卒業論文

多倍長演算ライブラリと機械区間演算を利用した IEEE 754に基づく複素数演算の最近点丸めの実装

指導教授

近藤 弘一 教授

田中 智之 助教

同志社大学理工学部電子工学科

2020年度 1116203040番

籾倉 丸紀

2024年2月21日

目 次

1　はじめに

現代科学技術では，数値計算が不可欠であり，気象予報から財務分析，エンジニアリング設計，医療研究，物理学，天文学に至るまで幅広い分野で活用されている．しかし，コンピュータが常に正確な値を提供するわけではなく，計算結果が近似値であることは珍しくない．これは、コンピュータが扱える数値の桁数に制限があることが一因である．

コンピュータの数値計算では，レジスタに数値を格納する際，記憶可能な桁数を超えた部分が切り捨てられることで誤差が発生し，演算が複雑になるほどこれが蓄積され，信頼性のある桁数が減少する．しかし，IEEE 754 標準に基づく浮動小数点数の取り扱い，ルンゲ＝クッタ法やテイラー展開による誤差の最小化，共役勾配法やジャコビ法を用いた大規模な線型方程式の効率的解決など，過去に開発されたアルゴリズムや計算法則がこれらの誤差を実用において無視できる程度に抑制している．これにより，多くの科学者やエンジニアは日常的にこれらの誤差を無視できている．本研究は，さらなる誤差の縮小方法を探求することから始まる．

前述の通り，コンピュータの数値を格納する桁数には限界があるものの，特定のライブラリを使用して，ビット数（二進数の桁数）を人為的に増やすことが可能である（多倍長）．基本的に，桁数が多ければ演算結果の誤差が減少し，理論上の真値により近い結果を得られる．ただし，C 言語の標準ライブラリでは多倍長演算の直接的なサポートは提供されておらず，MPFR という外部ライブラリを別途，インクルードする必要がある．

ところで，多くのプログラミング言語が準拠する IEEE 754 の丸め規格（浮動小数点数算術の国際標準規格）の定義によれば，四則演算の結果は，数学的に正しい値からコンピュータが表現可能な値の範囲内で最も近い値に丸められる．これは，上向き丸め（切り上げ）も下向き丸め（切り捨て）もそれぞれ真の値から最も近い浮動小数点数に丸められることを意味する．つまり，上向き丸めと下向き丸めの差は 1ulp （浮動小数点数表現における最小単位であり，数直線の 1 目盛り分のようなもの）以下になると推測される．

MPFR のマニュアルには，MPFR の四則演算や初等関数は基本的に IEEE 754 の丸め規格に準拠していると記述されている（ただし，一部の関数については明記されていない.）．そのため，MPFR の演算に関しても，上向き丸めと下向き丸めの差が 1ulp に抑えられることが期待される．しかし，いくつかの理由で，必ずしもこれが保証されるとは言い切れない．なぜなら，多倍長演算などの複雑な操作では，IEEE 754 の規定と動作が異なる可能性があったり，初等関数は数学的に正しい値から最も近い値へ丸められるとは限らない，と IEEE 754 に記述されているからだ．MPFR 内の四則演算及び初等関数が IEEE 754 の丸め規格通りに実行されるか不明瞭であるため，それらを検証し，もし 1ulp を超える関数が存在する場合は，改善することを暫定的な研究テーマとした．ところが，実際の数値実験では，ほとんどの場合，区間幅が 1ulp 以内に収まることがわかった．よって，MPFR の四則演算や初等関数の追加改善は不要と判断し，研究テーマを変えることした．

新たな研究テーマを探す過程で，IEEE 754 の丸め規格と MPFR マニュアルには，複素

数演算に関する丸め規格の定義が記述されていないことに注目した．そこで，多倍長精度の複素数演算後も，実部と虚部それぞれ，上向き丸めと下向き丸めの差が 1ulp 以内に収まるプログラムの開発を最終的な研究目的に設定した．その道中で，多倍長精度による実数及び複素数の機械区間演算というプログラム（四則演算のみ）の開発が必要となった．

使用する場面によっては，この機械区間演算は，大きな価値を有する．コンピュータの計算が近似値を出力する可能性を考慮すると，数学的に正確な値を常に保証するのは困難である．この問題に対処する一手法として，真の値を区間，たとえば a 以上 b 以下のように表現し，下限と上限で定義する区間演算がある．この理論をコンピュータ上で実行するための理論が機械区間演算であり，数学的な不確実性を排除し，確かな数値を提供する．

最後に，本研究の全体の流れを概説する．初期段階では，実験よりも C 言語，C++，MATLAB を用いた数値実験によってプログラミングと機械区間演算の知識を深めることに注力した．これらの実験内容は本研究と直接的な関連はなく，詳細は付録に記載している．次に，C 言語を使用して多倍長演算に着手し，MPFR を活用した研究に本格的に取り組んだ．具体的には，研究の前半で，MPFR における多倍長精度の四則演算と初等関数の改善可能性を探ることを目指し，その後，研究テーマを多倍長複素数の四則演算に絞り込み，この目標達成に向けて研究を推進した．

本論文の構成は以下の通りである．第 2 節では，IEEE754 規格と精度保証付き数値計算に基づいて，浮動小数点数，丸め，機械区間演算，多倍長などについて説明する．また，MPFR ライブラリのマニュアルについても確認する．第 3 節では，実際のプログラム上での丸めや区間演算について説明する．そして，どのようにして多倍長精度の機械区間演算のプログラムコードを記述するか具体的に検討し，解説する．第 4 節では，実数と複素数それぞれについて多倍長精度の機械区間演算に対する成果を記述する．第 5 節では，得られた多倍長精度の機械区間演算の結果について，まとめと今後の課題について述べる．

2　精度保証付き数値計算と MPFR ライブラリ

2.1　IEEE 754

IEEE 754 は，1985 年に導入された浮動小数点算術のための国際標準規格で，現代のシステムで広く使用されている．この規格は浮動小数点数の表現，丸め処理，例外処理，演算方法を定義し，二進数および十進数での表現方法を含む．基本形式には，通常の数値だけでなく，ゼロ，負のゼロ，非正規化数，正・負の無限大，NaN（Not a Number：数ではない値）が含まれる．交換形式は，異なるシステム間での数値の正確な交換を可能にするビット列としての浮動小数点数の表現である．丸め処理には，最も近い値，ゼロ方向，正・負の無限大方向への丸めの四種類があり，非正規化数を使用して非常に小さい値を表現し，演算の精度を向上させることができる．IEEE 754 はゼロ除算，オーバーフロー，アンダーフローなどの例外処理も規定している．四則演算の結果は，指定された丸めモー

ドに従って丸められる必要がある．その他の演算（例えば、三角関数）についても，可能な限り高い精度を求めることが推奨されている．

2.2 精度保証付き数値計算

精度保証付き数値計算は，数値計算の結果の精度を数学的に検証し，信頼性を高める手法である．この分野の研究は急速に進展し，数値計算の検算が高速化されている．この手法の核心は，真の値を含む区間を浮動小数点数で定義し，その区間内に真の値を包含することである．以下では，精度保証の原理について数式や概念を具体的に説明する．

