこで、 LILIB のアルゴリズムを生かしつつ、点の演算部分を GMP に置き換えれば、既存のものより高速な精度保証付き多倍長演算が実現できるはずである。これについては引き続き開発を進めて行きたい。

今後の課題としては、上記の他、各種数学関数の整備・連立一次方程式への対応・微分方程式への対応などが挙げられる。

目 次

第1章　はじめに

1.1　背景

計算機上で実数を扱う場合、値を符号・仮数部・指数部に分けて表現するのが一般的である。こうして表現された値を浮動小数点数と呼ぶ。現在広く使われている浮動小数点数の実装は、 IEEE 754 [1] で定義されているものである。

浮動小数点数の精度は、仮数部のビット数によって決まる。例えば、 IEEE 754 の単精度数の仮数部は 23 ビット、倍精度数の仮数部は 52 ビット、四倍精度数の仮数部は 112 ビットである。ここで、配列変数を用いて仮数部のビット数を大きくすれば、より高精度の計算が可能になる。これが多倍長演算である。多倍長演算環境では、多くの場合、精度は任意に設定できる。このため、任意精度演算・任意精度多倍長演算などとも呼ばれる。なお、ハードウェアレベルで多倍長演算を実装する場合もあるが、本論文で扱うのは、ソフトウェアレベルでの多倍長演算である。

多倍長演算を用いる目的は、大きく二つに分けられる。一つは、精度の高い計算結果を得ることである。例えば、ある計算結果を 10 進数で 100 桁知りたいとき、 IEEE 754 の四倍精度では精度が足りず、多倍長演算を行う必要がある。

もう一つの目的は、大きな丸め誤差が発生する問題で、丸め誤差を小さく抑えることである。計算機で扱える浮動小数点数は有限桁であるため、計算の過程で値を丸める必要がある。このとき発生するのが、丸め誤差である。一回の演算で発生する丸め誤差は小さく、計算結果にはほとんど影響を与えない。しかし、丸め誤差を含む計算結果を引き続く計算に用いると、誤差は伝播・拡大する。このため、最終的な計算結果が非常に大きな誤差を含むことがある。例えば、ある計算結果を 10 進数で 10 桁知りたいが、丸め誤差が大きいので、 10 桁を正しく求めるために 100 桁の計算が必要になる場合がある。

このような丸め誤差の累積を防ぐ手段の一つに、アルゴリズムの工夫がある。数学的には同値な計算でも、計算の順番などによって、丸め誤差の累積のしやすさが変わる場合がある。しかし、このアルゴリズムの工夫を行うには、解こうとする問題の特性を良く理解する必要があり、また、全ての問題に対して良いアルゴリズムが見つかるとは限らない。

丸め誤差の累積を防ぐより汎用的な手段が、多倍長演算である。浮動小数点数の精度 (仮数部のビット数) を高くすれば、発生する丸め誤差は相対的に小さくなり、累積する誤差も小さくなる。どんなに丸め誤差が累積しやすい問題でも、浮動小数点数の精度を十分高くすれば、理論上、誤差を小さく抑えることができる。

一方、浮動小数点数演算の丸め誤差を見積る手法として、精度保証付き数値計算がある。精度保証付き数値計算は、一般に機械区間演算によって実装される。区間演算とは、通常の数値の代わりに幅を持った区間同士の演算を行い、計算結果も区間で表す算法である。機械区間演算とは、計算機上の浮動小数点数を用いて実装された区間演算のことであ

る。機械区間演算では、 CPU に組み込まれた丸めの方向制御命令などを用いて、発生しうる丸め誤差を包含する区間を計算結果とする。

現在、多倍長演算や精度保証付き数値計算を簡単に利用できる数値計算ライブラリが、多数提供されている。多倍長演算ライブラリとしては、 GNU Multi-Precision Library [2] ・ MPFR [3] ・ exflib [4] など、精度保証付き数値計算ライブラリとしては、 kv [5] ・ XSC [6] ・ PROFIL/BIAS [7] ・ CAPD library [8] ・ INTLIB [9] ・ INTLAB [10] などがある。

1.2　目的

計算の結果を一種の製品と考え、その品質を保証・管理する「計算の品質保証」「計算の品質管理」という考えがある [11] 。多倍長演算は、丸め誤差の累積を抑えるのに有効な手段であるが、それだけでは、計算結果がどれだけ改善したのかは分からない。その効果を検証するには、何らかの検算が必要である。検算によって、「計算の品質保証」を行うのである。

また、高精度の多倍長演算には多くの記憶領域が必要であり、計算時間もかかる。そこで、求めたい計算結果の精度に対して、多倍長演算の精度を最小限に抑えることが望ましい。例えば、 10 進数 100 桁程度の精度で計算すれば十分な問題を、 10 進数 1000 桁の精度で計算するのは、計算機資源の無駄である。このような無駄を把握するには、累積した丸め誤差の大きさを知る必要がある。精度と計算コストの兼ね合いをはかり、「計算の品質管理」を行うのである。

多倍長演算に精度保証付き数値計算を組み合わせ、丸め誤差の大きさを把握することが考えられる。精度保証付き多倍長演算を行えるライブラリとしては、 MPFI [12] ・ kv ・ C-XSC ・ exflib ・ INTLAB などがある。このうち INTLAB 以外は、区間の下端と上端を多倍長浮動小数点数で表す、「下端・上端方式」の多倍長区間演算を行う。一方、INTLAB での多倍長区間演算は、区間の中心値と半径を保持する「中心値・半径方式」である。

精度保証付き区間演算の実装では、下端・上端方式が一般的である。しかし、これを多倍長演算に拡張した場合、実装方法を工夫することによって、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。その原理については、 4.1.2 節で述べる。INTLAB の多倍長区間演算は、このアイデアに基づいて実装されている。しかし、クラス構造の無駄が多い、四則演算しか実装されていない、各演算が誤差の大きい簡易的なアルゴリズムで実装されているなど、実用的なものとは言えない。

中心値・半径方式の精度保証付き多倍長区間演算には、下端・上端方式より実装が複雑になるというデメリットもある。原理的には中心値・半径方式の方が有利であるが、実際に実装してみないと分からない部分も多い。我々は、 INTLAB のアイデアを発展させ、新しい精度保証付き多倍長区間演算ライブラリを開発した。本論文の目的は、このライブラリ開発を通して、中心値・半径方式での演算アルゴリズムを新たに構築し、それらの下端・上端方式に対する優位性を示すことである。この検証プロセスは、実装されたライブラリに大きく依存するものとなる。したがって本論文中に、ライブラリの入手法・使用法についての詳細な記述を行った。

1.3 論文の構成

本論文では、まず第 2 章で、本研究の背景となる、多倍長演算と精度保証付き数値計算について説明する。ここには、多倍長演算と精度保証付き数値計算を組み合わせる意義、本論文を理解する上で必要な知識、本論文と関連のある既存のライブラリの紹介などが含まれる。

次に、第 3 章で、我々が開発したライブラリ LILIB について説明する。ここでは、一般利用者が知るべき LILIB の使用方法、機能一覧などを紹介する。

第 4 章では、 LILIB 内部でのクラス構造、演算のアルゴリズムといった動作原理について説明する。

第 5 章では、実際に LILIB で行った計算を示し、その結果から示唆されることを考察する。

最後に、第 6 章では、 LILIB の開発を通して得られた知見や今後の課題についてまとめる。

第2章　多倍長演算と精度保証付き数値計算

2.1　精度保証付き多倍長演算の意義

多倍長演算は、丸め誤差の累積を抑えるのに有効な手段であるが、それだけでは、計算結果がどれだけ改善したのかは分からない。その効果を検証するには、何らかの検算が必要である。また、高精度の多倍長演算には多くの記憶領域が必要であり、計算時間もかかる。そこで、求めたい計算結果の精度に対して、多倍長演算の精度を最小限に抑えることが望ましい。例えば、 10 進数 100 桁程度の精度で計算すれば十分な問題を、 10 進数 1000 桁の精度で計算するのは、計算機資源の無駄である。このような無駄を把握するには、累積した丸め誤差の大きさを知る必要がある。

任意精度の多倍長演算が可能な環境では、計算過程で生じる丸め誤差の影響を、異なる桁数での計算を複数回行なうことで見積もることがある。しかし、この方法も常に適切な見積りを行なえる保証はない。その例として、以下に Rump の例題として知られる著名な例を示す [13] 。

$$a = 77617, \quad b = 33096$$

に対して

$$f = (333.75 - a^2)b^6 + a^2(11a^2b^2 - 121b^4 - 2) + 5.5b^8 + \frac{a}{2b}$$

を単精度・倍精度・四倍精度で計算すると表 2.1 の結果が得られる。計算結果は参考文献 [14] からの引用であり、計算環境は Hitachi SR16000L2 における Fortran プログラムである。

表 2.1: Rump の例題の計算結果

計算精度	f の計算結果
単精度	1.17260396
倍精度	1.17260394005317869
四倍精度	1.17260394005317863185883490452018 01

これを見れば、 f の真値は 1.1726039400531786 と 1.1726039400531787 の間にあると予想され、少なくとも

$$f = 1.1726039\cdots$$

となることは間違いないように思える。しかし、真値は、

$$f = \frac{a}{2b} - 2 = -\frac{54767}{66192} = -0.827396\cdots$$

である。つまり上記の結果は符号すら合っていない。この例は、単純な方法で多倍長演算の信頼性を検証することの困難さを示唆している。

このように、多倍長演算の過程で発生する丸め誤差の見積りを高い信頼性で得ようとすれば、単純な方法を用いることはできない。そこで、精度保証付き数値計算との組み合わせによって、多倍長演算の欠点を補うことが考えられる。

精度保証付き数値計算は、計算過程での打ち切り誤差および丸め誤差の影響を厳密に見積もる計算手法であり、一般に区間演算によって実現される。すなわち、通常の数値の代わりに区間値を用い、誤差の影響を区間幅として取り込む。発生した誤差の絶対値は不等式によって上から評価され、この不等式が成立することが数学的に保証される。この意味で「厳密な」誤差評価と呼ばれる。

しかし、精度保証付き数値計算は計算結果の精度を向上させるわけではない。誤差見積りが常に上からの評価として得られることから、過大評価が本質的に避けられない。精度を保証し、かつこれを向上させるには、多倍長演算との組み合わせが必要とされるのである。

多倍長演算に精度保証を導入するにあたって、特に注意を要する点は、区間演算の表現依存性である。区間演算の特性から、同値であっても異なる数学表現の式については、一般に計算結果が一致しない。すなわち、適切な数式表現を選択しないと、区間幅の過大評価が拡大する恐れがある。これについては、 4.2 節で具体的に例示する。このことは、精度保証付き多倍長演算を実際に利用する場面でも常に考慮する必要がある。また、精度保証付き多倍長演算そのものの設計段階においても、区間表現の違いによって発生する誤差が異なるため、十分な対応策が必要となってくる。これらについては、 2.3 節および 4.2 節で詳説する。

区間演算における区間の実装としては、区間の下端と上端を保持する方法、区間の中心値と半径を保持する方法の二つが考えられる。例えば、「下端が 9 、上端が 11 」の区間は、「中心値が 10 、半径が 1 」と表すこともできる。

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

このうち、より多く用いられているのは、実装の容易な下端・上端方式であり、これは精度保証付き多倍長区間演算でも同様である。

しかし、精度保証付き多倍長区間演算の場合、実装方法を工夫することによって、下端・上端方式よりも中心値・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。我々は、中心値・半径方式の精度保証付き多倍長区間演算に適した演算手法を考案し、四則演算と平方根の計算が可能なライブラリ LILIB を開発した。

2.2　浮動小数点数と多倍長数

この節では、倍精度浮動小数点数の構造の概略を確認し、多倍長数への拡張法を記す。

2.2.1 倍精度浮動小数点数

計算機上で実数を扱う場合、値を符号・仮数部・指数部に分けて表現するのが一般的である。こうして表現された値を浮動小数点数と呼ぶ。現在広く使われている浮動小数点数の実装は、 IEEE 754 で定義されているものである。

IEEE 754 では、単精度・倍精度・四倍精度などの浮動小数点数が定義されている。現在最も広く使われている倍精度浮動小数点数の内部構造は、以下のようなものである。

$$(-1)^{\mathtt{sign}} \times \mathtt{1.fraction} \times 2^{\mathtt{exponent}-1023}$$

表 2.2: 倍精度浮動小数点数の内部構造

変数	意味	ビット数
sign	符号	1
exponent	指数部	11
fraction	仮数部	52

「 1.fraction 」とは、 1 の小数点以下に fraction のビット列を並べた小数である。最初の 1 を省略することで、 52 ビットで 53 ビットの値を表現している。この方法は、「けち表現」と呼ばれる。

なお、上記の表現は正規化数と呼ばれるもので、 IEEE 754 ではそれ以外にも、絶対値が非常に小さい数を表す非正規化数、 0 や無限大を表す特殊な表現が定義されている。

2.2.2 丸めの方向

浮動小数点数は、数直線上の不連続な集合である。浮動小数点数の演算結果は必ずしも浮動小数点数とはならないので、浮動小数点数に丸める必要がある。このときの真値と近似値の差を、丸め誤差と呼ぶ。

IEEE 754 では、以下の五つの丸めを定義している。

- $-\infty$ への丸め (下への丸め)
- $+\infty$ への丸め (上への丸め)
- 0 方向への丸め
- 最近接丸め (偶数)
  最近点が二つある場合、最下位ビットが 0 になる方へ丸める。これは、計算の過程で丸め方向が上下のうち一方に偏るのを防ぐための措置である。
- 最近接丸め (0 から遠い方へ)
  最近点が二つある場合、 0 から遠い方へ丸める。

計算機では通常、最近接丸め (偶数) が用いられるが、利用者が任意に他の丸め方向に切り替えることができる。

2.2.3　ulp と計算機イプシロン

ある浮動小数点数の値に対して、その値と隣の値との差を、ulp(Units in the Last Place) と呼ぶ。基準となる値とその下の値との差、上の値との差が異なる場合、より大きい差を ulp とする。

ある浮動小数点数系に対して、 1 に対する ulp を計算機イプシロンと呼ぶ。例えば、

- IEEE 754 の単精度浮動小数点数の計算機イプシロンは、 2^{-23}
- IEEE 754 の倍精度浮動小数点数の計算機イプシロンは、 2^{-52}

である。

2.2.4　correct rounding

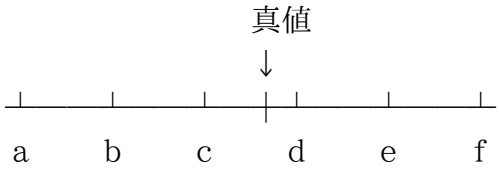
浮動小数点数の丸めを行ったとき、真値と近似値の間に他の浮動小数点数がない状態を、 “correct rounding” または “correctly rounded” と呼ぶ。

例えば、図 2.1 の状態を考える。 a 〜 f が浮動小数点数で、真値は c と d の間、 d 寄りにある。このとき、

- 下への丸めを行うと c に
- 上への丸めを行うと d に
- 最近接丸めを行うと d に

丸めるのが correct rounding である。

図 2.1: 浮動小数点数への丸め



correct rounding が実現されている場合、最近接丸めの誤差は 0.5 ulp 以下、それ以外の丸めの誤差は 1 ulp 以下となる。

IEEE 754 で定義されている単精度や倍精度の浮動小数点数演算は、 correct rounding の条件を満たしている。独自の浮動小数点数演算を実装するときも、この条件を満たすことが、演算精度の一つの目安になる。

2.2.5 多倍長数

浮動小数点数の精度は、仮数部のビット数によって決まる。
IEEE 754 の浮動小数点数の精度 (10 進数での有効桁数) を、表 2.3 に示す。

表 2.3: IEEE 754 の浮動小数点数の精度

浮動小数点数	仮数部のビット数	10 進数での有効桁数
単精度数	24	7.22
倍精度数	53	15.95
四倍精度数	113	34.02

配列変数を用いて仮数部のビット数を大きくすれば、より高精度の計算が可能になる。これが多倍長演算である。多倍長演算環境では、多くの場合、精度は任意に設定できる。このため、任意精度演算・任意精度多倍長演算などとも呼ばれる。

多倍長数の仮数部を表す配列変数の各要素を、 limb と呼ぶ。本論文で開発するライブラリ LILIB の場合、 32 ビット符号なし整数を limb として用いる。多倍長演算は基本的に、 limb を単位とした「筆算」によって実装される。さらに、単純な筆算より高速なアルゴリズムも知られている。例えば、多倍長の乗算の場合、 Karatsuba 法 [15] や高速 Fourier 変換を用いる方法 [16] がある。