2.2.1 浮動小数点数

浮動小数点数はコンピュータで実数を表現する主要手段であり，有限桁の小数として実数を近似し，計算，記憶，通信など広範囲に利用される．その標準構成は 1 ビットの符号部，固定長の指数部，及び仮数部であり，多様な数値を効率的に表現する．主要なフォーマットには，IEEE 754（広く採用されている標準規格），IBM 方式があり，研究用途では指数部と仮数部が可変の方式が用いられることもある．これらのフォーマットは数値の表現と計算に影響を及ぼし，浮動小数点数の計算にはオーバーフローとアンダーフロー（計算結果が指数部で表現可能な範囲を超えた場合），桁落ち誤差（絶対値が近い異符号の数値の加算や減算後に生じる誤差），情報落ち誤差（大きさが異なる数値の加減算時に生じる誤差），積み残し（情報落ちが連続して生じる状況），丸め誤差（仮数部の桁数の限界により生じる誤差）など，複数の誤差が生じ得る．

2.2.2 2 進規格化浮動小数点数

2 進規格化浮動小数点数は, 以下の形式で表される．

$$s = \pm \left(\frac{1}{2} + \frac{d_2}{2^2} + \frac{d_3}{2^3} + \cdots + \frac{d_N}{2^N} \right) \times 2^e, \quad d_i \in \{0, 1\} \tag{2.1}$$

コンピュータで数値を扱うとき，レジスタに数値を一時的に格納するので，必ずビット長には制限がある．つまり，式（2.1）の指数部分 e には制限がかかる．一般に指数部分 e の最大値を $e^{\max}$, 最小値を $e^{\min}$ とする．ビット長が 32bits の単精度浮動小数点数について考えてみる．単精度浮動小数点数は符号ビット 1bit，指数部 8bits，仮数部 23bits で構成されているので，絶対値の最大値は指数部の各ビットが全て 1 で，かつ，仮数部の各ビットが全て 1 のときである．次に最小値について考える．ここで，式 (2.1) において右辺の最初の項 $\frac{1}{2}$ が 0 にならない理由は浮動小数点数の丸め操作において，サービスビットと呼ばれるビットを導入することで，より精密な結果得られるからである．サービスビットを配置するとは式（2.1）の最初の項 $\frac{d_1}{2}$ における d_1 が 1 であることを意味する．よって，0 を除く絶対値の最小値は $2^{-1} \times 2^{e_{\min}}$ である．

2.2.3　丸め

丸めについて，実数 r が集合 $\mathbf{F}$ に含まれる二つの浮動小数点数の間にある場合，大きい方への丸めを上向き丸め（round upward），小さい方への丸めを下向き丸め（round downward）と呼ぶ．このプロセスは図 1 で示され，日常での切り上げや切り捨てと同様に理解される．浮動小数点数で数値を特定のビット数で表現する際，丸めが必要になる．IEEE 754 標準では，最も近い値への丸めである最近点丸め（round to nearest）がデフォルトであるが，他の丸め方法も選択でき，これらは計算結果に微妙な差異をもたらすことがある．

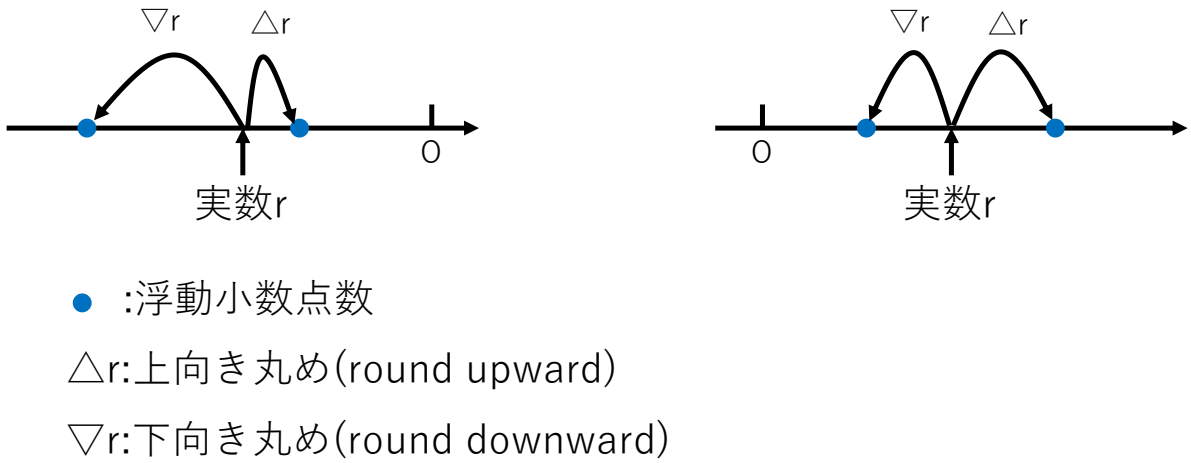

図 1: 上向き丸めと下向き丸めのイメージ図

図 1 において，上向き丸めは $\triangle : \mathbf{R} \to \mathbf{F}$, 下向き丸めは $\triangledown : \mathbf{R} \to \mathbf{F}$ と表す．

2.2.4　区間演算と機械区間演算

区間演算は，閉区間を $[\underline{x}, \overline{x}] = \{x \in \mathbf{R} \mid \underline{x} \leq x \leq \overline{x}\}$ と定義し，$\underline{x} \leq \overline{x} \in \mathbf{R}$ である場合，$\underline{x}$ を下限，$\overline{x}$ を上限とする．$\underline{x} = \overline{x}$ の場合，点区間 (point interval) と呼び，x で表す．

区間 $[x]$ ，$[y]$ に対し，四則演算は以下のように定義される．

$$[x] \circ [y] = \{x \circ y \mid x \in [x], y \in [y]\} \tag{2.2}$$

ここで，$\circ \in \{+, -, \times, /\}$ であり，これを区間演算（interval arithmetic）という．

機械区間演算は計算誤差を扱う手法で，数値はその最小値と最大値を含む区間として表現され，計算の不確実性や誤差を直接表現する．四則演算は以下で定義される．

$$[x] + [y] = \left[\triangledown\left(\underline{x} + \underline{y}\right), \triangle\left(\overline{x} + \overline{y}\right)\right] \tag{2.3}$$

$$[x] - [y] = \left[\triangledown\left(\underline{x} - \overline{y}\right), \triangle\left(\overline{x} - \underline{y}\right)\right] \tag{2.4}$$

$$[x] * [y] = \left[\min\left\{\bigtriangledown \underline{x}\underline{y}, \bigtriangledown \underline{x}\overline{y}, \bigtriangledown \overline{x}\underline{y}, \bigtriangledown \overline{x}\overline{y}\right\}, \max\left\{\bigtriangleup \underline{x}\underline{y}, \bigtriangleup \underline{x}\overline{y}, \bigtriangleup \overline{x}\underline{y}, \bigtriangleup \overline{x}\overline{y}\right\}\right] \tag{2.5}$$

$$[x]/[y] = \left[\min\left\{\bigtriangledown \underline{x}/\underline{y}, \bigtriangledown \underline{x}/\overline{y}, \bigtriangledown \overline{x}/\underline{y}, \bigtriangledown \overline{x}/\overline{y}\right\}, \max\left\{\bigtriangleup \underline{x}/\underline{y}, \bigtriangleup \underline{x}/\overline{y}, \bigtriangleup \overline{x}/\underline{y}, \bigtriangleup \overline{x}/\overline{y}\right\}\right] \tag{2.6}$$

これを基に機械区間演算を実装する．

2.2.5 多倍長

本研究で，機械区間演算とともに重要な概念である多倍長について説明する．2 進規格化浮動小数点数に関する説明で，数値をレジスタに格納するときにビット長に制限があると述べた．そのビット長をデフォルトよりも拡張することができるのが多倍長である．多倍長にすることで単精度や倍精度よりもさらに多くの数値を扱うことが可能となる．以下，プログラミングにおいて整数や小数を int 型，double 型と呼ぶように実数の多倍長を rmulti 型，複素数の多倍長を cmulti 型と呼ぶことにする．例えば，double 型では，符号部が 1bit，指数部が 11bits，仮数部が 52bits なのだが，これを rmulti 型にすることで仮数部の値を拡張でき，より精度の高い計算ができるのである．

2.3 MPFR ライブラリ

C 言語標準ライブラリでは仮数部の自由な操作ができないため, 多倍長計算を可能にする外部ライブラリの導入が必要となる．この要件を満たすため, 本研究では MPFR（Multiple Precision Floating-point Reliable）ライブラリを採用した．以下に, 本研究で用いる MPFR ライブラリの主要機能を紹介する．

2.3.1 MPFR の著作権

GNU MPFR ライブラリ（以降，MPFR）は公開されており，誰もが自由に利用及び再配布が可能なフリーライブラリである．

2.3.2 MPFR の基本的な説明

MPFR は任意の精度で浮動小数点数演算を行うことができる．ビット数で表現される精度桁数（precision）は，厳格に各変数ごとに設定される．また，低精度桁数の設定も可能である．本研究のプログラム内では精度桁数を決定する変数を prec と表現することが多い．MPFR は IEEE754-1985 規格で定められている，4 つの丸めモードをサポートしつつ，基本演算や数学関数の機能を提供している．これらのライブラリには区間演算を直接実装できるような機能は備わっていないので，多倍長と丸めモードをうまく切り替えつつ，機械区間演算を実装する必要がある．

2.3.3 ヘッダファイル

MPFR を使用するためにはインストールが必要だが，その詳細は割愛する．使用時に必要な変数や宣言は，C 言語でインクルード可能な mpfr.h に全て含まれている．

2.3.4 データ型

MPFR の内部構造について説明する．limb は，1 ワードを単位とする多倍長精度浮動小数点数の構成要素を意味している．1limb は，通常，32bits か 64bits である．limb を表す C データ型は `mp_limb_t` である．`mpfr_t` データ型は，内部的には構造体の配列の一つ分として定義されており，`mpfr_prt` 型は，この構造体へのポインタを表現しているデータ型である．`mpfr_t` データ型は，以下の 4 つのフィールドから構成されている．

- **_mpfr_prec**: 変数の仮数部の精度桁数（ビット数）を保持する．
- **_mpfr_sign**: 変数の符号を保持する．
- **_mpfr_exp**: 仮数部を保持する．
- **_mpfr_d**: LSB（Least Significant Bits）が先頭に来るリム配列へのポインタが格納される．

以下に，`__mpfr_struct` の構造体のコードを示す．

```
/* Definition of the main structure */
typedef struct {
  mpfr_prec_t  _mpfr_prec;
  mpfr_sign_t  _mpfr_sign;
  mpfr_exp_t   _mpfr_exp;
  mp_limb_t   *_mpfr_d;
} __mpfr_struct;
```

本研究中では，`__mpfr_struct` の代わりに rmulti を用いており，これは実数の多倍長型を明確に表すためであるが，構造自体は`__mpfr_struct` と同一である．しかし，構造体の中身自体は同じで，構造体の名前が異なるだけである．さらに，cmulti 型は，rmulti 型の実部と虚部を組み合わせた複素数を表すデータ型である．

2.3.5 初期化関数

多倍長浮動小数点数の初期化は，MPFR ライブラリを使用する上で重要である．初期化関数は，変数を使用可能な状態にし，必要に応じてメモリを確保する．以下に，初期化に関連する主要な関数を紹介する．

- **mpfr_set_default_prec**：デフォルトの精度桁数を設定する．引数には，設定したい精度桁数（ビット数）を指定する．精度桁数は，変数の仮数部のビット長を意味し，MPFR_PREC_MIN 以上，MPFR_PREC_MAX 以下の整数でなければならない．この関数を呼び出した後に初期化される変数は，設定した精度桁数を持つが，既に初期化されている変数の精度桁数は変更されない．デフォルトの精度桁数の初期値は 53 ビットである．

- **mpfr_get_prec**：指定された変数の精度桁数（仮数部のビット長）を返す．引数には，精度桁数を知りたい変数を指定する．

2.3.6 代入関数

MPFR ライブラリでは，変数への代入を行うための関数が提供されている．代入関数は，特定のデータ型から MPFR 変数へ値を設定するために使用される．ここでは，その中でも特に重要な 2 つの関数，`mpfr_set_d` と `mpfr_set_str` について説明する．

- **mpfr_set(rop, op, rnd)**：この関数は，値 `rmulti op` を，丸め方式 `rnd` で丸め，`rmulti rop` に代入する．実行環境が IEEE 754 標準をサポートしていない場合，符号付きゼロを正しく扱えない場合があるため注意が必要である．

- **mpfr_set_str(rop, s, base, rnd)**：この関数は，文字列 `s` を base 進数表現として解釈し，丸め方式 `rnd` で丸めた値を `rmulti rop` に代入する．文字列 `s` が完全に浮動小数点数として解釈できる場合は 0 を返し，そうでない場合は `rop` の値を書き換え，−1 を返す．

変数 rop を初期化し，丸め方式 rnd で丸めた op を代入する．rop の精度桁数は，`mpfr_set_default_prec` 関数で設定した現状のデフォルト値が設定される．

2.3.7 基本演算関数

以下に，MPFR ライブラリを使用して基本的な数学演算を行う関数を紹介する．

- **mpfr_add(rop, op1, op2, rnd), mpfr_sub(rop, op1, op2, rnd), mpfr_mul(rop, op1, op2, rnd), mpfr_div(rop, op1, op2, rnd)**：それぞれ，`rmulti op1` と `rmulti op2` の和，差，積，商を計算し，丸めモード `rnd` で丸めて `rmulti rop` に代入する．符号付きゼロの場合は IEEE 754 のルールに従う．乗算と除算に関しては，計算結果がゼロになるときの符号は，二数の符号の積で決定される．