2.3 精度保証付き区間演算

精度保証付き数値計算は、丸め誤差の影響を計算結果に取り込む計算法であり、一般に機械区間演算によって実装される。区間演算とは、通常の数値の代わりに幅を持った区間同士の演算を行い、計算結果も区間で表す算法である。機械区間演算とは、計算機上の浮動小数点数を用いて実装された区間演算のことである。機械区間演算では、 CPU に組み込まれた丸めの方向制御命令などを用いて、発生しうる丸め誤差を包含する区間を計算結果とする。

2.3.1 区間

二つの実数 $\underline{x}, \overline{x}$ で表される集合

$$X = [\underline{x},\ \overline{x}] = \{x \in R | \underline{x} \leq x \leq \overline{x}\}$$

を実数区間と呼ぶ。
実数区間 X の特性を表す値として、表 2.4 のようなものが挙げられる。

表 2.4: 実数区間 X の特性値

要素	内容
下端	$\underline{x}$
上端	$\overline{x}$
直径	$\overline{x} - \underline{x}$
半径	$(\overline{x} - \underline{x})/2$
中心値	$(\overline{x} + \underline{x})/2$

区間は、下端・上端によって表現する以外に、中心値・半径によって表現することもできる。

$$\langle c,\ r\rangle = \{x \in R | c - r \le x \le c + r\} \quad (c, r \in R,\ r \ge 0)$$

また、区間演算の特性を考える上で重要な指標が、相対誤差である。区間

$$\langle c,\ r\rangle \quad (c \ne 0)$$

の相対誤差は、

$$r/|c|$$

である。

相対誤差は、区間の中心値の有効桁数を示す指標になる。相対誤差が 10^{-n} であれば、中心値の上から $n+1$ 桁目辺りに誤差としての半径が作用するので、有効桁数はおよそ n と言える。一般に、計算結果の相対誤差が小さいほど、その区間演算は「精度が高い」と言える。ただし、意図的に中心値が 0 の区間を扱う場合などは、この限りではない。

2.3.2　下端・上端方式での区間演算

区間演算では、通常の数と同様に、区間を計算の対象とする。

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

のとき、下端・上端方式での四則演算と平方根の計算は、以下のようになる。

$$\begin{aligned}
[\underline{a},\ \overline{a}] + [\underline{b},\ \overline{b}] &= [\underline{a} + \underline{b},\ \overline{a} + \overline{b}] \\
[\underline{a},\ \overline{a}] - [\underline{b},\ \overline{b}] &= [\underline{a} - \overline{b},\ \overline{a} - \underline{b}] \\
[\underline{a},\ \overline{a}] \cdot [\underline{b},\ \overline{b}] &= [\underline{ab},\ \overline{a}\overline{b}] \\
[\underline{a},\ \overline{a}]/[\underline{b},\ \overline{b}] &= [\underline{a}/\overline{b},\ \overline{a}/\underline{b}] \\
\sqrt{[\underline{a},\ \overline{a}]} &= \left[\sqrt{\underline{a}},\ \sqrt{\overline{a}}\right]
\end{aligned}$$

ただし、これは丸め誤差を考慮していない式である。丸め誤差の存在する浮動小数点数演算で精度保証を行う場合、以下のように丸めの方向制御を用い、浮動小数点数で区間包囲を行う。

$$\begin{aligned}
[\underline{a},\ \overline{a}] + [\underline{b},\ \overline{b}] &\subseteq [\bigtriangledown(\underline{a}+\underline{b}),\ \bigtriangleup(\overline{a}+\overline{b})] \\
[\underline{a},\ \overline{a}] - [\underline{b},\ \overline{b}] &\subseteq [\bigtriangledown(\underline{a}-\overline{b}),\ \bigtriangleup(\overline{a}-\underline{b})] \\
[\underline{a},\ \overline{a}] \cdot [\underline{b},\ \overline{b}] &\subseteq [\bigtriangledown(\underline{a}\underline{b}),\ \bigtriangleup(\overline{a}\overline{b})] \\
[\underline{a},\ \overline{a}] / [\underline{b},\ \overline{b}] &\subseteq [\bigtriangledown(\underline{a}/\overline{b}),\ \bigtriangleup(\overline{a}/\underline{b})] \\
\sqrt{[\underline{a},\ \overline{a}]} &\subseteq \left[\bigtriangledown(\sqrt{\underline{a}}),\ \bigtriangleup(\sqrt{\overline{a}})\right]
\end{aligned}$$

$\bigtriangledown, \bigtriangleup$ は、「下への丸め」「上への丸め」を表す。

ここでは、非負の区間の計算についてのみ扱った。負の値を含む区間について計算する場合、乗算・除算には、符号による場合分けが必要になる。

下端・上端方式で区間を保持するには、点の場合の約 2 倍の記憶領域を必要とする。また、ほぼ同じ計算を 2 度繰り返すので、計算にかかる時間も点の場合の約 2 倍となる。

中心値・半径方式での区間演算については、 4.2 節で説明する。

2.4 既存の多倍長演算ライブラリ

ここでは、既存の多倍長演算ライブラリおよび、多倍長演算が可能な精度保証付き数値計算ライブラリを、簡単に紹介する。

2.4.1 GNU Multi-Precision Library

現在、最も広く使われている多倍長演算ライブラリは、 GNU Multi-Precision Library (GMP) [2] であろう。GMP は、任意精度の整数・有理数・浮動小数点数に対する四則演算・平方根の計算を行うライブラリである。標準で C 言語・C++ から利用できる他、様々な言語用のインターフェイスも開発されている。様々な計算機に対して、アセンブラによる最適化が行われており、非常に高速である。また、多倍長演算では、演算精度 (桁数) によって最適なアルゴリズムが異なることが知られているが、 GMP は精度に応じて最適なアルゴリズムを使い分ける。例えば乗算について、精度が低いときは筆算や Karatsuba 法、精度が高いときは高速 Fourier 変換による乗算など、複数のアルゴリズムを使い分ける。ただし、計算結果は correct rounding ではなく、誤差は不明確である。GMP は多くのソフトウェアに用いられている。例えば、 Mathematica で多倍長演算を行うと、内部で GMP が呼び出される。

2.4.2 MPFR

MPFR [3] は、 GMP を用いて実装された、任意精度浮動小数点数演算ライブラリである。標準で C 言語から利用できる他、様々な言語用のインターフェイスも開発されている。GMP の浮動小数点数演算にはない、以下のような特徴を持つ。

- IEEE 754 相当の丸めの方向制御ができる。
- 四則演算・平方根以外にも、三角関数など基本的なものから、 Gamma 関数・ Bessel 関数のような複雑なものまで、豊富な数学関数を実装している。
- 全ての演算・関数について correct rounding を実現している。

2.4.3　MPFI

MPFI [12] は、 MPFR を用いて実装された、精度保証付き多倍長区間演算ライブラリである。C 言語・ C++ から利用できる。下端・上端方式の区間演算ライブラリであり、区間の下端と上端をそれぞれ MPFR の任意精度浮動小数点数で保持する。このため、MPFR で同じ計算を行う場合の約 2 倍の記憶領域を必要とする。また、下端・上端方式での区間演算ではほぼ同じ計算を 2 度繰り返すので、計算にかかる時間も MPFR の約 2 倍となる。

2.4.4　kv

kv [5] は、様々な機能を持った C++ の数値計算ライブラリである。本研究に関係する機能としては、

- 倍精度の精度保証付き区間演算
- 倍精度の精度保証付き Affine 演算
- MPFR をベースにした精度保証付き多倍長区間演算
- MPFR をベースにした精度保証付き多倍長 Affine 演算

などが挙げられる。精度保証付き多倍長区間演算については、下端・上端方式であり、MPFI とほぼ同等のものである。Affine 演算とは、区間演算とは異なる精度保証付き数値計算の手法である。Affine 演算については、 5.4 節で説明する。

2.4.5　XSC

XSC [6] は、精度保証付き多倍長区間演算ライブラリである。C++ 版と Pascal 版がある。区間演算は、下端・上端方式である。

2.4.6　exflib

exflib [4] は、多倍長演算ライブラリである。C++ ・ FORTRAN から利用できる。また、一般公開されていないが、精度保証付き多倍長区間演算の機能もある。区間演算は、下端・上端方式である。

2.4.7 INTLAB

INTLAB [10] は、 MATLAB 上で精度保証付き区間演算を実現するためのライブラリである。INTLAB は、主に倍精度演算用のライブラリであるが、精度保証付き多倍長区間クラスも実装されている。ただし、この多倍長区間クラスは、倍精度の精度保証付き数値計算のためにライブラリ内部で利用するために作成されたものであり、機能は限定的である。この多倍長区間クラスは、中心値・半径方式で実装されている。以下にその仕様を紹介する。

INTLAB における精度保証付き多倍長区間演算

INTLAB の多倍長区間クラスは、以下のような構造を持つ。

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

表 2.5: INTLAB の多倍長区間クラスのメンバ変数

変数	意味	値
x.sign	中心値の符号	± 1
x.mantissa	中心値の仮数部	配列、各要素の値は 0 以上 $2^{23}-1$ 以下の整数
x.exponent	中心値の指数部	整数
x.error.mant	半径の仮数部	正の実数、 IEEE 754 の倍精度実数を使用
x.error.exp	半径の指数部	整数

多倍長演算の精度は、配列の要素数 n によって決定される。n は、 4 以上の整数である。

ここで注目すべきことは、区間の中心値が多倍長の浮動小数点数であるのに対して、半径が IEEE 754 の倍精度数と同等の精度しか持たないことである。なぜ半径が低精度でも良いのかについては、 4.1.3 節で後述する。

このクラスで最も記憶容量が大きい部分は、配列 x.mantissa である。配列 x.mantissa の各要素には、 IEEE 754 の倍精度実数が用いられている。IEEE 754 の倍精度実数一つの記憶容量は 64 ビットであるが、 x.mantissa では そのうち 23 ビットしか使用しておらず、無駄の大きい実装である。

また、このクラスでは四則演算のみが実装されているが、乗算・除算には簡単だが誤差の大きいアルゴリズムが用いられている。

第3章　ライブラリ LILIB の仕様

3.1　概要

LILIB(Long Interval LIBrary) は、以下のような特徴を持ったライブラリである。

- C++ で多倍長精度の精度保証付き区間演算を行うためのライブラリである。
- C++11 準拠の C++ コンパイラでコンパイル可能である。
- 以下の四つの多倍長数クラスを提供する。
  - 多倍長実数クラス `LongFloat`
  - 多倍長実数行列クラス `LongMatrix`
  - 精度保証付き多倍長区間クラス `LongInterval`
  - 精度保証付き多倍長区間行列クラス `LongIntervalMatrix`
- 上記の多倍長数クラスは、組み込み型の `double` などとほぼ同じように、四則演算・比較演算・平方根の計算が可能である。
- 区間値を、中心値と半径の組み合わせで保持する。
- 多倍長演算の精度は、 10 進数の桁数で任意に設定できる。

LILIB は、 `https://osdn.jp/projects/lilib/` で公開している。LGPL [17] ライセンスのオープンソースソフトウェアである。

3.2　使用方法

LILIB のインストールは、以下の様に行う。

1. `https://osdn.jp/projects/lilib/` にアクセスし、最新の zip ファイルをダウンロードする。
2. 適当なディレクトリ内で、 zip ファイルを展開する。
3. ディレクトリ内で、 make コマンドを実行する。
4. ライブラリファイル `libli.a` が生成される。また、 `sample.cpp` がコンパイルされ、実行ファイル `a.exe` も生成される。

sample.cpp の内容については次節で記述する。
一般に、ユーザの作成したプログラム src.cpp をコンパイルするには、

- src.cpp
- lilib.h
- libli.a

の三つのファイルが必要である。lilib.h は、 zip ファイルに含まれている。
src.cpp 内では、まず lilib.h を include する。次に、多倍長変数を宣言する前に、関数

```
void lilib::setPrecision(int precision)
```

を一度だけ呼ぶ。precision は、使用したい多倍長演算の精度 (10 進数の桁数) である。これで、 precision 桁以上の精度の多倍長演算が可能になる。
LILIB での多倍長数は 32 ビット単位で構成されているので、実際に扱える桁数は、precision より多くなることがある。例えば、 precision = 100 とした場合、 105 桁の多倍長演算が可能になる。
実際に扱える桁数は、関数

```
int lilib::getPrecision()
```

によって取得できる。
例えば GCC の場合、コンパイルは以下のように行う。

```
g++ src.cpp libli.a
```

3.3　サンプルプログラム

以下に示すサンプルプログラムと実行結果から、次のようなことが分かる。

- 多倍長演算の精度を 10 進数 100 桁と設定すると、 105 桁の精度が確保される。
- 1/3 を精度保証付きで計算すると、中心値が $0.333\cdots$ で半径の小さな区間が得られる。
- $(1/3) \times 3$ を精度保証付きで計算すると、中心値が 1 程度で半径の小さな区間が得られる。ただし、中心値が厳密に 1 かどうかは、別途比較演算などで確かめる必要がある。
- 関数 trans で、行列の転置行列が得られる。
- 行列の乗算は、 c = a * b; と簡単に記述できる。

ソースコード

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

using namespace std;

int main(){
  lilib::setPrecision(100);
  cout << "Long precision is " << lilib::getPrecision() << "." << endl;
  cout << endl;

  LongInterval x;

  x = 1;
  cout << "x = " << x << endl;

  x /= 3;
  cout << "x = " << x << endl;

  x *= 3;
  cout << "x = " << x << endl;
  cout << endl;


  int m = 3, n = 2, i, j;
  LongIntervalMatrix a(m, n), b, c;

  for(i = 0; i < m; i++){
    for(j = 0; j < n; j++){
      a[i][j] = n * i + j;
    }
  }

  b = trans(a);
  c = a * b;

  cout << "a =" << endl << a << endl;
  cout << "b =" << endl << b << endl;
  cout << "c =" << endl << c << endl;

  return 0;
```

```
}
```

実行結果

```
Long precision is 105.

x = < 1.000000, 0.000000>
x = < 3.333333e-1, 2.537942e-116>
x = < 1.000000, 7.613826e-116>

a =
< 0.000000, 0.000000>   < 1.000000, 0.000000>
< 2.000000, 0.000000>   < 3.000000, 0.000000>
< 4.000000, 0.000000>   < 5.000000, 0.000000>

b =
< 0.000000, 0.000000>   < 2.000000, 0.000000>   < 4.000000, 0.000000>
< 1.000000, 0.000000>   < 3.000000, 0.000000>   < 5.000000, 0.000000>

c =
< 1.000000, 0.000000>   < 3.000000, 0.000000>   < 5.000000, 0.000000>
< 3.000000, 0.000000>   < 1.300000e1, 0.000000> < 2.300000e1, 0.000000>
< 5.000000, 0.000000>   < 2.300000e1, 0.000000> < 4.100000e1, 0.000000>
```

3.4 仕様

3.4.1 名前空間 lilib

多倍長演算の精度を制御する以下の関数は、名前空間 `lilib` に含まれる。名前空間とは、 C++ のソースコード内での変数名・関数名などの衝突を避けるための機能である。例えば、名前空間 `lilib` 内の関数 `setPrecision` は、他の名前空間内の `setPrecision` とは区別される。

- `void setPrecision(int precision)`
  多倍長演算の精度を、 10 進数 `precision` 桁以上に設定する。`LongFloat` などの変数を宣言する前に、必ず一度呼ぶ必要がある。また、二度目に呼ぶとエラーメッセージを表示し、プログラムを終了する。

- `int getPrecision()`
  現在の多倍長演算の精度を、 10 進数の桁数で返す。桁数は、 `setPrecision` で設定したものより多い場合がある。

3.4.2 多倍長実数クラス LongFloat

コンストラクタ

- `LongFloat()`
  値は不定である。
- `LongFloat(int x)`
- `LongFloat(double x)`
- `LongFloat(const LongFloat &x)`
  `x` で初期化する。
- `LongFloat(std::string s)`
  文字列 `s` で表される値で初期化する。

コンストラクタとは、 C++ でクラス変数が宣言されたときに呼ばれる特殊な関数である。例えば、ソースコード中で、

```
LongFloat a;
```

と宣言すると、値が不定な変数 `a` が生成される。
また、 `LongFloat` クラスの変数は、宣言時に `int`, `double`, `LongFloat` の値、文字列で初期化することができる。つまり、

```
LongFloat b(123);
LongFloat c(3.14);
LongFloat d(c);
LongFloat e("123.456e-789");
```

などと宣言すれば、

- `b` の値は 123
- `c` の値は 3.14
- `d` の値は `c` と同じ
- `e` の値は 123.456×10^{-789}

になる。なお、 `c` の値は、「`double` で表された 3.14 の近似値」 であり、数学的な意味での 3.14 とは一致しないことに注意が必要である。同様に、 `e` の値も、文字列から 2 進数表現への変換誤差を含む近似値である。

メンバ関数

- `void setDouble(double x)`
  値を `x` にする。
- `void setString(std::string s)`
  値を文字列 `s` で表される値にする。
- `double getDouble()`
  `double` での近似値を取得する。
- `double getDouble(int round)`
  丸めの方向を指定して `double` での近似値を取得する。`round < 0` なら $-\infty$ への丸め、 `round == 0` なら最近接丸め (偶数)、 `round > 0` なら $+\infty$ への丸めである。
- `std::string getString()`
  値を表す文字列を取得する。
- `std::string getInternalData()`
  内部データを表す文字列を取得する。主にデバッグ用の関数である。

メンバ関数とは、 C++ のクラス変数に付随する関数である。例えば、

```
LongFloat a;
a.setDouble(3.14);
std::cout << a.getDouble() << std::endl;
```

のように使用する。

非メンバ関数

- `LongFloat abs(const LongFloat &x)`
  `x` の絶対値を取得する。
- `pow(const LongFloat &x, int n)`
  `x` の `n` 乗を取得する。
- `sqrt(const LongFloat &x)`
  `x` の平方根を取得する。

非メンバ関数は、クラス変数に付随しない、通常の関数である。ここでは、 `LongFloat` クラスの変数を引数に取るものを紹介している。例えば、

```
LongFloat two, sqrtTwo;
two = 2;
sqrtTwo = sqrt(two);
```

のように使用する。

3.4.3 多倍長実数行列クラス LongMatrix

行列要素へのアクセス例

4 行 3 列の行列を作る場合、以下のように宣言する。

```
LongMatrix a(4, 3);
```

行列 a の 2 行 1 列目の要素には、以下のようにアクセスできる。行数・列数が 0 から始まっていることに注意が必要である。

```
a[1][0] = 1;
a[1][0] /= 3;
std::cout << a[1][0] << std::endl;
```

コンストラクタ

- `LongMatrix()`
  1 行 1 列の行列を生成する。値は不定である。
- `LongMatrix(int rows, int columns)`
  `rows` 行 `columns` 列の行列を生成する。値は不定である。
- `LongMatrix(const LongMatrix &x)`
  `x` で初期化する。

メンバ関数

- `void resize(int rows, int columns)`
  サイズを `rows` 行 `columns` 列 にする。値は不定になる。主にライブラリ内部で使用される関数である。
- `int rows()`
  行数を取得する。
- `int columns()`
  列数を取得する。
- `std::string getString()`
  値を表す文字列を取得する。

非メンバ関数

- `LongMatrix abs(const LongMatrix &a)`
  各要素が a の各要素の絶対値となる行列を取得する。
- `LongMatrix sqrt(const LongMatrix &a)`
  各要素が a の各要素の平方根となる行列を取得する。
- `LongMatrix trans(const LongMatrix &a)`
  a の転置行列を取得する。
- `LongMatrix zeros(int rows, int columns)`
  rows 行 columns 列の全要素が 0 の行列を取得する。
- `LongMatrix ones(int rows, int columns)`
  rows 行 columns 列の全要素が 1 の行列を取得する。
- `LongMatrix eye(int size)`
  size 行 size 列 の単位行列を取得する。
- `void qr(LongMatrix &q, LongMatrix &r, const LongMatrix &a)`
  a を QR 分解し、結果を q，r に代入する。簡易的な実装であり、精度は低い。この関数を用いれば、常微分方程式の精度保証法である Lohner 法の実装が可能である [18] 。

3.4.4 精度保証付き多倍長区間クラス LongInterval

コンストラクタ

- `LongInterval()`
  値は不定である。
- `LongInterval(int x)`
- `LongInterval(double x)`
- `LongInterval(const LongFloat &x)`
- `LongInterval(const LongInterval &x)`
  x で初期化する。
- `LongInterval(std::string s)`
  文字列 s で表される値を含む区間で初期化する。
- `LongInterval(int mid, int rad)`
- `LongInterval(int mid, const LongFloat &rad)`

- LongInterval(const LongFloat &mid, int rad)
- LongInterval(const LongFloat &mid, const LongFloat &rad)
  中心値を mid 、半径を rad で初期化する。

メンバ関数

- void setDouble(double x)
  中心値を x 、半径を 0 にする。
- void setString(std::string s)
  文字列 s で表される値を含む区間にする。
- void setMidRad(int mid, int rad)
- void setMidRad(int mid, const LongFloat &rad)
- void setMidRad(const LongFloat &mid, int rad)
- void setMidRad(const LongFloat &mid, const LongFloat &rad)
  中心値を mid 、半径を rad にする。
- double getDouble()
  中心値の double での近似値を取得する。
- std::string getMidRad()
  中心値と半径を表す文字列を取得する。
- std::string getInfSup()
  下端と上端を表す文字列を取得する。
- std::string getInternalData()
  内部データを表す文字列を取得する。主にデバッグ用の関数である。
- LongFloat mid()
  中心値を取得する。
- LongFloat rad()
  半径を取得する。
- LongFloat diam()
  直径を取得する。
- LongFloat inf()
  下端を取得する。
- LongFloat sup()
  上端を取得する。

- `LongFloat mig()`
  最小絶対値を取得する。

- `LongFloat mag()`
  最大絶対値を取得する。

- `int contains(int x)`

- `int contains(const LongFloat &x)`

- `int contains(const LongInterval &x)`
  区間が x を含むか判定する。含むなら 1 、含まないなら 0 、不明なら -1 を返す。x が区間の境界に重なっている場合、含まないとする。「不明」という判定結果がある理由は、 3.4.7 節で述べる。

- `int containsEqual(int x)`

- `int containsEqual(const LongFloat &x)`

- `int containsEqual(const LongInterval &x)`
  区間が x を含むか判定する。含むなら 1 、含まないなら 0 、不明なら -1 を返す。x が区間の境界に重なっている場合、含むとする。

非メンバ関数

- `LongInterval pow(const LongInterval &x, int n)`
  x の n 乗を取得する。

- `LongInterval sqrt(const LongInterval &x)`
  x の平方根を取得する。

3.4.5 精度保証付き多倍長区間行列クラス LongIntervalMatrix

コンストラクタ

- `LongIntervalMatrix()`
  1 行 1 列の行列を生成する。値は不定である。

- `LongIntervalMatrix(int rows, int columns)`
  rows 行 columns 列の行列を生成する。値は不定である。

- `LongIntervalMatrix(LongMatrix x)`

- `LongIntervalMatrix(LongIntervalMatrix x)`
  x で初期化する。

メンバ関数

- `void resize(int rows, int columns)`
  サイズを `rows` 行 `columns` 列 にする。値は不定になる。主にライブラリ内部で使用される関数である。

- `int rows()`
  行数を取得する。

- `int columns()`
  列数を取得する。

- `std::string getMidRad()`
  中心値と半径を表す文字列を取得する。

- `std::string getInfSup()`
  下端と上端を表す文字列を取得する。

- `LongMatrix mid()`
  中心値を取得する。

- `LongMatrix rad()`
  半径を取得する。

- `LongMatrix diam()`
  直径を取得する。

- `LongMatrix inf()`
  下端を取得する。

- `LongMatrix sup()`
  上端を取得する。

- `LongMatrix mig()`
  最小絶対値を取得する。

- `LongMatrix mag()`
  最大絶対値を取得する。

非メンバ関数

- `LongIntervalMatrix sqrt(const LongIntervalMatrix &a)`
  各要素が `a` の各要素の平方根となる行列を取得する。

- `LongIntervalMatrix trans(const LongIntervalMatrix &a)`
  `a` の転置行列を取得する。

3.4.6　スカラー・行列間の四則演算

数学的には、スカラー・行列間の四則演算は、乗算以外定義されていない。しかし、LILIB ではコーディングの利便性のため、以下の演算を実装している。

- スカラー + 行列
- スカラー - 行列
- スカラー * 行列
- スカラー / 行列
- 行列 + スカラー
- 行列 - スカラー
- 行列 * スカラー
- 行列 / スカラー
- 行列 += スカラー
- 行列 -= スカラー
- 行列 *= スカラー
- 行列 /= スカラー

これらの演算では、行列の各要素に対してスカラー演算が行われる。

サンプルプログラム

例として、以下の計算を行うソースコードと、その実行結果を示す。

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} + 5 \;\rightarrow\; \begin{pmatrix} 1+5 & 2+5 \\ 3+5 & 4+5 \end{pmatrix}$$

$$1/\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \;\rightarrow\; \begin{pmatrix} 1/1 & 1/2 \\ 1/3 & 1/4 \end{pmatrix}$$

ソースコード

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

using namespace std;

int main(){
  lilib::setPrecision(100);

  LongMatrix a(2, 2);

  a[0][0] = 1;
  a[0][1] = 2;
  a[1][0] = 3;
  a[1][1] = 4;

  cout << "a =" << endl << a << endl;
  cout << "a + 5 =" << endl << a + 5 << endl;
  cout << "1 / a =" << endl << 1 / a << endl;

  return 0;
}
```

実行結果

```
a =
 1.000000        2.000000
 3.000000        4.000000

a + 5 =
 6.000000        7.000000
 8.000000        9.000000

1 / a =
 1.000000        5.000000e-1
 3.333333e-1     2.500000e-1
```

3.4.7 区間の比較演算

LongInterval クラスの変数は、比較演算子を用いて int, LongFloat, LongInterval クラスと比較できる。区間の比較には、点の比較とは異なる性質があるので、注意が必要

である。例えば

$$A < B$$

は、区間 A の全ての要素が、区間 B のどの要素よりも小さいことを意味する。これはすなわち、

$$\sup(A) < \inf(B)$$

と同値である。

このような区間の比較においては、例えば [1, 3] と [2, 4] のように、「大きくも小さくも等しくもない」という関係が存在する。

また、 `LongInterval` では区間を中心値・半径方式で保持しているが、この方式には区間の下端・上端が厳密に求められないことがあるという欠点がある。例えば、区間の下端を求める式は

$$\text{下端} = \text{中心値} - \text{半径}$$

であるが、この減算でも丸め誤差が発生しうる。そこで、

$$\text{下端} = \bigtriangledown(\text{中心値} - \text{半径})$$

として、「下端の取りうる最小値」を求める必要がある。この様に求めた「下端・上端」は真値ではないので、二つの区間の端が非常に近い場合、厳密な大小比較が行えないことがある。

そこで、 `LongInterval` の大小比較では、真・偽・不明の三通りの結果を返す。このために、 `LongInterval` の大小比較では、戻り値の型として、 `bool` ではなく `int` を用いる。この `int` 値の内容は以下のようなものである。

- 真のとき、 1
- 偽のとき、 0
- 不明のとき、 -1

なお、「二つの区間が等しいか」は厳密に調べることができるので、

- `LongInterval == LongInterval`
- `LongInterval != LongInterval`

の二つの演算は、従来通り bool 型の値を返す。

第4章　ライブラリ LILIB の実装

4.1　実装方針

4.2 節では、 LILIB の具体的なアルゴリズムと実装について述べるが、本節ではその前に、なぜそのアルゴリズム・実装を採用したのか、その理由を述べる。

4.1.1　limb のビット数

GMP などの多倍長演算ライブラリでは、 limb のビット数が 64 であるものが多い。一方、 LILIB の limb のビット数は 32 とした。これは、 LILIB がアセンブラを使わず、C++ だけで実装されていることに起因している。limb を 64 ビットとすると、乗算において

$$128\text{ビット} = 64\text{ビット} \times 64\text{ビット}$$

という計算が必要になる。しかし、標準的な C++ 環境では、 128 ビットの整数を扱うことができない。環境依存の実装を行えば 128 ビットの整数を扱うこともできるが、移植性を重視して、 LILIB の limb は 32 ビットとした。

4.1.2　中心値・半径方式のメリット

LILIB では、区間を中心値と半径の組み合わせで保持する。中心値・半径方式の第一のメリットは、条件次第で記憶容量を節約できることである。例えば、下端・上端方式で以下のように表される区間を考える。簡単のため、値は 10 進数で格納されているとする。

$$X = [1.234567889 \times 10^{10},\ 1.234567891 \times 10^{10}]$$

区間 X の表現には、下端・上端それぞれに 10 桁の仮数を要する。一方、同じ区間を中心値・半径方式で表した場合、以下のようになる。

$$X = \left\langle 1.234567890 \times 10^{10},\ 1 \times 10^{1} \right\rangle$$

この場合、中心値の仮数は 10 桁、半径の仮数は 1 桁となるが、下端・上端方式の場合と比べて情報は失われていない。このように、区間の相対誤差が比較的小さい場合には、半径の精度を小さくして記憶容量を節約できる。

半径を低精度で保持する場合、第二のメリットとして、計算速度も向上する。いわゆる「筆算の桁数」が減るからである。

以上の理由から、 LILIB の区間クラス `LongInterval` では、中心値を多倍長数 `LongFloat` で保持するのに対して、半径は仮数部 32 ビットの浮動小数点数 `Radius` で保持する。

半径を低精度で保持するこの方法は、扱う区間の相対誤差が十分小さいことを前提としている。この方法で相対誤差が大きい区間を扱うと、誤差が非常に大きくなってしまい、多倍長演算の意味が失われてしまう。しかし、相対誤差が大きい区間は、そもそも多倍長演算に不向きであり、想定することに意味がない。以下にその理由を説明する。

4.1.3 半径が低精度でも良い理由

精度保証付き区間演算の目的は、以下の二つの誤差を上から評価して、その影響を区間半径として取り込むことである。

(i) 計算過程で現れる打ち切り誤差

(ii) 浮動小数点数演算で生じる丸め誤差

区間演算で発生する誤差には、上記二つの誤差の他、

(iii) 機械区間演算に由来する区間拡大

がある。これは特に、区間変数の関数値を区間で包み込む際に発生する。関数値の計算方法によってある程度改善はできるものの、完全に除去することは一般には難しい。

一方で、多倍長演算の効用は、 (ii) の丸め誤差の影響を小さくすることにあると言える。注意すべき点は、多倍長演算の利用で (i) の打ち切り誤差や (iii) の区間拡大を減少させることは原理的にできない、ということである。以下では、 (iii) について説明する。

区間演算に由来する区間拡大は、計算の内容と計算対象の区間の相対誤差に依存する。以下に例を示す。

区間

$$X = \langle 1,\ r \rangle$$

に対して関数

$$f(X) = X^3 - 6X^2 + 8X$$

を計算することを考える。r を変化させ、真の計算結果と機械区間演算の結果を比較すると、表 4.1 のようになる。

表 4.1: 機械区間演算で発生する区間拡大

区間半径 r	真の計算結果 $f(X)$	機械区間演算の結果 $[f(X)]$
0	$\langle 3.0000,\ 0.0000\rangle$	$\langle 3.0000,\ 0.0000\rangle$
10^{-10}	$\langle 3.0000,\ 1.0000\times10^{-10}\rangle$	$\langle 3.0000,\ 7.0000\times10^{-10}\rangle$
10^{-9}	$\langle 3.0000,\ 1.0000\times10^{-9}\rangle$	$\langle 3.0000,\ 7.0000\times10^{-9}\rangle$
10^{-8}	$\langle 3.0000,\ 1.0000\times10^{-8}\rangle$	$\langle 3.0000,\ 7.0000\times10^{-8}\rangle$
10^{-7}	$\langle 3.0000,\ 1.0000\times10^{-7}\rangle$	$\langle 3.0000,\ 7.0000\times10^{-7}\rangle$
10^{-6}	$\langle 3.0000,\ 1.0000\times10^{-6}\rangle$	$\langle 3.0000,\ 7.0000\times10^{-6}\rangle$
10^{-5}	$\langle 3.0000,\ 1.0000\times10^{-5}\rangle$	$\langle 3.0000,\ 7.0000\times10^{-5}\rangle$
10^{-4}	$\langle 3.0000,\ 1.0000\times10^{-4}\rangle$	$\langle 3.0000,\ 7.0000\times10^{-4}\rangle$
10^{-3}	$\langle 3.0000,\ 1.0000\times10^{-3}\rangle$	$\langle 3.0000,\ 7.0000\times10^{-3}\rangle$
10^{-2}	$\langle 2.9997,\ 9.9990\times10^{-3}\rangle$	$\langle 3.0005,\ 7.0001\times10^{-2}\rangle$
10^{-1}	$\langle 2.9700,\ 9.9000\times10^{-2}\rangle$	$\langle 3.0500,\ 7.0100\times10^{-1}\rangle$