- **mpfr_sqrt(rop, op, rnd)**：この関数は，`rmulti op` の平方根を求め，丸めモード `rnd` で丸めて `rmulti rop` に代入する．op が負の時は，rop に NaN が代入される．

- **mpfr_pow(rop, op1, op2, rnd)**：この関数は，rmulti op1 の rmulti op2 乗を計算し，丸めモード rnd で丸めて rmulti rop に代入する．特殊値に対しては，ISO C99 及び IEEE 754-2008 規格が定める pow 関数に準じた振る舞いをする．

- **mpfr_sqrt_ui(rop, unsingned long int op, rnd)**：この関数は，unsingned long int op の平方根を求め，丸めモード rnd で丸めて rmulti rop に代入する．op が負の時は，rop に NaN が代入される．

- **mpfr_ui_pow_ui(rop, unsingned long int op1, unsingned long int op2, rnd)**：この関数は，unsingned long int op1 の unsingned long int op2 乗を計算し，丸めモード rnd で丸めて rmulti rop に代入する．特殊値に対しては，ISO C99 及び IEEE 754-2008 規格が定める pow 関数に準じた振る舞いをする．

2.3.8 比較関数

以下に，MPFR ライブラリで使用される比較関数を紹介する．

- **mpfr_cmp(op1, op2)**：この関数は，rmulti op1 と rmulti op2 を比較する．op1> op2 の場合は正の値を，op1=op2 の場合はゼロを，op1<op2 の場合は負の値を返す．op1 と op2 がどちらも同じ精度桁数フルで値が入っている場合は，差を取って判断する．どちらかの値が NaN の場合は，範囲エラーフラグが立てられて，ゼロを返す．

この比較関数は 3 種類のケースを見分けることができる．比較される値に NaN がある場合は，IEEE 754 規格の比較と同じ振る舞いをする．比較は浮動小数点数のみ対象なので，必要に応じてデータ変換も行われる．

2.3.9 初等関数

以下に，MPFR ライブラリを使用して初等関数を計算する関数を紹介する．

- **mpfr_log(rop, op, rnd)**，**mpfr_log2(rop, op, rnd)**，**mpfr_log10(rop, op, rnd)**：それぞれ，rmulti op の自然対数，$\log_2$(op)，$\log_{10}$(op) を計算し，丸めモード rnd で丸めて rmulti rop に代入する．丸め方式に関わらず，op が 1 の時は，rop は+0 になる．

- **mpfr_exp(rop, op, rnd)**，**mpfr_exp2(rop, op, rnd)**，**mpfr_exp10(rop, op, rnd)**：それぞれ，rmulti op の e のべき乗，2 のべき乗，10 のべき乗を計算し，丸めモード rnd で丸めて rmulti rop に代入する．

- **mpfr_cos(rop, op, rnd)**，**mpfr_sin(rop, op, rnd)**，**mpfr_tan(rop, op, rnd)**：それぞれ，rmulti op の cos(op)，sin(op)，tan(op) の値を計算し，丸めモード rnd で丸めて rmulti rop に代入する．

引数がある領域にある場合，要求精度桁数が小さくても，初等関数の計算 (正確な丸め処理を行うため) に時間がかかることがある．例えば，三角関数の引数が大きい場合がそれにあたる．他にも，関数によっては，メモリの使用量が必ずしも出力する値の精度桁数に依存しないこともある．

2.4 独自のヘッダファイル

2.4.1 isys.h

#include<isys.h>は, 数値計算を行う際に利用する様々なデータ型と関数を提供する独自のヘッダファイルである．このヘッダファイルは複数の他のヘッダファイルをインクルードしており，整数ベクトル，整数行列，複素数ベクトル，複素数行列，多倍長実数ベクトル，多倍長実数行列など，数値計算でよく使用されるデータ型をサポートしている．また，このヘッダファイルには mpfr.h ヘッダファイルも含まれている．

多倍長実数演算では, rmulti 型を利用して多倍長の実数計算を行うことができる．rallo-cate 関数を用いて動的にメモリを確保し，rmfree 関数でメモリを解放する．

- rallocate 関数：
  rmulti 型の変数のメモリを動的に確保する関数である．関数が呼び出されると，rmulti 型のポインタが返され，浮動小数点数の値は NaN に設定される．浮動小数点数の仮数の精度 (ビット数) は，set_default_prec 関数によって設定された値が使用される．

- rmfree 関数：
  rallocate 関数によって確保された rmulti 型のメモリを解放する関数である．この関数が呼び出されると，指定された rmulti 型のポインタが解放され，NULL ポインタが返される．

また, 多倍長複素数演算では, cmulti 型を利用して多倍長の複素数計算を行う．callocate 関数を用いて動的にメモリを確保し，cmfree 関数でメモリを解放する．

2.4.2 mi.c

mi.c ファイルは，区間計算を扱うために作成された，様々な構造体と関数を集約したものである．ファイル内の型名や関数名に mi を含むものは，区間を示している．たとえば，mir_r_sqrt 関数は rmulti 型（点区間）の平方根を計算し，mi_r 型（区間）で返す．また，mic_micmic_mul 関数は mi_c 型（区間）二つの積を mi_c 型（区間）で返す．（mi_r 型や mi_c 型の詳細は本論文の第 3.1 節，p.10 及び第 3.3 節，pp.11–14 参照.）

mi.c ファイル内に関数名には，str（文字列），strstr（文字列 2 つ），r（実数の多倍長型），rr（実数の多倍長型 2 つ），c（複素数の多倍長型），cc（複素数の多倍長型 2 つ），mir（実数の多倍長型の区間），mirmir（実数の多倍長型の区間 2 つ），mic（複素数の多倍長型の区間），micmic（複素数の多倍長型の区間 2 つ），add（加算），sub（減算），mul（乗算），div（除算），allocate（メモリの確保），free（メモリの解放），set（格納）などのパターンがある．（mi.c のソースコードは付録 A.3.6，pp.26–47 参照.）