この例では、区間演算結果の半径は真の半径の約 7 倍となり、この比率は r を変えてもほぼ一定であることが分かる。この現象は区間演算の性質によるものであって、計算過程を多倍長で行なっても比率が変わることはない。すなわち、

- 多倍長演算は、区間拡大を改善しない。
- 区間半径の精度は、区間の相対誤差によって支配される。

したがって、区間の相対誤差が大きい場合には、高精度な半径の表現は冗長なものとなり、多くの場合その意味を失ってしまう。

上記の理由により、 LILIB では区間半径を低精度で保持することにした。では、具体的に何ビット保持するべきだろうか。

極端な方法としては、半径を 1 ビット保持するということも考えられる。これはつまり、中心値の何ビット目に半径があるかという情報のみを保持する方法である。しかし、この方法だと、演算で丸めが発生する度に半径が 1 ビット繰り上がることになる。これでは、半径が急速に拡大してしまう。半径のビット数は、中心値より少なくて良いが、やはりある程度は必要である。

そこで、 LILIB では区間半径を 32 ビット保持することにした。`LongFloat` の limb が 32 ビットであり、それと同サイズにするのが、実装上扱いやすいからである。

4.1.4 LongFloat の演算アルゴリズム

我々は、 `LongFloat` の演算アルゴリズムとして、

- 乗算には、筆算
- 除算には、$1/x$ を求める Newton 法

- 平方根の計算には、$1/\sqrt{x}$ を求める Newton 法

を採用した。

しかし、これらのアルゴリズムは、実装が容易だが低速なものである。2.2.5 節で述べたように、例えば乗算なら Karatsuba 法や高速 Fourier 変換を用いる方法など、より高速なアルゴリズムが存在する。本来なら、より高速なアルゴリズムを採用すべきだが、今回それは見送った。LongFloat の演算の高速化は、優先順位が低かったからである。

そもそも本研究の目的は、「精度保証付き多倍長区間演算において、中心値・半径方式が下端・上端方式より優れていることを示す」ことである。つまり、重要なのは、LongFloat と LongInterval の演算速度の比なのである。

下端・上端方式の区間演算では、点の演算とほぼ同じ計算を 2 回行う。よって、区間の計算時間は、点の計算時間の約 2 倍となる。LILIB の目的は、この計算時間の比を 2 未満に抑えることである。そのために重要なのは、 LongInterval のアルゴリズムを工夫することである。LongFloat の演算を高速化しても、それをベースにした LongInterval も同程度高速化され、速度比は変わらない。以上のような理由で、 LongFloat の演算アルゴリズムは、簡易的なもので済ませている。

4.1.5 LongFloat の演算精度

2.2.4 節で述べたように、浮動小数点数演算を実装する場合、 correct rounding が演算精度の一つの目安になる。しかし、 LILIB の LongFloat の演算は correct rounding ではない。これは、演算精度より演算速度を優先したためである。

IEEE 754 を始め、多くの実装では、浮動小数点数 x は以下のような形で表現される。

$$x = f \times 2^e$$

一方、 LongFloat の実装は、以下の形である。

$$x = f \times 2^{32e}$$

ここで、 e は整数である。

これは、通常の実装では小数点が 1 ビット単位で動くのに対して、 LongFloat では小数点が 32 ビット単位でしか動かないことを意味している。この構造だと、 correct rounding は実現できないが、シフト操作の回数を減らすことができ、演算速度は速くなる。

次に、 LILIB では correct rounding が必要ないことを説明する。そもそも、 correct rounding とは、精度保証の一種である。ある演算を最近点への correct rounding で行った場合、その演算結果の誤差は 0.5 ulp 以下であることが保証される。同様に、下・上への correct rounding では、演算結果の相対誤差は 1 ulp 未満である。ただし、 correct rounding の精度保証は、演算一回分についてしか有効ではない。

例えば、最近点への correct rounding で、以下の計算を行う。

$$\begin{aligned} c &= a + b \\ e &= cd \end{aligned}$$

c の計算値を $\tilde{c}$ とすると、その相対誤差は、以下のように包囲できる。

$$|\tilde{c} - c| \leq 0.5\,\mathtt{ulp}$$

しかし、 $\tilde{c}$ を用いて計算した $\tilde{e}$ の相対誤差については、特に保証はない。

一方、精度保証付き区間演算を行った場合、計算の各段階で発生した誤差は全て区間半径に取り込まれ、評価される。

つまり、 correct rounding は、精度保証付き区間演算に劣る簡易的な精度保証であり、`LongInterval` が利用できる LILIB においては、演算速度を落としてまで実現する必要はないというのが、我々の考えである。

4.2 実装と演算原理

4.2.1 倍精度演算の丸め方向制御

LILIB の多倍長演算は、基本的に整数演算の組み合わせで行う。しかし、除算と平方根の計算の一部では、倍精度演算を用いる。この倍精度演算の精度保証のために、 IEEE 754 で定義されている丸め方向の制御を用いる。C++11 に準拠したコンパイラであれば、以下の方法で丸め方向の制御が行える。

```
#include <cfenv>

fesetround(FE_TONEAREST);        // 最近接丸め (偶数)
fesetround(FE_DOWNWARD);         // -∞への丸め
fesetround(FE_UPWARD);           // +∞への丸め
fesetround(FE_TOWARDZERO);       // 0 方向への丸め
```

また、以下の方法で現在の丸め方向を知ることができる。

```
switch (fegetround()){
  case FE_TONEAREST:
    printf("最近接丸め (偶数)\n");
    break;

  case FE_DOWNWARD:
    printf("-∞への丸め \n");
    break;

  case FE_UPWARD:
    printf("+∞への丸め \n");
    break;

  case FE_TOWARDZERO:
```

```
    printf("0 方向への丸め\n");
    break;
}
```

4.2.2 固定長整数型

LILIB が扱う多倍長数の仮数部は、 32 ビット符号なし整数の配列で表現される。また、計算過程で 33 ビット以上の値を扱う場面では、 64 ビット符号なし整数を用いる。これらの固定長整数型は、 C++11 で以下の名前で定義されている。

- `uint32_t` : 32 ビット符号なし整数
- `uint64_t` : 64 ビット符号なし整数

`int` や `long` などの型は、計算機環境によってビット数が変わることがあり、移植性が低下する。`uint32_t` などを用いれば、移植性を保つことができる。

4.2.3 多倍長実数クラス LongFloat

以下に、多倍長実数を表現するクラス `LongFloat` のクラス構造を示す。`LongFloat` の仮数部は、 `uint32_t` の配列であり、シフト操作などは、 1 ビット単位ではなく 1 limb(32 ビット) 単位で行う。

```
class LongFloat{
  int sign;              // 符号
  uint32_t *limb;        // 仮数部配列へのポインタ
  int exponent;          // 指数部
};
```

配列 `*limb` の要素数は、 `int lilib::limbs` である。`lilib::limbs` は、関数 `lilib::setPrecision` によって設定されるグローバル変数である。以降、 `lilib::limbs` については、単に `limbs` と表記する。

`double` から `LongFloat` へ誤差なしで変換を行いたいので、 `LongFloat` は `double` より高精度である必要がある。そのため、 `limbs` の最小値は 3 とする。

`LongFloat x` は、以下の値を持つ多倍長実数である。

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

$x = 0$ のとき、 `x.limb` が指す配列の全要素と `x.exponent` は共に 0 である。

$x \neq 0$ のとき、 `x.limb[0]` は非ゼロになるよう正規化される。

`LongFloat` クラスは、以下の演算をサポートしている。

加減算

加減算は、対象の符号や絶対値の大小によって演算の内容が変わるが、以下の二つの演算に集約される。

$$a + b \quad (a \geq b \geq 0)$$
$$a - b \quad (a \geq b \geq 0)$$

LongFloat = LongFloat + LongFloat

```
LongFloat a, b, c;       // a >= b > 0
```

のとき、

$$\mathtt{c} \leq \mathtt{a} + \mathtt{b}$$

となる最大の c を計算する。

この演算では、以下のような場合分けを行う。

1. a.exponent − limbs ≥ b.exponent のとき
   これは、 a と b の仮数部が「重なっていない」状態である。この場合、 c に a を代入して、 b は切り捨てる。

2. a.exponent − limbs < b.exponent のとき
   これは、 a と b の仮数部が「重なっている」状態である。この場合、 *a.limb と *b.limb の桁を合わせて筆算を行う。計算過程では uint64_t を用い、繰り上がり処理も適切に行う。*a.limb と重なっていない *b.limb の下位 limb は、切り捨てる。

LongFloat = LongFloat - LongFloat

```
LongFloat a, b, c;       // a >= b > 0
```

のとき、

$$\mathtt{c} \geq \mathtt{a} - \mathtt{b}$$

となる最小の c を計算する。

この演算では、以下のような場合分けを行う。

1. a.exponent − limbs ≥ b.exponent のとき
   これは、 a と b の仮数部が「重なっていない」状態である。この場合、 c に a を代入して、 b は切り捨てる。

2. a.exponent − limbs < b.exponent のとき
   これは、 a と b の仮数部が「重なっている」状態である。この場合、 *a.limb と *b.limb の桁を合わせて筆算を行う。計算過程では uint64_t を用い、繰り下がり処理も適切に行う。*a.limb と重なっていない *b.limb の下位 limb は、切り捨てる。

乗算

ここでは、正の実数同士の乗算についてのみ説明する。0 を含む場合は演算結果は 0 であり、負の値を含む場合は、符号を適切に処理すれば良い。

```
LongFloat a, b, c;      // a > 0, b > 0
```

のとき、

$$\texttt{c} \leq \texttt{ab}$$

となる最大の c を計算する。

*a.limb, *b.limb は要素数 limbs の uint32_t の配列なので、この乗算結果を格納するには、要素数 2 * limbs の uint32_t の配列が必要である。そこで、そのような配列を用意し、その配列に一時的に乗算結果を格納する。

この乗算は、 limb を単位とする筆算で行う。32 ビット同士の乗算結果は 64 ビットとなるので、 uint64_t 上で乗算を行い、上位 32 ビットを次の limb へ繰り上げる。

最終的に、乗算結果の上位 limbs limb を取り出し、 *c.limb に格納する。乗算結果の下位の部分は、切り捨てる。

除算

ここでは、正の実数同士の除算についてのみ説明する。分子が 0 の場合は演算結果は 0 、分母が 0 の場合は演算不能である。負の値を含む場合は、符号を適切に処理すれば良い。

LongFloat = LongFloat / int

```
LongFloat a;    // a > 0
int b;          // b > 0
```

のとき、

$$\texttt{c} \leq \texttt{a}/\texttt{b}$$

となる最大の c を計算する。

この除算は、 limb を単位とする筆算で行う。各 limb での計算には、上の limb からの繰り下がりを含めた 64 ビット演算が必要なので、 uint64_t を用いる。最下位 limb で発生した余りは、切り捨てる。

LongFloat = int / LongFloat
LongFloat = LongFloat / LongFloat

分母が LongFloat の場合、筆算の実装は難しい。そこで、以下の Newton 法を用いて分母 x の逆数 $\bar{x}$ の近似値 y を求める。この方法は、 INTLAB の実装を参考にした。

$$y_{k+1} = (2 - xy_k)y_k$$

LongFloat y の値が変化しなくなった時点で、反復計算を終了する。

このとき、反復計算の初期値 y_0 は、ある程度真の $\bar{x}$ に近い必要がある。そこで、 x を一度、 double に変換し、その逆数を求める。その結果を再び LongFloat に変換し、これを y_0 とする。

double で表せる数は、

$$10^{-324} < |x| < 10^{307}$$

程度の範囲に限られるが、 LongFloat はより広い範囲を扱うことができる。そのため、x の値をそのまま倍精度数に変換できるとは限らない。そこで、 x を仮数部と指数部に分け、仮数部のみを double に変換する。指数部については、符号を反転させることで、逆数の近似値が求められる。

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 Newton 法を用いる。

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

LongFloat y の値が変化しなくなった時点で、反復計算を終了する。

ここで、直接 $\sqrt{x}$ を求めず $1/\sqrt{x}$ を求めるのは、このようにすれば Newton 法の計算過程において、計算量の多い多倍長の除算を回避できるからである。

ここでも除算と同様に、反復計算の初期値 y_0 は、ある程度真の y に近い必要がある。そこで、 double の組み込み関数 sqrt を用い、 y_0 を求める。ここでも、 double への変換において、仮数部と指数部に分ける工夫が必要である。

比較演算

LongFloat 同士、または LongFloat と int の間で、以下の比較演算が行える。

==, !=, <, >, <=, >=

int との比較においては、 int 値を変換し、LongFloat 同士の比較を行う。

LongFloat 値の大小関係を調べる手順は、以下のようになる。

1. sign を比較する。符号が異なれば、大小関係は決定する。
2. exponent を比較する。指数部が異なれば、大小関係は決定する。
3. limb[0], limb[1], limb[2], …を順に比較する。仮数部が一箇所でも異なれば、大小関係は決定する。
4. 仮数部が最後まで等しければ、二つの値は等しい。

4.2.4 区間半径クラス Radius

LILIB では、区間半径を専用クラス Radius に格納する。このクラスは LILIB 内部でのみ使われ、利用者が意識する必要はない。以下に Radius のクラス構造を示す。

```
class Radius{
  uint32_t significand; // 仮数部
  int exponent;         // 指数部
};
```

Radius x は、以下の値を持つ非負の実数である。

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

$x = 0$ のとき、 x.significand と x.exponent は共に 0 である。
$x \neq 0$ のとき、 x.significand は最上位ビットが 1 になるよう正規化される。
Radius クラスは、以下の演算をサポートしている。

Radius = Radius * int

```
Radius a, c;
int b;          // b >= 0
```

のとき、

$$\mathtt{c} \geq \mathtt{ab}$$

となる最小の c を計算する。
この演算では、桁あふれを考慮して仮数部を一度 uint64_t work に格納し、計算する。例えば、 work の値が 2 進数で

```
0000010101010101010101010101010101010101010101010101010101010101
```

だったとき、まず、最上位ビットが 1 になるよう正規化する。

```
1010101010101010101010101010101010101010101010101010101010101010
```

ここでは、 5 ビット左にシフトした分、 c.exponent から 5 を減じる。
次に、上位 32 ビットの値を、上への丸めで取り出す。c.significand の値は

```
10101010101010101010101010101011
```

となる。上への丸めを行なっているので、最下位ビットは、 0 ではなく 1 となる。
このように、 Radius に関する演算は、常に上への丸めで行われる。精度保証付き区間演算では、常に取りうる最大の半径を保持する必要があるからである。これが、 LILIB による精度保証の最も低レベルな部分である。

Radius = Radius * Radius

`Radius = Radius * int` とほぼ同じ演算である。

Radius = Radius / int

```
Radius a, c;
int b;          // b > 0
```

のとき、

$$c \geq a/b$$

となる最小の c を計算する。

ここでも乗算と同様に、仮数部を一度 `uint64_t work` に格納し、計算する。除算で余りが発生した場合、切り上げを行う。その後の丸め処理は、乗算と同様である。

Radius = Radius + Radius

```
Radius a, b, c;          // a >= b
```

のとき、

$$c \geq a + b$$

となる最小の c を計算する。

この演算では、以下のような場合分けを行う。

1. $b = 0$ のとき
   c に a を代入する。

2. $b \neq 0$, a.exponent $\geq$ b.exponent $+ 32$ のとき
   これは、 a と b の仮数部が「重なっていない」状態である。この場合、 c に a を代入し、 c.significand の最下位ビットに 1 を加える。最下位ビットの加算で繰り上がりが発生する場合、これも適切に処理する。

3. $b \neq 0$, a.exponent $<$ b.exponent $+ 32$ のとき
   これは、a と b の仮数部が「重なっている」状態である。この場合、uint64_t を用いて、a.significand と b.significand の桁を合わせて加算を行う。c.significand への丸めは、上への丸めで行う。