3　プログラム上での機械区間演算の実装方法とその検証

3.1　区間を表す構造体と多倍長演算の関数

本節では，多倍長精度演算における丸め誤差の検証に必要な構造体と関数について述べる．実数の区間を表す構造体と，この区間に演算結果を格納する関数を導入し，多倍長演算の精度と丸め誤差を評価する．以下は，数値実験で使用したソースコードの抜粋である．

```
// 多倍長型の上限と下限を持つ，実数の区間を表す構造体
typedef struct {
    rmulti* down;
    rmulti* up;
} mi_r;

// 2 つの実数の点区間を受け取り，加算して区間で返す関数
void mir_rr_add(mi_r* z, rmulti* x, rmulti* y) {
    mpfr_add(z->down, x, y, MPFR_RNDD);
    mpfr_add(z->up, x, y, MPFR_RNDU);
}
```

mi_r 型は多倍長精度の下限（down）と上限（up）を持つ, 区間を表す構造体である．mir_rr_add 関数は二つの数値を加算し, MPFR の丸めモード（MPFR_RNDD で切り捨て, MPFR_RNDU で切り上げ）を用いて，演算結果を切り捨てた値を区間の down に，切り上げた値を up にそれぞれ格納する．up と down の差の値から，IEEE 754 規格の丸め処理が MPFR の多倍長精度演算でどれだけ適切に機能するかを評価できる．

3.2　相対誤差

数値を区間で表す際の up と down の差を評価する方法について述べる．本研究では，以下のコードを用いて up と down の差の精度を計算する．

```
mpfr_sub(a, x->up, x->down, MPFR_RNDN);

b = mpfr_get_exp(x->up) - mpfr_get_prec(x->up);

// a/2^b を c に格納
rdiv_2exp(c, a, b);
```

このコードの意味を数式を通じて説明する．数値 s が

$$s = \pm\left(\frac{d_1}{2} + \frac{d_2}{2^2} + \frac{d_3}{2^3} + \cdots + \frac{d_N}{2^N} + \cdots\right) \times 2^e, \quad d_i \in \{0, 1\} \tag{3.1}$$

で表される場合，仮数部が N ビットまで表せると仮定すると，s の上限 $\overline{s}$ と下限 $\underline{s}$ は

$$\overline{s} = \begin{cases} +\left(\dfrac{d_1}{2} + \dfrac{d_2}{2^2} + \dfrac{d_3}{2^3} + \cdots + \dfrac{d_{N-1}}{2^{N-1}} + \dfrac{1}{2^N}\right) \times 2^e, & d_i \in \{0, 1\} \\ -\left(\dfrac{d_1}{2} + \dfrac{d_2}{2^2} + \dfrac{d_3}{2^3} + \cdots + \dfrac{d_{N-1}}{2^{N-1}} + \dfrac{0}{2^N}\right) \times 2^e, & d_i \in \{0, 1\} \end{cases} \tag{3.2}$$

$$\underline{s} = \begin{cases} +\left(\dfrac{d_1}{2} + \dfrac{d_2}{2^2} + \dfrac{d_3}{2^3} + \cdots + \dfrac{d_{N-1}}{2^{N-1}} + \dfrac{0}{2^N}\right) \times 2^e, & d_i \in \{0, 1\} \\ -\left(\dfrac{d_1}{2} + \dfrac{d_2}{2^2} + \dfrac{d_3}{2^3} + \cdots + \dfrac{d_{N-1}}{2^{N-1}} + \dfrac{1}{2^N}\right) \times 2^e, & d_i \in \{0, 1\} \end{cases} \tag{3.3}$$

と表せ，$\overline{s}$ と $\underline{s}$ の差は 2^{e-N} となる．従って，正確に丸めが行われた場合，この差と 2^{e-N} の比は 1 に等しい．

ソースコードで計算される数値 c が $c > 1$ の場合，up と down の差が 1ulp を超えていることを意味し，$c = 0$ の場合は，上限と下限が同一の点区間であることを示す．したがって，$c \leq 1$ であれば IEEE 754 の丸め規格に従っていると評価できる．

3.3 区間同士の機械区間演算

第 1 節で述べたように，MPFR を用いて四則演算や初等関数に関する計算を行うと，ほとんどの場合で，上限と下限の差が 1ulp で抑えられることがわかった．これを改善するのは困難かつ，あまり必要性がない．そこで，複素数の演算に関する丸め規格はマニュアルに書かれていないため，実数の場合と同様に 1ulp で収まりつつ，数学的に正確な複素数の演算をコンピュータでも行えるようにプログラムを作成することを目標とした．

複素数の乗除算は，複数の四則演算の連鎖によって成り立ち，この過程での丸め誤差が計算の数学的信頼性に影響を及ぼすことがある．初回の計算結果が IEEE 754 の丸め規格

に準拠していたとしても，これは一時点での入力に対してのみ数学的に正確であることを意味し，最終的な結果の数学的正確性を保証するものではない．

本来, 変数に初期値を設定する際に生じる丸めは, 数学的誤差として考慮すべきだが, 本研究では全ての初期値を最近点に丸め，これを誤差として考慮しない．これは，IEEE 754 の丸め規格が，演算時の初期値設定による丸め誤差について言及していないためである．

複数回の演算をコンピュータ上で実行する際，数値を区間として扱うことで数学的に正確な結果を保証する機械区間演算の理論を採用した．この理論に基づき，多倍長精度を活用して実数と複素数の機械区間演算のプログラムを開発し，実装した．以下では，これらのプログラムの具体的な実装方法を示す．

3.3.1 実数における区間同士の機械区間演算

区間の四則演算は第 2.2.4 節の式（2.3）〜（2.6）に基づいて実装される．数値実験で使用したソースコードの一部を以下に示す．加減算は比較的単純な一方で，乗除算は最大値や最小値の決定に分岐が必要とする複雑さを持つ．（完全なソースコードと他の四則演算の詳細は mi.c に記されており，付録 A.3.6，pp.26–47 参照.）

```
// 2つの実数の区間を受け取り，加算して区間で返す関数
void mir_mirmir_add(mi_r* z, mi_r* x, mi_r* y) {
    mpfr_add(z->down, x->down, y->down, MPFR_RNDD);
    mpfr_add(z->up, x->up, y->up, MPFR_RNDU);
}

// 2つの実数の区間を受け取り，乗算して実数の区間で返す関数
void mir_mirmir_mul(mi_r* z, mi_r* x, mi_r* y) {
    // 関数内で使用するポインタ変数の宣言
    rmulti *z_min, *z_max, *a;

    // rmulti 型のメモリを確保し，初期化
    z_min = rallocate(); // 一時的な最小値用のメモリを確保し，初期化
    z_max = rallocate(); // 一時的な最大値用のメモリを確保し，初期化
    a = rallocate(); // 一時的な値を保持するメモリを確保し，初期化

    mpfr_mul(z_min, x->down, y->down, MPFR_RNDD); // x の下限と y の下限を乗算し，下向きに丸めて一時的な最小値に格納

    mpfr_mul(a, x->down, y->up, MPFR_RNDD); // x の下限と y の上限を乗算し，下向きに丸めて一時的な変数 a に格納
```

```
    if(mpfr_cmp(a, z_min) < 0) {
        mpfr_set(z_min, a, MPFR_RNDD);
    } // 最小値を更新

    // 同様の操作を他の上限と下限の組み合わせで行い，最小値を更新（ソースコードは省略…）

    mpfr_set(z->down, z_min, MPFR_RNDD); // 最終的な下限を z の下限に格納

    // 同様の操作を上向きに丸めて行い，最終的な上限を z の上限に格納（ソースコードは省略…）
}
```