LongFloat と Radius の相互変換

LongFloat と Radius は、コンストラクタ

```
LongFloat(const Radius &x)
Radius(const LongFloat &x)
```

を用いて、相互に変換できる。

- LongFloat の精度は Radius より高いので、 Radius から LongFloat への変換では、誤差は発生しない。
- LongFloat から Radius への変換では、上への丸めを行う。

4.2.5 精度保証付き多倍長区間クラス LongInterval

以下に、多倍長区間を表現するクラス LongInterval のクラス構造を示す。

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

LongInterval x は、以下の中心値と半径で表される区間である。

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

LongInterval クラスは、以下の演算をサポートしている。

加減算

LongFloat の加減算と同様に、以下の二つの演算について考える。

$$\begin{aligned} \mathtt{a} + \mathtt{b} \quad & (\mathtt{a.center} \geq \mathtt{b.center} \geq 0) \\ \mathtt{a} - \mathtt{b} \quad & (\mathtt{a.center} \geq \mathtt{b.center} \geq 0) \end{aligned}$$

LongInterval = LongInterval + LongInterval

```
LongInterval a, b, c;   // a.center >= b.center > 0
```

のとき、

$$\mathtt{a} + \mathtt{b} \subseteq \mathtt{c}$$

となる c を計算する。c.radius は、小さい方が望ましい。
この演算のためには、まず

$$\mathtt{a.center} + \mathtt{b.center} \in \mathtt{d}$$

となる LongInterval d を求める必要がある。
d の計算では、以下のような場合分けを行う。

1. a.center.exponent − limbs ≥ b.center.exponent のとき
   これは、 a.center と b.center の仮数部が「重なっていない」状態である。この場合、まず d.center に a.center を代入する。次に、

$$\mathtt{d.radius} \geq \mathtt{b.center}$$

   となるように d.radius を設定する。

2. a.center.exponent − limbs < b.center.exponent のとき
   これは、 a.center と b.center の仮数部が「重なっている」状態である。この場合、 *a.center.limb と *b.center.limb の桁を合わせて筆算を行う。計算過程では uint64_t を用い、繰り上がり処理も適切に行う。*a.center.limb と重なっていない *b.center.limb の下位 limb を bLower とし、

$$\mathtt{d.radius} \geq \mathtt{bLower}$$

   となるように d.radius を設定する。

d が求められたら、以下のように c を計算する。

```
c.center = d.center;
c.radius = a.radius + b.radius + d.radius;
```

LongInterval = LongInterval - LongInterval

```
LongInterval a, b, c;   // a.center >= b.center > 0
```

のとき、

$$\mathtt{a} - \mathtt{b} \subseteq \mathtt{c}$$

となる c を計算する。c.radius は、小さい方が望ましい。
この演算のためには、まず

$$\mathtt{a.center} - \mathtt{b.center} \in \mathtt{d}$$

となる LongInterval d を求める必要がある。
d の計算では、以下のような場合分けを行う。

1. a.center.exponent − limbs ≥ b.center.exponent のとき
   これは、 a.center と b.center の仮数部が「重なっていない」状態である。この場合、まず d.center に a.center を代入する。次に、

$$\mathtt{d.radius} \geq \mathtt{b.center}$$

   となるように d.radius を設定する。

2. a.center.exponent − limbs < b.center.exponent のとき
   これは、 a.center と b.center の仮数部が「重なっている」状態である。この場合、 *a.center.limb と *b.center.limb の桁を合わせて筆算を行う。計算過程では uint64_t を用い、繰り下がり処理も適切に行う。*a.center.limb と重なっていない *b.center.limb の下位 limb を bLower とし、

$$\texttt{d.radius} \geq \texttt{bLower}$$

   となるように d.radius を設定する。

d が求められたら、以下のように c を計算する。

```
c.center = d.center;
c.radius = a.radius + b.radius + d.radius;
```

mid()

x.mid() は、 x.center の値をそのまま返す。

rad()

x.rad() は、 x.radius を LongFloat に変換して返す。

diam()

x.diam() は、 x.rad() の 2 倍の値を返す。これは、 x.radius の exponent に 1 を加えた値を、 LongFloat に変換することで実現している。

inf()

x.inf() は、区間 x の下端を返す。ただし、 3.4.7 節で述べたように、中心値・半径形式の区間演算では、常に正確な下端を求められるとは限らない。そこで、 x.inf() は、区間 x の下端が取りうる最小の値を返す。具体的には、以下のような処理を行う。

1. 「中心値 - 半径」を精度保証付きで計算し、結果を LongInterval y とする。
2. y の半径が 0 なら、 y の中心値を返す。
3. y の中心値が 0 なら、 -y.rad() を返す。
4. y の中心値と半径の仮数部が「重なっている」なら、 y.inf() を呼び、その結果を返す。y.inf() 内で y.inf() を呼ぶことになるが、これ以上、 y.inf() が再帰的に呼ばれることはない。
5. y の中心値と半径の仮数部が「重なっていない」なら、
   - y の中心値が正なら、 y の中心値の最下位ビットから 1 減じた値を返す。
   - y の中心値が負なら、 y の中心値の最下位ビットに 1 加えた値を返す。

sup()

`x.sup()` は、区間 `x` の上端を返す。アルゴリズムは `inf()` と同様である。

mig()

`x.mig()` は、区間 `x` の最小絶対値を返す。すなわち、 `x` が 0 を含む場合は 0 を、それ以外の場合は、 `abs(x.inf())` と `abs(x.sup())` の小さい方を返す。

mag()

`x.mag()` は、区間 `x` の最大絶対値を返す。すなわち、 `abs(x.inf())` と `abs(x.sup())` の大きい方を返す。

乗算

$$c_a \geq 0, \quad c_b \geq 0$$

のとき、区間 $\langle c_a,\ r_a \rangle$ と $\langle c_b,\ r_b \rangle$ の乗算は、丸め誤差を考慮しなければ、以下のようになる。

1. $c_a \geq r_a,\ c_b \geq r_b$ のとき、

$$\langle c_a,\ r_a \rangle \cdot \langle c_b,\ r_b \rangle \ = \ \langle c_a c_b + r_a r_b,\ c_a r_b + c_b r_a \rangle \tag{4.1}$$

2. 1. を満たさず、 $c_a r_b \geq c_b r_a$ のとき、

$$\langle c_a,\ r_a \rangle \cdot \langle c_b,\ r_b \rangle \ = \ \langle c_a c_b + c_b r_a,\ c_a r_b + r_a r_b \rangle \tag{4.2}$$

3. 1. も 2. も満たさないとき、

$$\langle c_a,\ r_a \rangle \cdot \langle c_b,\ r_b \rangle \ = \ \langle c_a c_b + c_a r_b,\ c_b r_a + r_a r_b \rangle \tag{4.3}$$

c_a, c_b が負の場合は、符号を適切に処理すれば良い。

これらの式を LILIB で実装する場合、 $c_a c_b$ などの乗算でも丸め誤差が発生することに注意が必要である。

この演算のためには、以下の演算が必要である。

```
LongFloat x, y;          // x >= 0, y >= 0
LongInterval z;
```

のとき、

$$\mathtt{xy} \in \mathtt{z}$$

となる z を計算する。z.radius は、小さい方が望ましい。

ここでは、 LongFloat の乗算と同様に、要素数 2 * limbs の uint32_t の配列を用意し、その配列に xy の結果を格納する。この上位 limbs limb を取り出し、*z.center.limb に格納する。乗算結果の下位の部分を xyLower とし、

$$\mathtt{z.radius} \geq \mathtt{xyLower}$$

となるように z.radius を設定する。

LILIB での式 (4.1) の実装は、以下のようになる。

1. 上記の方法で、 a.center × b.center ∈ d となる LongInterval d を求める。
2. e = a.radius × b.radius となる LongFloat e を求める。
3. LongInterval c = d + e とする。
4. c.radius += Radius(a.center) * b.radius
   + Radius(b.center) * a.radius; とする。

条件 1. を満たさない場合、 $c_a r_b$ と $c_b r_a$ の大小を比較する必要がある。比較のためには $c_a r_b, c_b r_a$ を厳密に計算する必要がある。LongFloat * Radius の結果を誤差なしに保持するには、小数点の移動を考慮すると、 limbs + 2 limb が必要になる。そこで、以下の演算は、一時的に limbs を 2 増やした状態で行う。このために、 limbs を一時的に増減させる機能も用意した。ただし、この機能はライブラリ内部でのみ使えるものである。使い方が複雑なため、一般利用者には開放していない。

LILIB での式 (4.2) の実装は、以下のようになる。

1. a.center × b.center ∈ c となる LongInterval c を求める。
2. c.radius += a.radius * b.radius; とする。
3. LongFloat e = a.center * LongFloat(b.radius); とする。
4. LongFloat f = b.center * LongFloat(a.radius); とする。
5. e ≥ f なら、c += f; c.radius += Radius(e); とする。
6. e < f なら、c += e; c.radius += Radius(f); とする。

式 (4.3) の実装も、同様に行う。

除算

以下の区間の逆数区間を求めることを考える。

$$\langle c,\ r\rangle \quad (c > r \geq 0)$$

区間 $\langle c,\ r\rangle$ の逆数区間は、

$$\begin{aligned}
\frac{1}{\langle c,\ r\rangle} &= \frac{1}{[c-r,\ c+r]} \\
&= \left[\frac{1}{c+r},\ \frac{1}{c-r}\right] \\
&= \frac{[c-r,\ c+r]}{c^2-r^2} \\
&= \frac{\langle c,\ r\rangle}{c^2-r^2}
\end{aligned}$$

と表せるので、

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

の精度保証ができれば、区間の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、 `LongFloat` の除算を用いて $1/z$ の近似値 $\bar{z}$ を求める。
$z\bar{z}$ の 1 に対する誤差を 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。

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

実際の計算では、 $\varepsilon_1,\ \varepsilon_2$ は具体的には求められないので、区間包囲を行う。
w を `LongInterval` で計算し、

$$w \in W$$

となる区間 W を求めると、

$$\begin{aligned}
z &= \mathtt{mid}(W) \\
\varepsilon_1 &\in W - z \\
z - \varepsilon_1 &\in W
\end{aligned}$$

となる。

ε_2 についても、

$$\varepsilon_2 = z\bar{z} - 1 \in E$$

となる区間 E を LongInterval で計算できる。

以上より、δ は以下の式で区間包囲される。

$$\delta \in \frac{WE - W + z}{zW}$$

よって、$1/w$ は以下の式で区間包囲される。

$$\frac{1}{w} \in \left\langle \bar{z},\, \mathtt{mag}\left(\frac{WE - W + z}{zW}\right)\right\rangle$$

ここで、半径の計算に除算が必要になるが、半径に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。

つまり、$1/w$ を含む区間の半径は、以下の手順で計算できる。

1. $WE - W + z \subseteq A$ を満たす `LongInterval A` を計算する。
2. `LongFloat magA = A.mag()` を求める。
3. `int eA = magA.exponent, magA.exponent = 0` とする。
4. `magA` を上へ丸め、 `double dA` を求める。
5. $zW \subseteq B$ を満たす `LongInterval B` を計算する。
6. `LongFloat infB = B.inf()` を求める。
7. `int eB = infB.exponent, infB.exponent = 0` とする。
8. `infB` を下へ丸め、 `double dB` を求める。
9. `double dR = dA / dB` を、上への丸めで計算する。
10. `dR` を上へ丸め、 `Radius r` を求める。
11. `r.exponent += 32 * (eA - eB)` とする。

最終的に、$1/w$ は以下の式で区間包囲される。

$$\frac{1}{w} \in \langle \bar{z},\, r\rangle$$

LongFloat から double への変換の前に指数部を分けているが、これは LongFloat の指数部のビット数が double のものより大きいからである。

平方根

最初に、点区間

$$\langle c,\ 0\rangle \quad (c \geq 0)$$

の平方根の精度保証について考える。

`LongFloat c` に対して、 $\sqrt{c}$ の近似値 s は、

```
LongFloat s = sqrt(c);
```

によって求められる。

c の s^2 に対する相対誤差を ε とおく。

$$c = s^2(1+\varepsilon) \tag{4.4}$$

よって、

$$\sqrt{c} = s\sqrt{1+\varepsilon} \tag{4.5}$$

ここで、 Maclaurin 展開により、

$$\sqrt{1+\varepsilon} = 1 + \frac{1}{2}\varepsilon - \frac{1}{8}\varepsilon^2 + \frac{1}{16}\varepsilon^3 - \cdots$$

である。この式は $|\varepsilon| < 1$ のとき、単調減少する交代級数なので、

$$\sqrt{1+\varepsilon} \in \left\langle 1 + \frac{1}{2}\varepsilon,\ \frac{1}{4}\varepsilon^2 \right\rangle \tag{4.6}$$

という区間包囲ができる。

よって、

$$\sqrt{c} \in s\left\langle 1 + \frac{1}{2}\varepsilon,\ \frac{1}{4}\varepsilon^2 \right\rangle$$

と区間包囲できる。

これは、無限級数を第 3 項で打ち切るという大まかな精度保証である。しかし、 ε は 1 ulp 程度の大きさなので、 2 乗以上の項はほとんど値に影響を与えないため、これで十分である。

実際の計算では、誤差 ε は具体的には求められないが、

$$\varepsilon = \frac{c}{s^2} - 1 \in E$$

を満たす `LongInterval E` は計算できる。

以上より、 $\sqrt{c}$ は以下の式で区間包囲される。

$$\begin{aligned} \sqrt{c} &\in s(1 + \frac{1}{2}E + \frac{1}{4}E^2) \\ &= \frac{s}{4}\{(E+1)^2 + 3\} \end{aligned} \tag{4.7}$$

区間の平方根の計算には、以下の式を用いる。

$$\begin{aligned}\sqrt{\langle c,\ r\rangle} &\subseteq \left\langle \sqrt{c},\ \sqrt{c}-\sqrt{c-r} \right\rangle \\ &= \left\langle \sqrt{c},\ \frac{r}{\sqrt{c}+\sqrt{c-r}} \right\rangle \end{aligned} \tag{4.8}$$

半径の式を変形するのは、桁落ちを防ぐためである。また、この式は区間の相対誤差が小さいことを前提としており、相対誤差が大きいと誤差が大きくなる欠点がある。

(4.7), (4.8) より、区間の平方根は以下のように区間包囲できる。

$$\sqrt{\langle c,\ r\rangle} \in \frac{s}{4}\{(E+1)^2+3\} + \left\langle 0,\ \frac{r}{\sqrt{c}+\sqrt{c-r}} \right\rangle$$

ここで、半径の計算に平方根が必要になるが、半径に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。ここでも、除算の場合と同様に、倍精度演算の前に仮数部と指数部を分けて処理する必要がある。

比較演算

LongInterval == LongInterval

```
LongInterval a, b;
```

のとき、 `a == b` は、 `a.center == b.center` かつ `a.radius == b.radius` なら `true` を、それ以外なら `false` を返す。

LongInterval != LongInterval

`a != b` は、 `!(a == b)` を返す。

LongInterval < LongInterval

`LongInterval` のメンバ関数 `inf()`, `sup()` は、区間の下端・上端を「外側へ丸めて」求めるものであるが、これだけでは `LongInterval` の厳密な大小比較はできない。そこで、「内側へ丸める」プライベートメンバ関数 `infIn()`, `supIn()` を用意する。

`a < b` は、以下の三通りの結果を返す。

- `a.sup() < b.inf()` なら、「真」の意味の 1
- `a.supIn() >= b.infIn()` なら、「偽」の意味の 0
- それ以外なら、「不明」の意味の -1

LongInterval > LongInterval

`a > b` は、以下の三通りの結果を返す。

- `a.inf() > b.sup()` なら、「真」の意味の 1
- `a.infIn() <= b.supIn()` なら、「偽」の意味の 0
- それ以外なら、「不明」の意味の -1

LongInterval <= LongInterval
`a <= b` は、以下の三通りの結果を返す。

- `a.sup() <= b.inf()` なら、「真」の意味の 1
- `a.supIn() > b.infIn()` なら、「偽」の意味の 0
- それ以外なら、「不明」の意味の -1

LongInterval >= LongInterval
`a >= b` は、以下の三通りの結果を返す。

- `a.inf() >= b.sup()` なら、「真」の意味の 1
- `a.infIn() < b.supIn()` なら、「偽」の意味の 0
- それ以外なら、「不明」の意味の -1

4.3 INTLAB の実装方法との違い

LILIB は、 INTLAB のアイデアを発展させた中心値・半径方式の精度保証付き多倍長区間演算ライブラリである。以下に、 LILIB と INTLAB との差を列挙する。

4.3.1 乗算

区間同士で乗算を行う場合、

$$\langle c_a,\ r_a\rangle \cdot \langle c_b,\ r_b\rangle \ \subseteq\ \langle c_a c_b,\ |c_a| r_b + |c_b| r_a + r_a r_b\rangle$$

の式を用いる方法 [19] が良く知られている。INTLAB の多倍長区間乗算でも、この方法が用いられている。この方法は、場合分けなどが必要なく、実装が容易で速度も高速である。しかし、最悪の場合、半径が本来の 1.5 倍という過大評価が起こる。

LILIB では、区間同士の乗算に、より厳密な方法を用いている。LILIB の方法では、丸め誤差以外の誤差は発生しない。場合分けを行うので実装は複雑になるが、速度はほぼ低下しない。これは、場合分けにかかる時間が、多倍長乗算自体にかかる時間に比べて、非常に短いからである。

4.3.2 除算

INTLAB では多倍長区間除算についても、 LILIB より簡単だが誤差が大きいアルゴリズムで実装されている。

4.3.3　平方根

INTLAB では、多倍長区間の平方根演算に対応していない。

4.3.4　区間の大小比較

INTLAB では、多倍長区間の大小比較に対応していない。

4.4　LILIB 内部で使用される関数

LILIB の開発にあたっては、以下のような一般利用者のアクセスを想定していない関数を作成した。これは、 LILIB の構造の記述には必要ないが、実際にライブラリの製作を行うにあたっては参考となるものなので、以下に簡単に説明する。

4.4.1　名前空間 lilib

名前空間 `lilib` に含まれる関数は、構造上、誰でもアクセスできる。しかし、一般利用者の使用は想定していない。

- `uint32_t high32(uint64_t x)`
  64 ビット整数から、上位 32 ビットを取り出す。
- `uint32_t low32(uint64_t x)`
  64 ビット整数から、下位 32 ビットを取り出す。
- `int normalize64(uint64_t *x)`
  64 ビット整数を正規化する。ポインタ `x` が指す値の最上位ビットが 1 になるまで左シフトを行い、シフトしたビット数を返す。
- `void setRound(int round)`
  倍精度演算の丸め方向を設定する。`round` の値が負なら　$-\infty$ への丸め、 0 なら最近接丸め (偶数)、正なら　$+\infty$ への丸めとする。
- `int getRound()`
  現在の倍精度演算の丸め方向を取得する。$-\infty$ への丸めなら -1 、最近接丸め (偶数) なら 0 、　$+\infty$ への丸めなら 1 を返す。

4.4.2　多倍長実数クラス LongFloat

以下は、 `LongFloat` クラスのプライベートメンバ関数である。一般利用者はアクセスできない。

- `void normalize()`
  仮数部を正規化する。`limb[0]` が 0 でなくなるまで、 limb 単位の左シフトを行い、その分 `exponent` の値を減じる。

- `void increase()`
  仮数部の最下位ビットに 1 加える。

- `void decrease()`
  仮数部の最下位ビットから 1 減じる。

- `void copy(const LongFloat &x, int size)`
  limb 数の異なる `LongFloat` 値を代入する。一部の演算で、一時的に limb 数を変えるときに使用される。

- `void readString(std::string s)`
  10 進数の文字列を読み込むサブ関数。`setString` 関数から呼ばれる。

- `int compareAbs(const LongFloat &x) const`
  `|*this|` (自身の絶対値) と `|x|` を比較する。`|*this| < |x|` なら -1 、`|*this| == |x|` なら 0 、 `|*this| > |x|` なら 1 を返す。

- `void addAbs(const LongFloat &a, const LongFloat &b)`
  自身の値を $|a|+|b|$ にする。ただし、 $|a| \geq |b|$ でなければならない。`add`, `sub` 関数から呼ばれる。

- `void subAbs(const LongFloat &a, const LongFloat &b)`
  自身の値を $|a|-|b|$ にする。ただし、 $|a| \geq |b|$ でなければならない。`add`, `sub` 関数から呼ばれる。

- `void add(const LongFloat &a, const LongFloat &b)`
  自身の値を $a+b$ にする。加算が行われると、まずこの関数が呼ばれる。a, b の符号と絶対値を調べて、 `addAbs`, `subAbs` 関数を呼ぶ。例えば、 $-1+10$ を $-(10-1)$ に変換し、 `subAbs(10, 1)` を呼ぶ。

- `void sub(const LongFloat &a, const LongFloat &b)`
  自身の値を $a-b$ にする。減算が行われると、まずこの関数が呼ばれる。a, b の符号と絶対値を調べて、 `addAbs`, `subAbs` 関数を呼ぶ。例えば、 $-1-10$ を $-(10+1)$ に変換し、 `addAbs(10, 1)` を呼ぶ。

- `void mul(const Radius &a, const Radius &b)`
  自身の値を ab にする。

- `void mul(const LongFloat &a, int b)`
  自身の値を ab にする。

- `void mul(const LongFloat &a, const LongFloat &b)`
  自身の値を ab にする。

- `void div(const LongFloat &a, int b)`
  自身の値を a/b にする。

- `void inverse(const LongFloat &x)`
  自身の値を $1/x$ にする。除算の分母が `LongFloat` のとき、この関数が呼ばれる。

- `LongFloat sqrtInf() const`
  自身の値を、自身の値の平方根にする。計算結果は、下に丸められる。値を一度倍精度に変換して、組み込み関数 `sqrt` を用いる。誤差は大きいが、高速である。この関数は、平方根の区間演算において、半径の計算に使用される。

4.4.3 区間半径クラス Radius

以下は、 `Radius` クラスのプライベートメンバ関数である。一般利用者はアクセスできない。

- `void div(const LongFloat &a, const LongFloat &b)`
  自身の値を $|a/b|$ にする。計算結果は、上に丸められる。この関数は、区間の逆数区間を求めるとき、半径の計算に使用される。

4.4.4 精度保証付き多倍長区間クラス LongInterval

以下は、 `LongInterval` クラスのプライベートメンバ関数である。一般利用者はアクセスできない。

- `void copy(const LongInterval &x, int size)`
  limb 数の異なる `LongInterval` 値を代入する。一部の演算で、一時的に limb 数を変えるときに使用される。

- `void readString(std::string s)`
  10 進数の文字列を読み込むサブ関数。`setString` 関数から呼ばれる。読み込む過程で発生する丸め誤差も考慮される。

- `void addAbs(const LongFloat &a, const LongFloat &b)`
  自身の値を、 $|a| + |b|$ を含む区間にする。ただし、 $|a| \geq |b|$ でなければならない。`add`, `sub` 関数から呼ばれる。

- `void subAbs(const LongFloat &a, const LongFloat &b)`
  自身の値を、 $|a| - |b|$ を含む区間にする。ただし、 $|a| \geq |b|$ でなければならない。`add`, `sub` 関数から呼ばれる。

- `void add(const LongFloat &a, const LongFloat &b)`
  自身の値を、 $a + b$ を含む区間にする。加算が行われると、まずこの関数が呼ばれる。a, b の符号と絶対値を調べて、 `addAbs`, `subAbs` 関数を呼ぶ。

- `void sub(const LongFloat &a, const LongFloat &b)`
  自身の値を、 $a - b$ を含む区間にする。減算が行われると、まずこの関数が呼ばれる。a, b の符号と絶対値を調べて、 `addAbs`, `subAbs` 関数を呼ぶ。

- `void mul(const LongFloat &a, int b)`
  自身の値を、 ab を含む区間にする。

- `void mul(const LongFloat &a, const LongFloat &b)`
  自身の値を、 ab を含む区間にする。

- `void div(const LongFloat &a, int b)`
  自身の値を、 a/b を含む区間にする。

- `void inverse(const LongInterval &x)`
  自身の値を $1/x$ を含む区間にする。除算の分母が `LongInterval` のとき、この関数が呼ばれる。

- `LongFloat infIn() const`
  区間の下端の最大値を返す。区間の大小比較で使用される。

- `LongFloat supIn() const`
  区間の上端の最小値を返す。区間の大小比較で使用される。

第5章　数値実験

ここでは、以下の三つの実験の結果と、そこから得られる考察を示す。

- 5.1 では、 LILIB での点演算と区間演算の速度を比較する。これによって、中心値・半径方式の精度保証付き区間演算の、下端・上端方式に対する優位性を示す。
- 5.2 では、下端・上端方式を採用している既存の精度保証付き多倍長演算ライブラリ MPFI と、 LILIB の速度比較を行う。これによって、現時点の LILIB が、速度面で既存のライブラリに劣ることが分かる。演算速度の向上は、今後の課題である。
- 5.3 では、丸め誤差が累積しやすい問題を通して、多倍長演算と精度保証付き数値計算を組み合わせることの意義を示す。
- 5.4 では、 5.5 の前提として、 Affine 演算について簡単に紹介する。Affine 演算は、区間演算とは別の精度保証付き数値計算の手法である。区間演算と Affine 演算では、それぞれ得意とする問題が異なる。
- 5.5 では、区間演算が上手く機能しない問題を示し、 Affine 演算との比較を行う。また、区間演算が不得手とする問題でも、多倍長演算と組み合わせれば、計算結果の精度を上げられることを示す。
- 5.6 では、実際に精度保証付き多倍長演算が必要となる問題を、 LILIB で計算する。

5.1　点演算と区間演算の速度比較

ここでは、「区間演算にかかる時間 / 点演算にかかる時間」という比を考える。ここで、「点演算」とは、 `LongFloat` による精度保証なしの多倍長演算のことである。下端・上端方式の区間演算では、点演算とほぼ同じ計算を 2 回行うので、演算時間の比はおよそ 2 になる。つまり、 LILIB での演算時間の比が 2 より小さくなれば、中心値・半径方式が下端・上端方式より速いといえる。

表 5.1 は、無作為に生成した値について、点と区間の演算時間を比較した結果である。具体的な実験方法は、以下のようなものである。

1. 無作為な `LongFloat a[0]` から `a[n - 1]` までを生成する。
2. 同様に、無作為な `LongFloat b[0]` から `b[n - 1]` までを生成する。
3. `a[k]` を中心値とする `LongInterval c[k]` を生成する。ただし、 `c[k]` の半径は、中心値の最下位 limb に相当する程度の無作為な値とする。

4. 同様に、 `b[k]` を中心値とする `LongInterval d[k]` を生成する。
5. 全ての `k` に対して `a[k] + b[k]` の計算を行い、その合計時間を `n` で割ったものを、点演算の時間とする。
6. 同様に、全ての `c[k] + d[k]` の計算を行い、その合計時間を `n` で割ったものを、区間演算の時間とする。

実験は、「 Windows 7 64bit + Cygwin, Intel Core i7-4790 3.60GHz, メモリ 32GB 」の環境で行った。

表 5.1: 加減算 (指数部ランダム) の演算時間の比

桁数	試行回数	点演算の時間 [s]	区間演算の時間 [s]	演算時間の比
105	10000000	1.2600×10^{-7}	2.8500×10^{-7}	2.2619
1001	1000000	2.2100×10^{-7}	1.3920×10^{-6}	6.2986
10008	100000	2.7100×10^{-6}	9.9900×10^{-6}	3.6863

表 5.1 を見ると、演算時間の比は 2 を超えており、下端・上端方式より遅いという結果になっている。これは、 `a[k]` や `b[k]` の値が完全に無作為だからである。無作為な値同士の加減算を行うと、ほとんどの場合、二つの値の指数部は大きく異なり、「仮数部が重なっていない」状態である。この場合、点演算が値のコピーだけで済むのに対して、区間演算では半径の計算が必要になり、演算時間が長くなってしまう。ただし、ここでの半径計算はそれほど高精度である必要はなく、現時点の LILIB では計算を丁寧に行いすぎている。今後、計算を簡略化することによって、この場合の演算速度は改善できると考えられる。

表 5.2: 加減算 (指数部ゼロ) の演算時間の比

桁数	試行回数	点演算の時間 [s]	区間演算の時間 [s]	演算時間の比
105	1000000	1.5600×10^{-7}	1.7200×10^{-7}	1.1026
1001	1000000	4.2100×10^{-7}	4.3700×10^{-7}	1.0380
10008	100000	3.0800×10^{-6}	3.1600×10^{-6}	1.0260

表 5.2 は、表 5.1 の実験の `a[k]`, `b[k]` の指数部を 0 で固定したときの結果である。この場合、演算対象の「仮数部が重なっている」状態なので、「点演算の時間 + 精度保証のためのわずかな計算時間」で区間演算ができる。

このように、 LILIB の加減算では、演算対象の絶対値の差によって演算効率が大きく変わり、場合によっては下端・上端方式より遅くなってしまう。しかし、加減算は乗算・除算・平方根の計算に比べて非常に速い。中心値・半径方式によって乗算・除算・平方根の計算が十分高速化できればメリットの方が大きい、というのが我々の考えである。

表 5.3: 乗算の演算時間の比

桁数	試行回数	点演算の時間 [s]	区間演算の時間 [s]	演算時間の比
105	1000000	6.0400×10^{-7}	1.6700×10^{-6}	2.7649
1001	100000	2.6931×10^{-5}	3.1841×10^{-5}	1.1823
10008	1000	2.5611×10^{-3}	2.5951×10^{-3}	1.0133

表 5.4: 除算の演算時間の比

桁数	試行回数	点演算の時間 [s]	区間演算の時間 [s]	演算時間の比
105	100000	6.7200×10^{-6}	1.7391×10^{-5}	2.5879
1001	10000	4.1892×10^{-4}	5.7823×10^{-4}	1.3803
10008	100	5.9963×10^{-2}	7.2274×10^{-2}	1.2053

表 5.5: 平方根の演算時間の比

桁数	試行回数	点演算の時間 [s]	区間演算の時間 [s]	演算時間の比
105	100000	1.0010×10^{-5}	3.3581×10^{-5}	3.3547
1001	10000	6.1894×10^{-4}	1.2944×10^{-3}	2.0913
10008	100	8.7995×10^{-2}	1.6904×10^{-1}	1.9210

表 5.3 ・ 5.4 ・ 5.5 は、それぞれ乗算・除算・平方根の計算について演算時間の比を調べた結果である。表 5.1 と同様に、指数部も含めて無作為な値について測定した。ただし、平方根の計算については値を正に限っている。

表を見ると、桁数が小さいうちは、オーバーヘッドにより比が 2 以上の部分がある。しかし、桁数を大きくすれば、いずれの演算でも比が 2 未満になることが確認できる。特に乗除算に限れば、 10008 桁のとき、通常の多倍長演算の 1.2 倍の時間で、精度保証付き多倍長演算が可能である。

これは、半径を低精度で保持していることによる効果である。中心値・半径方式の精度保証付き数値計算にかかる時間は、ほぼ「中心値の計算時間 + 半径の計算時間」であり、半径の計算時間は半径の精度に大きく依存する。つまり、半径を低精度で保持していると、中心値の精度が大きくなるほど、半径の計算時間が相対的に小さくなり、計算時間の比は 1 に近づく。

現時点の LILIB は中心値の計算に低速なアルゴリズムを用いており、今後より高速なアルゴリズムに置き換えることも考えられる。しかし、中心値の計算のアルゴリズムが変わっても、「中心値の精度が大きくなるほど演算時間の比が 1 に近づく」という本質は変わらない。

5.2　MPFI との速度比較

ここでは、 LILIB と MPFI で同じ計算を行い、その速度を比較する。
表 5.6 は、両ライブラリで

$$\sqrt{3}+\sqrt{2}$$

を計算し、かかった時間を比較した結果である。実験方法は、次のようなものである。

1. LILIB で、 `sqrt` 関数によって「 $\sqrt{3}$ を含む区間」を求め、 `LongInterval a` とする。
2. 同様に、 $\sqrt{2}$ を含む区間を `LongInterval b` とする。
3. 用意した `a`，`b` に対して、 `a + b` の計算にかかる時間を測定する。
4. MPFI でも同様の `a`，`b` を用意し、 `a + b` の計算にかかる時間を測定する。

LILIB と MPFI では桁数の設定方法が異なるが、両ライブラリで目標桁数以上の最小の桁数を用いた。
実験は、「 Windows 10 64bit + Cygwin, AMD A4-6320 3.80GHz, メモリ 4GB 」の環境で行った。