3.3.2 複素数における区間同士の機械区間演算

複素数 x，y を $x = x_r + ix_i$，$y = y_r + iy_i$，（$x_r, x_i, y_r, y_i \in \mathbb{R}$）と定義する．複素数の四則演算は以下の通りである．

$$x + y = (x_r + y_r) + i\,(x_i + y_i) \tag{3.4}$$

$$x - y = (x_r - y_r) + i\,(x_i - y_i) \tag{3.5}$$

$$xy = (x_r y_r - x_i y_i) + i\,(x_r y_i + x_i y_r) \tag{3.6}$$

$$\frac{x}{y} = \frac{x_r y_r + x_i y_i}{y_r^2 + y_i^2} + i\frac{-x_r y_i + x_i y_r}{y_r^2 + y_i^2} \tag{3.7}$$

複素数の加算と減算は実部と虚部に MPFR 関数を一度呼び出すだけで求められる．しかし，乗算と除算は実部と虚部に対して複数の四則演算が必要である．これらの演算は，実数の機械区間演算を組み合わせて実装する．

複素数の機械区間演算の除算を示すソースコードを以下に示す．（完全なソースコードと他の四則演算の詳細は mi.c に記されており，付録 A.3.6，pp.26–47 参照．）

```
// 多倍長型の上限と下限をもつ mi_r を用いた実部と虚部を表す構造体
typedef struct {
    mi_r* r;
    mi_r* i;
```

```
} mi_c;

// 複素数の区間を受け取り，除算して複素数の区間で返す関数
void mic_micmic_div(mi_c* z, mi_c* x, mi_c* y) {
    // 使用する変数の宣言
    mi_r *a, *b, *c, *d; // mi_r 型のポインタ変数の宣言

    // a，b，c，d のメモリを確保 (ソースコードは省略...)

    // 式 (3.9) の実部の分子
    mir_mirmir_mul(a, x->r, y->r); // 区間 x_r と y_r の積を区間 a に格納
    mir_mirmir_mul(b, x->i, y->i); // 区間 x_i と y_i の積を区間 a に格納
    mir_mirmir_add(c, a, b); // 区間 a と b の和を区間 c に格納

    // a，b のメモリを解放し，再確保 (ソースコードは省略...)

    // 式 (3.9) の実部の分子
    mir_mirmir_mul(a, y->r, y->r); // 区間 y_r と y_r の積を区間 a に格納
    mir_mirmir_mul(b, y->i, y->i); // 区間 y_i と y_i の積を区間 a に格納
    mir_mirmir_add(d, a, b); // 区間 a と b の和を区間 d に格納

    // 式 (3.9) の実部
    mir_mirmir_div(z->r, c, d); // 区間 c と d の商を区間 z_r に格納

    // a，b，c，d のメモリを解放 (ソースコードは省略...)

    // 同様の操作を式（3. 9）の虚部の求め方に従って行い，最終的な虚部を z_i に格
納 (ソースコードは省略...)
}
```

3.4 区間幅を 1ulp に抑えるために改良した複素数の機械区間演算

第 3.3.2 節で紹介した複素数の機械区間演算では，複数の区間演算を要するため，乗算と除算の精度が 1ulp を超える可能性があった．これを解決するため，演算前に精度を一時的に向上させ，演算後に元の精度に戻して結果を保存する手法を採用した．改良版の乗算コードを以下に示し，除算は mic_micmic_mul を mic_micmic_div に置き換えた．精度向上のために，元の 2 倍して，16 ビット増加させた理由は第 4.4 節に記述している．（完全

なソースコードと除算の詳細は mi.c に記されており，付録 A.3.6，pp.26–47 参照.）

```
// 複素数の区間を受け取り，乗算して複素数の 1ulp 区間で返す関数
void mic_micmic_1ulp_mul(mi_c* z, mi_c* x, mi_c* y) {
    // 使用する変数の宣言
    mi_c *x0, *y0, *z0;

    // 精度を元の 2 倍して 16 増やすように設定
    set_default_prec(mpfr_get_prec(x->r->down) * 2 + 16);

    // x0，y0，z0 のメモリを確保（ソースコードは省略...）

    // x，y の値をそれぞれ x0，y0 に格納
    mic_mic_set(x0, x);
    mic_mic_set(y0, y);

    // 複素数の区間同士の機械区間演算
    mic_micmic_mul(z0, x0, y0); // 複素数の区間 x0 と y0 の積を区間 z0 に格納

    // 元の精度に戻す
    set_default_prec(mpfr_get_prec(x->r->down));

    // z0 の値を z に格納
    mic_mic_set(z, z0);

    // x0，y0，z0 のメモリを解放（ソースコードは省略...）
}
```

4　研究結果とその考察

本研究の数値実験は，MacBook Pro 14 インチ 2021 （Apple M1 Pro チップ，16 GB メモリ，macOS Sonoma バージョン 14.2.1）で実施した．ターミナル上で，以下のコマンドを用いて，コンパイルと実行を行った．

```
gcc -c sample.c -I/usr/local/include -I../include -Wall -Wextra -Werror

gcc -o sample sample.o -I/usr/local/include -I../include -Wall -Wextra
-Werror  -L/usr/local/lib -L../lib -lis -lmpfr -lgmp -lm
```

数値実験で使用した変数には，3^{628}，5^{429}，$\sqrt{3}$，$\sqrt{5}$，L^{a}_{+b}（Large number），S^{c}_{-d}（Small number）が含まれる．L^{a}_{+b} は，有効桁数が a 桁，オーダーが 10^{+b} の，仮数部が 1234567890 を繰り返す Large number で，たとえば $L^{300}_{+151} = (1.23\ldots901\ldots890)\times10^{151}$ の形をとる．S^{c}_{-d} は，有効桁数が c 桁，オーダーが 10^{-d} の，仮数部が 1234567890 を繰り返す Small number で，たとえば $S^{302}_{-75} = (1.23\ldots901\ldots89012)\times10^{-75}$ の形をとる．つまり，添字の上は有効桁数，下はオーダーを示す．これらの値を選んだのは，大きな数と小さな数の組み合わせ，及び無理数を使用することで，演算時のエラー検出可能性を高めるためである．ここでのエラーは，コンパイルエラーのみならず，IEEE 754 の丸め規格に準じない動作も含む．

4.1 多倍長精度の四則演算と相対誤差（実数）

実験目的 この実験では，MPFR ライブラリが IEEE 754 の丸め規格に準拠しているかどうかを検証する．具体的には，MPFR の四則演算後の上向き丸めと下向き丸めの差が最大で 1ulp である，という仮説を数値実験によって検証する．

実験方法 実験に使用したソースコードは，付録 C.1，pp.72–73 を参照．各四則演算に応じて，mir_rr_add 関数の部分を適切な関数に置き換えて演算を行った．

実験結果 四則演算による相対誤差の表とグラフは，付録 C.2.1〜C.2.4，pp.74–81 を参照．

考察 図 2 から，入力値が同一オーダーである場合，加算操作における相対誤差が 0 から 1 の範囲で振動する傾向が確認できる．この現象は，精度（仮数部のビット数）によって，有効桁数が増加する場合とそうでない場合が存在することに起因すると推察される．具体的には，同じオーダーの数値を加算した際に桁上がりが生じると，有効桁数が増加し，結果として相対誤差が 1 になる可能性がある．一方で，桁上がりが発生しない場合，相対誤差は 0 になると考えられる．さらに，図 3 及び表 7，8 のデータから，同一オーダーでの異符号の加算や同符号の減算を行った場合に相対誤差が最大で 0 になることも明らかになった．これは，演算を通じて有効桁数が増加しないため，情報損失が生じないことに基づくと考えられる．

図 2〜5 及び表 7〜10 に示されるデータから，どの精度，どの入力値であっても，相対誤差が 1 以下であることが明らかである．この結果は，演算結果の下限と上限の差が 1ulp 以内に収まること，そして演算後の下向き丸めと上向き丸めが数学的に正しい値に最も近い浮動小数点数へと適切に丸められていることを示している．従って，多倍長精度の四則演算が IEEE 754 の丸め規格に準拠しているといえる．

4.2　多倍長精度の四則演算以外の関数と相対誤差（実数）

実験目的　この実験では，MPFR ライブラリの多倍長精度における四則演算以外の関数の演算精度を検証する．IEEE 754 の丸め規格に則った四則演算の精度を確認した第 4.1 節の実験と同様に，MPFR の非四則演算の関数においても，上向き丸めと下向き丸めの差がどの程度になるのかを評価する．マニュアルの記述から，MPFR の非四則演算関数が IEEE 754 の丸め規格に完全に準拠していない可能性があるので，この検証は重要である．

実験方法　実験に使用したソースコードは，付録 C.1，pp.72–73 を参照．各非四則演算に応じて，mir_rr_add 関数の部分を適切な関数に置き換えて演算を行った．

実験結果　非四則演算による相対誤差の表とグラフは，付録 C.2.5〜C.2.31，pp.82–110 を参照．

考察　図 25，27 から，入力値のオーダーが大きい場合，双曲線正割と双曲線余割での相対誤差が指数関数的に増大し，1 を超えることがわかる．グラフから明らかな特徴として，精度が 1 増加する毎に相対誤差が 2 倍になる傾向が観察される．この現象は，入力値のオーダーが大きい場合に IEEE 754 の丸め規格に従わない動作が発生するか，または扱う数値が大き過ぎてエラーが生じるためと推測される．さらに，表 13，17〜19，32，33 から，入力値のオーダーが大きい場合に相対誤差の最大値が顕著に増大することがわかる．これは，指数関数，べき乗，双曲線正割，双曲線余割などの多倍長精度の演算が，特定の入力において IEEE 754 の丸め規格に準拠していないことを示唆している．しかし，相対誤差が 1 を超える入力値は，オーダーが大きい特殊なケースに限られるため，改善の必要性は限定的であると考えられる．仮数部のビット数を増やす改善策を試みたものの，顕著な改善は見られなかった．これらの結果を踏まえ，MPFR 内で 1ulp を超える関数の改善から，多倍長の複素数演算を 1ulp に抑える研究目的へとシフトすることを決定した．

図 6〜24，26，28〜31 から，segmentation fault による記録の欠如があるものを除き，ほぼすべての精度で相対誤差が 1 であることがわかる．これは，演算後に有効桁数が増加し，結果として情報の損失が生じるためであると考えられる．また，図 6，7，9〜11，15〜24，29〜31 及び表 11，12，14〜16，20〜31，34〜37 から，segmentation fault による記録の欠如があるものを除き，どの入力値でも，相対誤差の最大値は 1 以下であることがわかる．これは，多倍長精度の平方根，平方根の逆数，対数関数，余弦，正弦，正接，正割，余割，余接，余弦の逆関数，正弦の逆関数，正接の逆関数，双曲線余弦，双曲線正弦，双曲線正接，双曲線余接，双曲線余弦の逆関数，双曲線正弦の逆関数，双曲線正接の逆関数が IEEE 754 の丸め規格に準拠していることを示唆している．

4.3　多倍長精度の機械区間演算と相対誤差（複素数）

実験目的　本節では，複素数の四則演算を行ったとき，どれほど区間幅が広がるか確認する．その結果によって，点区間で受け取ったとき，実数と同様に複素数も 1ulp で抑えるためにどの程度ビット数を拡張すればいいか把握することができる．

実験方法　実験に使用したソースコードは付録 C.3，pp.111–112 を参照．各四則演算に応じて，mic_micmic_add 関数の部分を適切な関数に置き換えて演算を行った．

実験結果　複素数の四則演算による相対誤差の表とグラフは，付録 C.4.1〜C.4.4，pp.113–120 を参照．

考察　図 32，33 から，多倍長精度の複素数の加減算において，実部と虚部の両方で，ほぼすべての精度に対して相対誤差が 1 であることがわかる．これは，オーダーが異なる数値の加減算が行われることで有効桁数が増加し，結果として情報損失が発生するためであると推測される．また，表 38，39 から，どの入力値でも，相対誤差の最大値は 1 以下であることがわかる．この結果から，実部と虚部ともに，下限と上限の差が 1ulp に収まっており，演算後の下向き丸めと上向き丸めがそれぞれ，数学的に正しい値から最も近い浮動小数点数に丸められているといえる．つまり，多倍長精度の複素数の加減算は IEEE 754 の丸め規格に準拠した動作をしていることを示唆している．また，同じオーダーで同符号の数値を減算した場合に，相対誤差の最大値が 0 になることがわかる．これは，減算後に有効桁数が増加せず，情報の損失が生じないことを意味している．

図 34 から，多倍長精度の複素数の乗算において，実部と虚部ともに，ほぼすべての精度で相対誤差が 1 を超えていることがわかる．これは，複数回の機械区間演算により区間幅が広がるためであると考えられる．また，表 40 から，どの入力値でも，相対誤差の最大値は 1 以上であることがわかる．これは，多倍長精度の複素数の乗算は IEEE 754 の丸め規格に準拠した動作をしていないことを示唆している．相対誤差が 2 や 3 になる事象が頻出するのは，複素数の乗算が合計で 2 回の演算を伴うことに起因すると推測される．

図 35 から，多倍長精度の複素数の除算において，実部と虚部ともに，ほぼすべての精度で相対誤差が 1 を超えていることがわかる．これは，複数回の機械区間演算により区間幅が広がるためであると考えられる．さらに，実部では，相対誤差が 3 と 4 の間で，虚部では，2 と 3 の間で振動する現象が観察された．これは，計算途中でオーダーが同じ加減算があり，精度によっては，有効桁数が増加する場合とそうでない場合があるからだと考えられる．また，表 41 から，どの入力値でも，相対誤差の最大値は 1 以上であることがわかる．これは，多倍長精度の複素数の徐算は IEEE 754 の丸め規格に準拠した動作をしていないことが示唆される．相対誤差が 3 や 4 になる事象が頻出するのは，複素数の乗算が合計で 3 回の演算を伴うことに起因すると推測される．