表 5.6: 加算の速度比較

桁数	LILIB		MPFI		演算時間の比
	試行回数	演算時間 [s]	試行回数	演算時間 [s]	
100	1000000	1.5900×10^{-7}	1000000	1.8700×10^{-7}	8.5027×10^{-1}
1000	1000000	4.1200×10^{-7}	1000000	1.7100×10^{-7}	2.4094
10000	1000000	2.5810×10^{-6}	1000000	8.5800×10^{-7}	3.0082

加算の場合と同様に、

- 表 5.7 は $\sqrt{3}-\sqrt{2}$ の計算時間
- 表 5.8 は $\sqrt{3}\cdot\sqrt{2}$ の計算時間
- 表 5.9 は $\sqrt{3}/\sqrt{2}$ の計算時間
- 表 5.10 は $\sqrt{\sqrt{2}}$ の計算時間

を調べた結果である。

表 5.7: 減算の速度比較

桁数	LILIB		MPFI		演算時間の比
	試行回数	演算時間 [s]	試行回数	演算時間 [s]	
100	1000000	1.7400×10^{-7}	1000000	3.4300×10^{-7}	5.0729×10^{-1}
1000	1000000	4.4400×10^{-7}	1000000	4.2100×10^{-7}	1.0546
10000	1000000	3.1190×10^{-6}	1000000	1.1230×10^{-6}	2.7774

表 5.8: 乗算の速度比較

桁数	LILIB		MPFI		演算時間の比
	試行回数	演算時間 [s]	試行回数	演算時間 [s]	
100	1000000	1.4720×10^{-6}	1000000	3.4300×10^{-7}	4.2915
1000	100000	2.9471×10^{-5}	1000000	2.6200×10^{-6}	1.1248×10
10000	1000	2.5541×10^{-3}	100000	7.6120×10^{-5}	3.3554×10

表 5.9: 除算の速度比較

桁数	LILIB		MPFI		演算時間の比
	試行回数	演算時間 [s]	試行回数	演算時間 [s]	
100	100000	1.3400×10^{-5}	1000000	7.6400×10^{-7}	1.7539×10
1000	10000	2.8782×10^{-4}	100000	4.2100×10^{-6}	6.8366×10
10000	100	2.5371×10^{-2}	10000	1.3880×10^{-4}	1.8279×10^{2}

表 5.10: 平方根の計算の速度比較

桁数	LILIB		MPFI		演算時間の比
	試行回数	演算時間 [s]	試行回数	演算時間 [s]	
100	100000	2.8731×10^{-5}	1000000	7.4800×10^{-7}	3.8410×10
1000	10000	1.0565×10^{-3}	1000000	4.0240×10^{-6}	2.6255×10^{2}
10000	100	1.1874×10^{-1}	10000	1.2630×10^{-4}	9.4014×10^{2}

現時点での LILIB は、小さな桁数の加減算以外、全ての演算において MPFI に演算速度で劣っている。各演算においては桁数が大きいほど MPFI が有利、同じ桁数においては複雑な演算ほど MPFI が有利である。これは、 MPFI のベースとなっている GMP の

特性に起因する。GMP では、扱う桁数によって最適なアルゴリズムを自動的に使い分ける。その高速化の影響は、大きな桁数・複雑な演算ほど大きい。単純な加減算などでは、高速化の工夫の余地も少ないのである。どちらにせよ、我々が独自に実装した LILIB では、複数の専門家たちが長年かけて実装してきた GMP の速度に追いつくのは困難である。しかし、我々はこの課題の解決策を考えている。解決策については、 6.2.1 で述べる。

5.3　丸め誤差が累積しやすい漸化式

以下の漸化式を考える。この式は、参考文献 [20] から引用したものである。

$$\begin{aligned} a_{n+2} &= \frac{34}{11}a_{n+1} - \frac{3}{11}a_n \\ a_0 &= 1 \\ a_1 &= \frac{1}{11} \end{aligned}$$

この漸化式の一般解は、

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

である。しかし、漸化式の手順通りに計算すると、丸め誤差が累積・拡大し、非常に誤差が大きくなる。

表 5.11 に、 a_n の

- 真値
- 倍精度浮動小数点数 (`double`) での計算結果
- kv による精度保証付き倍精度区間演算
  (`interval<double>`) での計算結果

を示す。

表 5.11: a_n の真値と倍精度での計算結果

n	真値 a_n	double	interval ＜double＞		
			中心値	半径	相対誤差
0	1.00	1.00	1.00	0.00	0.00
1	9.09×10^{-2}	9.09×10^{-2}	9.09×10^{-2}	6.94×10^{-18}	7.63×10^{-17}
2	8.26×10^{-3}	8.26×10^{-3}	8.26×10^{-3}	8.33×10^{-17}	1.01×10^{-14}
3	7.51×10^{-4}	7.51×10^{-4}	7.51×10^{-4}	2.67×10^{-16}	3.56×10^{-13}
4	6.83×10^{-5}	6.83×10^{-5}	6.83×10^{-5}	8.49×10^{-16}	1.24×10^{-11}
5	6.21×10^{-6}	6.21×10^{-6}	6.21×10^{-6}	2.70×10^{-15}	4.34×10^{-10}
6	5.64×10^{-7}	5.64×10^{-7}	5.64×10^{-7}	8.57×10^{-15}	1.52×10^{-8}
7	5.13×10^{-8}	5.13×10^{-8}	5.13×10^{-8}	2.72×10^{-14}	5.31×10^{-7}
8	4.67×10^{-9}	4.67×10^{-9}	4.67×10^{-9}	8.65×10^{-14}	1.85×10^{-5}
9	4.24×10^{-10}	4.24×10^{-10}	4.24×10^{-10}	2.75×10^{-13}	6.48×10^{-4}
10	3.86×10^{-11}	3.88×10^{-11}	3.86×10^{-11}	8.73×10^{-13}	2.26×10^{-2}
11	3.50×10^{-12}	4.09×10^{-12}	3.52×10^{-12}	2.77×10^{-12}	7.87×10^{-1}
12	3.19×10^{-13}	2.09×10^{-12}	3.76×10^{-13}	8.81×10^{-12}	2.34×10
13	2.90×10^{-14}	5.33×10^{-12}	2.02×10^{-13}	2.80×10^{-11}	1.39×10^{2}
14	2.63×10^{-15}	1.59×10^{-11}	5.21×10^{-13}	8.89×10^{-11}	1.71×10^{2}
15	2.39×10^{-16}	4.77×10^{-11}	1.56×10^{-12}	2.82×10^{-10}	1.81×10^{2}
16	2.18×10^{-17}	1.43×10^{-10}	4.67×10^{-12}	8.97×10^{-10}	1.92×10^{2}
17	1.98×10^{-18}	4.30×10^{-10}	1.40×10^{-11}	2.85×10^{-9}	2.03×10^{2}
18	1.80×10^{-19}	1.29×10^{-09}	4.20×10^{-11}	9.05×10^{-9}	2.15×10^{2}
19	1.64×10^{-20}	3.87×10^{-09}	1.26×10^{-10}	2.88×10^{-8}	2.28×10^{2}
20	1.49×10^{-21}	1.16×10^{-08}	3.78×10^{-10}	9.14×10^{-8}	2.42×10^{2}

a_n の真値と `double` での計算結果を比較すると、 $n = 11$ の時点で 1 桁も一致しなくなっている。さらに、 `interval<double>` での計算結果を見ると、相対誤差がどのように拡大しているかが分かる。しかし、精度保証付き数値計算によって、計算精度そのものが向上しているわけではない。

次に、同じ計算を `LongInterval` で行った結果を示す。

図 5.1: 105 桁で計算した a_n の相対誤差

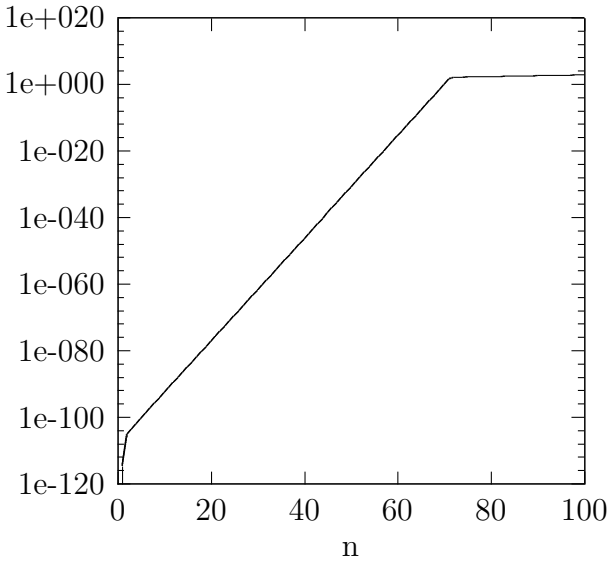


図 5.2: 1001 桁で計算した a_n の相対誤差

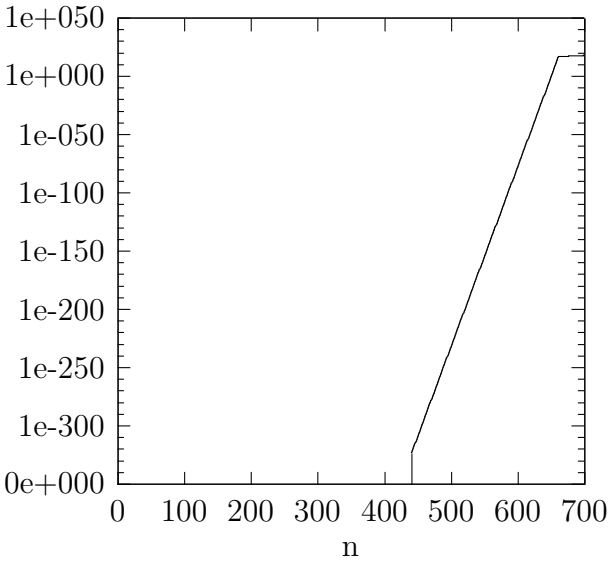


図 5.3: 10008 桁で計算した a_n の相対誤差

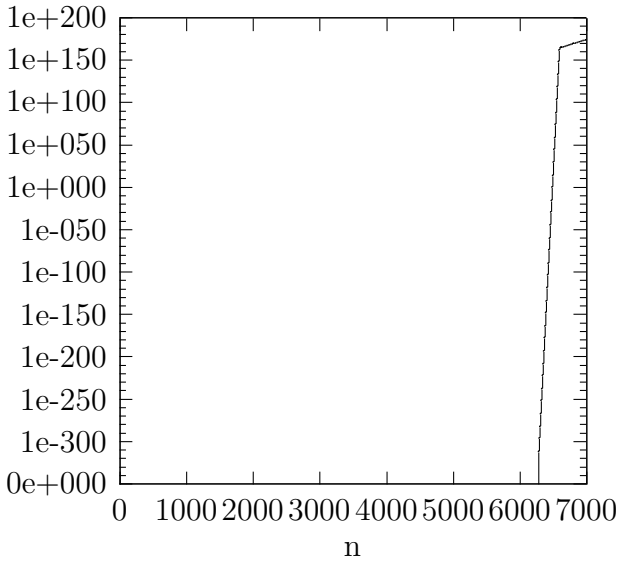


図 5.1, 5.2, 5.3 は、それぞれ桁数 105, 1001, 10008 の `LongInterval` で漸化式を計算したときの、 a_n の相対誤差を示したものである。なお、図 5.2, 5.3 では、相対誤差が 10^{-324} 程度より小さい部分は、正しくプロットされていない。これは、 gnuplot が 倍精度浮動小数点数の範囲内の値しか扱えないためである。

ここで、相対誤差がいつ 1 以上になるかに注目すると、

- 105 桁のとき、 $n = 70$
- 1001 桁のとき、 $n = 651$
- 10008 桁のとき、 $n = 6487$

である。多倍長演算の精度を大きくすると、計算結果の相対誤差が小さくなり、より大きな n まで計算できることが分かる。また、この様に計算誤差の変化を観測できるのは、精度保証付き数値計算の利点である。

5.4 Affine 演算

次節の QRT 写像に関する数値実験では、比較対象として kv による精度保証付き倍精度 Affine 演算を用いる。そこで、まず Affine 演算について簡単に紹介する。この節は、参考文献 [21] を要約したものである。

Affine 演算とは、変数間の相関性を考慮することにより、区間演算の過大評価の問題を解決する方法の一つである。Affine 演算では、変動範囲が

$$-1 \leq \varepsilon_k \leq 1$$

であるようなダミー変数 ε_k を用いて、その線形結合

$$a_0 + a_1\varepsilon_1 + \cdots + a_n\varepsilon_n$$

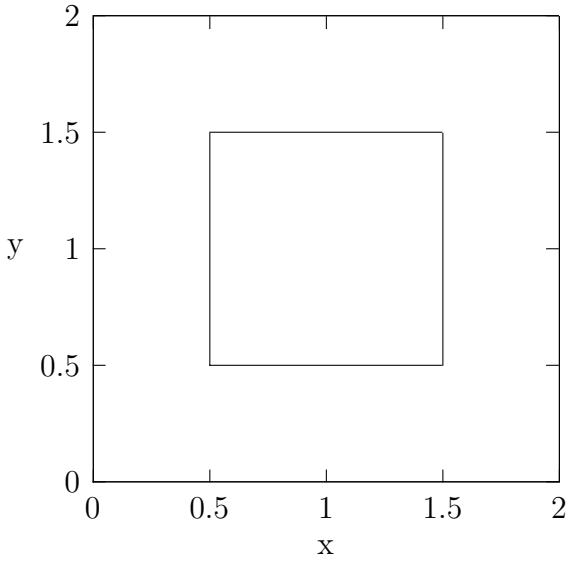
の形 (Affine 形式) で数 (区間) を表現する。計算機にはその係数を記憶する。ダミー変数の個数 n は、必要に応じて、計算の途中で変化させる。

例として、 Affine 形式

$$\begin{aligned} x &= 1 + 0.5\varepsilon_1 \\ y &= 1 + 0.5\varepsilon_2 \end{aligned}$$

を考える。これは、 x と y に相関がない状態である。このとき、 (x, y) が取り得る領域は、図 5.4 のようになる。

図 5.4: x と y に相関がない場合

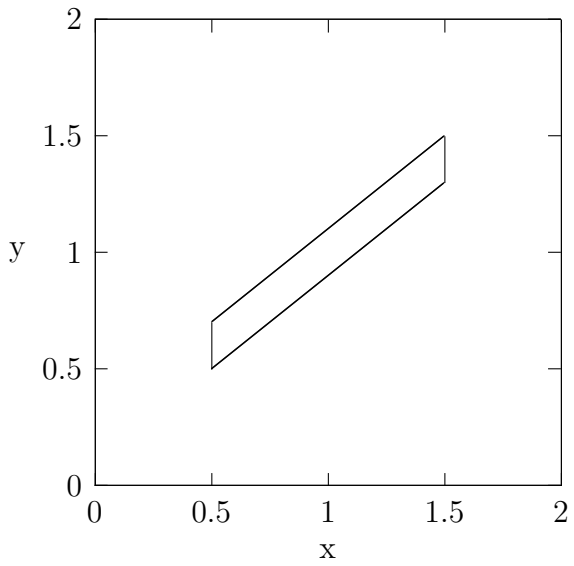


次に、

$$\begin{aligned} x &= 1 + 0.5\varepsilon_1 \\ y &= 1 + 0.4\varepsilon_1 + 0.1\varepsilon_2 \end{aligned}$$

の場合を考える。このとき、 x, y それぞれが取り得る範囲は $[0.5,\ 1.5]$ で変っていないが、両者には、 ε_1 による強い相関がある。このとき、 (x, y) が取り得る領域は、図 5.5 のようになる。

図 5.5: x と y に相関がある場合



この場合、 x と y の相関を記述できない区間演算よりも、 Affine 演算の方が、より狭い領域を記述できる。この特徴は、区間ベクトルの演算結果に対する区間拡大である wrapping effect [23] が発生する問題で、特に有利に働く。

Affine 演算の実装例として、以下に加減算と乗算の場合を示す。

5.4.1 加減算

Affine 形式間の加減算は、以下のように定義される。

$$\begin{aligned} x &= x_0 + x_1\varepsilon_1 + \cdots + x_n\varepsilon_n \\ y &= y_0 + y_1\varepsilon_1 + \cdots + y_n\varepsilon_n \\ x \pm y &= (x_0 \pm y_0) + (x_1 \pm y_1)\varepsilon_1 + \cdots + (x_n \pm y_n)\varepsilon_n \end{aligned}$$

5.4.2 乗算

Affine 形式間の乗算には、いくつかの実装方法があるが、良く用いられるのは以下の方法である。

$$xy = x_0y_0 + \sum_{i=1}^{n}(y_0x_i + x_0y_i)\varepsilon_i + \left(\sum_i^n |x_i|\right)\left(\sum_i^n |y_i|\right)\varepsilon_{n+1}$$

この方法では、 n を 1 増やし、誤差を ε_{n+1} の項で包含することによって、誤差を比較的小さく抑えることができる。

上記の実装例は、丸め誤差を考慮しないものである。丸め誤差を考慮する場合、実装はより複雑なものになる。

5.5　QRT 写像

この節で扱う問題は、早稲田大学の高橋大輔教授の示唆によるものである [22] 。

以下の Quispel-Roberts-Thompson(QRT) 写像 (漸化式) を考える。

$$\begin{aligned} x_{n+1} &= \frac{1+2x_n}{x_{n-1}x_n^2} \\ x_0 &= 1 \\ x_1 &= 1 \end{aligned}$$

この漸化式は、非線形性が強いため、区間演算を行うと、計算の度に過大評価が引き起こされる。これに対して、 Affine 演算では、非線形項間の関連がある程度考慮されるため、区間演算に比して過大な評価を抑えることができる。この漸化式を、 kv による精度保証付き倍精度 Affine 演算 (`affine<double>`) で計算した結果を、表 5.12 に示す。

実験は、「 Windows 10 64bit + Cygwin, AMD A4-6320 3.80GHz, メモリ 4GB 」の環境で行った。

表 5.12: affine ＜ double ＞ による x_n の計算結果

n	中心値	半径	計算時間
2000	7.5642×10^{-1}	2.4104×10^{-12}	1.2030
4000	6.8457×10^{-1}	2.1190×10^{-12}	4.8980
6000	7.5226×10^{-1}	9.4185×10^{-12}	1.1336×10
8000	9.8995×10^{-1}	4.8322×10^{-11}	2.0453×10
10000	1.4728	1.4445×10^{-10}	3.3656×10

次に、同じ問題を `LongInterval` で計算した結果を、表 5.13 に示す。ここでは、各 x_n の区間半径が `affine<double>` のとき以下になるように、多倍長演算の桁数を調整した。

表 5.13: LongInterval による x_n の計算結果

n	桁数	中心値	半径	計算時間
2000	982	7.5642×10^{-1}	3.7167×10^{-16}	1.6635
4000	1955	6.8457×10^{-1}	5.5248×10^{-20}	1.4477×10
6000	2918	7.5226×10^{-1}	3.4480×10^{-14}	5.4305×10
8000	3891	9.8995×10^{-1}	4.7709×10^{-18}	1.2260×10^{2}
10000	4855	1.4728	2.4976×10^{-12}	2.5496×10^{2}

倍精度小数の有効桁数が 16 程度であることを考えると、 Affine 演算の誤差は十分小さいと言える。表 5.12, 5.13 から、多倍長区間演算で Affine 演算と同等の結果を得るためには、大きな桁数が必要になり、計算時間もかかることが分かる。この様に、非線形性の影響が大きい問題に対しては、多倍長区間演算より、 Affine 演算の方が精度・速度両面で有利である。逆に、区間演算が不得意とする問題であっても、多倍長演算の精度を上げ時間さえかければ、 Affine 演算に追いつけるとも言える。また、全ての問題で Affine 演算が有効なわけではない。例えば、 5.3 節で挙げた漸化式は線形問題であるため、 Affine 演算では倍精度の区間演算と同等の結果しか得られない。

5.6 臨界 Reynolds 数の計算

この節で扱う問題は、流体力学で実際に使われているものである。詳細については、九州大学の渡部善隆准教授から教示を受けている [24] 。

5.6.1 問題

流体力学の基礎方程式のひとつである Orr-Sommerfeld 方程式

$$\begin{cases} (-D^2+a^2)^2u + iaR[V(-D^2+a^2)+V'']u = \lambda(-D^2+a^2)u, \quad \text{on} \ \ \Omega=[x_1,x_2] \\ u(x_1)=u(x_2)=u'(x_1)=u'(x_2)=0 \end{cases} \tag{5.1}$$

を考える。

式 (5.1) は非圧縮性流れの線形安定性理論から導かれる非自己共役固有値問題であり、$D=d/dx$ 、 i は虚数単位、 $a>0$ は波数、 $R>0$ は Reynolds 数、 $V \in C^2(\Omega)$ は基本流れを表す [25] 。

流れの不安定化を起こす最小の Reynolds 数 (臨界 Reynolds 数) の精度保証付き数値計算のため、固有関数 u と固有値 λ を実部と虚部

$$\begin{aligned} u &= u_r + iu_i \\ \lambda &= \lambda_r + i\lambda_i \end{aligned}$$

に分解し、

$$Re(\lambda) = \lambda_r = 0$$

となる中立曲線を考える。各 u, R, λ が波数 a に依存すると見なし、式 (5.1) を a で微分する。臨界 Reynolds 数となる条件 $R_a = 0$ を代入し、

$$\begin{aligned} v_r &= \frac{du_r}{da} \\ v_i &= \frac{du_i}{da} \\ \mu_i &= \frac{d\lambda_i}{da} \end{aligned}$$

と置くことで

$$
\begin{aligned}
(-D^2+a^2)^2u_r &= aR[V(-D^2+a^2)+V'']u_i - \lambda_i(-D^2+a^2)u_i \\
(-D^2+a^2)^2u_i &= -aR[V(-D^2+a^2)+V'']u_r + \lambda_i(-D^2+a^2)u_r \\
(-D^2+a^2)^2v_r &= -4a(-D^2+a^2)u_r \\
&\quad +\Big(R[V(-D^2+a^2)+2a^2V+V''] - \mu_i(-D^2+a^2) - 2\lambda_i a\Big)u_i \\
&\quad +\Big(-\lambda_i(-D^2+a^2) + aR[V(-D^2+a^2)+V'']\Big)v_i \qquad (5.2) \\
(-D^2+a^2)^2v_i &= -4a(-D^2+a^2)u_i \\
&\quad +\Big(-R[V(-D^2+a^2)+2a^2V+V''] + \mu_i(-D^2+a^2) + 2\lambda_i a\Big)u_r \\
&\quad +\Big(\lambda_i(-D^2+a^2) - aR[V(-D^2+a^2)+V'']\Big)v_r
\end{aligned}
$$

を得る。

5.6.2　計算

適当に与えた正規化条件のもと、方程式系 (5.2) を満たす $[u_r, u_i, v_r, v_i, a, R, \lambda_i, \mu_i]$ の包み込みを考える。無限次元 Newton 法に基づく計算機援用証明理論を適用する場合には、系 (5.2) に対する近似解の残差を可能な限り小さく取ることが要請される [26] 。高精度な近似解を得るため、課せられた境界条件を満足する Legendre 多項式 [27] を基底とした系 (5.2) の近似解を Newton-Raphson 法で求める。

しかし、 Orr-Sommerfeld 方程式は丸め誤差の影響を強く受けるため、通常の計算では満足な残差を得ることができない。例えば、

$$
\begin{aligned}
V &= 1-x^2 \\
x &= \begin{pmatrix} -1 \\ 1 \end{pmatrix}
\end{aligned}
$$

において INTLAB を用いた L^2 -ノルムの意味での残差の上限は、 200 次の Legendre 多項式において 1.97187×10^{-3} であり、最終的な解の検証のためには不十分である。

ここで、同様の計算を 10 進数 100 桁の `LongInterval` で行ったところ、

$$
\left\langle 1.1801 \times 10^{-23},\ 9.1922 \times 10^{-76} \right\rangle
$$

という微小な残差を得ることに成功した。実際の計算は、 Mathematica で残差を表す有理式を出力し、これを LILIB のソースコードに書き換える、という手順で行った。このことにより、系 5.2 の解の包み込みの展望が大きく拓けた。

第6章　まとめ

6.1　得られた成果

- 四則演算と平方根の計算について、中心値・半径方式の精度保証付き多倍長区間演算が実装できた。
- 乗算・除算・平方根の計算について、中心値・半径方式の特性を考慮したアルゴリズムを考案、実装できた。
- 精度保証付き多倍長区間演算の実装方法として、下端・上端方式より中心値・半径方式の方が優れている部分が確認できた。
- 中心値・半径方式の精度保証付き多倍長区間演算において、半径を低精度で保持することが、計算効率の向上に寄与することを示せた。
- LILIB は、`https://osdn.jp/projects/lilib/` で公開し、一般の使用のために提供している。LGPL ライセンスのオープンソースソフトウェアである。

6.2　今後の課題

LILIB では、基本的な精度保証付き多倍長区間演算が可能であるが、以下の点で、ライブラリとして不完全である。今後、これらの点を補っていきたい。

- `LongFloat` の値を、短い桁数の近似値でしか表示できない。これでは不便なので、長い桁数で表示できるようにしたい。
- `LongMatrix, LongIntervalMatrix` では、空行列を扱えない。
- 各演算で、アンダーフロー・オーバーフローのチェックを行っていない。このため、絶対値が非常に小さい・大きい値を扱ったとき、不正な結果を返すことがある。

6.2.1　演算速度の改善

LILIB を開発することによって、精度保証付き多倍長区間演算の実装方法として、下端・上端方式より中心値・半径方式の方が優れている部分が確認できた。しかし、現在の LILIB の計算速度は、下端・上端方式で実装された MPFI よりもかなり遅い。これは、MPFI のベースとなっている GMP が非常に高速であることに起因する。GMP は、アセ

ンブラによる高度な最適化が行われており、 C++ による LILIB の実装では、その速度に追いつくのは困難である。

そこで今後は、中心値・半径方式の LILIB のアルゴリズムを生かしつつ、内部での多倍長演算を GMP や MPFR に置き換えていきたい。

比較的簡単なのは、 MPFR を直接的に利用する方法である。`LongFloat` と `Radius` をそれぞれ、高精度・低精度の MPFR の多倍長実数で置き換えるのである。`LongInterval` での精度保証は、 MPFR の correct rounding という性質を利用して行う。この方法で、MPFI より高速な精度保証付き多倍長区間演算が実現できると予想している。しかし、この方法には無駄がある。MPFR での correct rounding の実現自体が限定的な精度保証であり、その上で精度保証を行うのは、二度手間と言える。

この無駄を解消するためには、 MPFR ではなく GMP を用いることになる。
具体的には、 `LongFloat` の仮数部を、 GMP の整数型で置き換えるのである。この場合、丸め処理におけるビット操作などは、 GMP の整数型に合わせて新たに実装する必要があり、単純に GMP の機能を呼び出すだけではすまなくなる。

以上を踏まえつつ、まずは実用的なバージョンとして、 MPFR を利用して高速化したものを実装したい。

謝辞

本研究を進めるにあたり、ご指導を頂いた山本野人教授に感謝します。また、 LILIB 開発のヒントとなったライブラリ INTLAB を開発した Hamburg University of Technology の Siegfried M. Rump 教授、数値実験の比較対象としたライブラリ kv を開発した早稲田大学の柏木雅英教授、数値実験の例題を参考にさせていただいた京都大学の藤原宏志助教、早稲田大学の高橋大輔教授、九州大学の渡部善隆准教授に感謝します。

参考文献

[1] IEEE standard for binary floating-point arithmetic, ANSI/IEEE Std 754-2008.

[2] GNU Multi-Precision Library, `http://gmplib.org/`.

[3] Laurent Fousse, Guillaume Hanrot, Vincent Lefévre, Patrick Pélissier, Paul Zimmermann, "MPFR:A Multiple-Precision Binary Floating-Point Library With Correct Rounding", ACM Transactions on Mathematical Software, Vol.33, No.2, Article 13, 2007.

[4] 藤原宏志, exflib, `http://www-an.acs.i.kyoto-u.ac.jp/~fujiwara/exflib/`.

[5] 柏木雅英, kv, `http://verifiedby.me/kv/`.

[6] XSC, `http://www2.math.uni-wuppertal.de/~xsc/`.

[7] Knuppel O., "PROFIL/BIAS-A fast interval library", Computing, 53, pp.277–287, 1994.

[8] CAPD library, `http://capd.ii.uj.edu.pl/`.

[9] INTLIB, `https://people.sc.fsu.edu/~jburkardt/f_src/intlib/intlib.html`.

[10] S.Rump, "INTLAB - INTerval LABoratory", Developments in Reliable Computing (eds. T.Csendes), Kluwer Academic Publishers, pp.77–104, 1999.

[11] 久保田光一, 伊理正夫, 「アルゴリズムの自動微分と応用」, コロナ社, 1998.

[12] N.Revol, F.Rouillier, MPFI,
`http://perso.ens-lyon.fr/nathalie.revol/software.html`.

[13] E.Loh, and G.W.Walster, "Rump's example revised", Reliable Computing, Vol.8, No.3, pp.245–248, 2002.

[14] 中尾充宏, 渡部善隆, 「実例で学ぶ精度保証付き数値計算 理論と実装」, サイエンス社, 2011.

[15] A.Karatsuba, Yu.Ofman, "Multiplication of multidigit numbers on automata", Soviet Physics Doklady, Vol.7, pp.595–596, 1963.

[16] Donald E.Knuth, 有澤誠 (監訳), 和田英一 (監訳), 斎藤博昭 (訳), 長尾高弘 (訳), 松井祥悟 (訳), 松井孝雄 (訳), 山内斉 (訳), 「The Art of Computer Programming Volume 2 Seminumerical algorithms Third Edition 日本語版」, アスキー, 2004.

[17] The GNU Lesser General Public License, `http://www.gnu.org/licenses/lgpl.html`.

[18] R.Rihm, "Interval methods for initial value problems in ODEs", Topics in validated computations (ed. by J.Herzberger), Elsevier, North-Holland, Amsterdam, 1994.

[19] 大石進一, 「精度保証付き数値計算」, コロナ社, 2000.

[20] 藤原宏志, 「高速多倍長計算環境における数値解析」, 日本応用数理学会論文誌, 15(3), pp.403–417, 2005.

[21] 柏木雅英, 「Affine Arithmetic について」, `http://verifiedby.me/kv/affine/affine.pdf`.

[22] 高橋大輔, 「Quispel-Roberts-Thompson(QRT) 写像の精度保証？」, private communication, 2014.

[23] 広中平祐 (編集代表), 「第 2 版 現代数理科学事典」, 丸善株式会社, 2009.

[24] 渡部善隆, 「臨界 Reynolds 数の精度保証付き数値計算」, private communication, 2016.

[25] P.G.Drazin, "Introduction to Hydrodynamic Stability", Cambridge University Press, Cambridge, 2002.

[26] 渡部善隆, "Orr-Sommerfeld 方程式の臨界 Reynolds 数に対する精度保証付き数値計算 (中)", 応用数学合同研究集会, 龍谷大学, 2015 年 12 月 17 日.

[27] T. Kinoshita, M.T.Nakao, "On very accurate enclosure of the optimal constant in the a priori error estimates for H_0^2-projection", Journal of Computational and Applied Mathematics, 234, pp.526–537, 2010.

関連論文の印刷公表の方法および時期

- Nozomu Matsuda, Nobito Yamamoto, “On the basic operations of Interval Multiple-Precision Arithmetic with center-radius form”, Nonlinear Theory and Its Applications, IEICE 2(1), pp.54–67, 2011.
  (第 3 章・第 4 章の内容に関連)

参考論文の印刷公表の方法および時期

- 山本野人, 松田望, 「多倍長演算を利用した Bessel 関数の精度保証付き数値計算」, 日本応用数理学会論文誌, Vol.15, No.3, pp.347–359, 2005.
- 山本野人, 松田望, 「精度保証付き多倍長演算の方法と構成」, 計測と制御, Vol.49, No.5, pp.297–302, 2010.

著者略歴

松田 望 (まつだ のぞむ)

1979 年 1 月 東京都に生まれる

1998 年 4 月 電気通信大学 電気通信学部 情報工学科 入学

2005 年 3 月 同 卒業

2005 年 4 月 電気通信大学 電気通信学研究科 情報工学専攻 博士前期課程 入学

2007 年 3 月 同 博士前期課程 修了

2007 年 4 月 同 博士後期課程 入学

2016 年 3 月 同 博士後期課程 修了予定