表 40，41 から，一部の入力値の組み合わせにおいて相対誤差が顕著に大きくことがわかる．この現象の原因を分析した結果，up と down の差に顕著な特徴は見られなかった．しかし，計算結果の指数部を解析した際，通常見られる規則性が失われていることが判明した．通常，精度が増加しても指数部の値は一定を保つが，数値のオーダーがほぼ同等の場合には指数部の値が突然下がる現象が確認された．これは，オーダーが同等の加減算を行う際に桁落ち誤差が生じることが原因であると推測される．この分析から，プログラム設計時には桁落ち誤差の可能性も考慮する必要があることが明らかになった．

4.4 改良後の機械区間演算と相対誤差（複素数）

実験目的 第 4.3 節の結果を受けて，各値がすべて 1.0 以内になるように，第 3.4 節で示した関数内の精度の調整を行う．

実験方法 実験に使用したソースコードは付録 C.3，pp.111–112 を参照．ただし，mic_micmic_mul の部分を, 付録 A.3.6 に記載してある mic_micmic_mul_plusnbits や mic_micmic_div_timesnbits に変えて，数値実験を行う．関数内の精度は元の精度に加えて，1，2，4，. . . ，128bits と増やしたり，元の精度の 2 倍，3 倍にして数値実験を行った．

実験結果 改良後の複素数の乗算と除算による相対誤差の表とグラフは，それぞれ付録 C.4.5 と C.4.6，pp.121–128 を参照．

考察 改良後の複素数の乗算について，表 42〜49 から，ビット数を 16 以上増やすことで桁落ち誤差が生じない部分が 1ulp 以内に収まることが確認された．また，表 50，51 から，ビット数を 2 倍以上にすることで，桁落ち誤差が生じる部分も 1ulp 以内に改善されることがわかる．しかし，ビット数を 2 倍にしても精度が低い場合，1ulp 以内に収まらない可能性があるため，ビット数を 2 倍にし，さらに 16 ビット増やすことで，低精度でも，桁落ち誤差が生じる部分でも，1ulp 以内に収めることが可能であると推測される．この仮説を検証した結果，図 36 と表 52 に示される通り，どの入力値でも，相対誤差の最大値が 1 以下であることが明らかになった．これは，実部と虚部ともに，下限と上限の差が 1ulp に収まり，演算後の下向き丸めと上向き丸めがそれぞれ，数学的に正確な値に最も近い浮動小数点数に正しく丸められていることを示している．したがって，改良されたプログラムによる多倍長精度の複素数乗算が IEEE 754 の丸め規格に準拠していることが示唆される．これにより，複素数の乗算に関する実験目的は達成されたと結論づけられる．

改良後の複素数の除算について，表 53〜60 から，ビット数を 16 以上増やすことで桁落ち誤差が生じない部分が 1ulp 以内に収まることが確認された．また，表 61，62 から，ビット数を 2 倍以上にすることで，桁落ち誤差が生じる部分も 1ulp 以内に改善されることがわかる．しかし，ビット数を 2 倍にしても精度が低い場合，1ulp 以内に収まらない可

能性があるため，ビット数を 2 倍にし，さらに 16 ビット増やすことで，低精度でも，桁落ち誤差が生じる部分でも，1ulp 以内に収めることができると考えられる．この仮説を検証したところ，図 37 と表 63 から，どの入力値でも，1ulp 以内に収まることを確認し，複素数の除算に関する実験目的も達成された．

5　まとめ

本研究では，IEEE 754 の丸め規格に MPFR ライブラリがどの程度準拠しているかを数値実験により検証した．その結果，MPFR ライブラリの改善の必要はないことが明らかになった．この発見に基づき，我々は複素数演算における新たな課題に取り組んだ．IEEE 754 の丸め規格が存在しない複素数演算に対し，実数の四則演算と同様に，上向き丸めと下向き丸めの差が 1ulp 以内に収まるような方法を開発した．これは，コンピュータでの演算において，丸めの規則が存在することで，演算結果が数学的に正しい値からどの程度離れているかを把握できる点において重要であるといえる．

研究目的を達成するために実装した機械区間演算は，値を区間として扱うことで，複数回の四則演算を行う際に，演算の途中で丸め誤差が生じても，数学的に正しい値として保証されるというメリットがある．大きな桁数や無理数が初期値である演算であっても，入力値の上向き丸めと下向き丸めをそれぞれの上限と下限にセットし，機械区間演算で演算することによって，数学的に正しい値が下限と上限に挟まれた区間として返ってくる．このアプローチは，入力時に生じる丸め誤差に対しても有効であるので，初期値を含めた，最初から最後まで，演算の正確性が保証される．機械区間演算のプログラムは計算結果の正確性を保証する上で極めて有用であり，特に科学計算や工学分野において，精度の高い計算が求められる状況において，その応用範囲の拡大が期待される．

今回の研究では，すべての変数の精度を均一に設定し，ゼロに対する分岐処理を詳細に考慮せず，特定の入力値を用いて，1000 以下の精度で数値実験を実施した．今後の課題としては，精度が異なる演算の扱い，ゼロの扱い方，さらに入力値と精度の範囲を広げた議論をすることである．他の課題としては，四則演算以外の計算手法である機械区間演算の実装，複素数に対する新たな演算法の開発が挙げられる．また，この研究の成果をより広く利用しやすくするため，現在 C 言語で実装されているプログラムを MATLAB など他のプログラミング言語に組み込むことである．これにより，より多くの研究分野や応用分野で精度の高い計算手法が普及し，その利益を享受することが期待される．

謝辞

本研究の進行に際し，近藤弘一教授と田中智之助教の熱心な指導に感謝いたします．また，充実した環境と指導を提供してくださった応用数学研究室の皆様に御礼申し上げます．

参考文献

[1] 大石進一，精度保証付き数値計算，コロナ社，2000，1–28.

A　本研究に関する理論や資料の補足（付録）

A.1　精度保証付き数値計算（追加）

A.1.1　2 進規格化浮動小数点数の具体例

第 2.2.2. 節で 2 進規格化浮動小数点数は一般に式 (2.1) のようにと表されると説明した．具体例として，たとえば，10 進数で -27.625 という数値は

$$\begin{aligned} -27.625 &= -\left(2^4 + 2^3 + 2^1 + 2^0 + 2^{-1} + 2^{-3}\right) \\ &= -\left(2^{-1} + 2^{-2} + 2^{-4} + 2^{-5} + 2^{-6} + 2^{-8}\right) \times 2^5 \end{aligned} \tag{A.1}$$

と 2 進規格化浮動小数点数で表せる．

A.1.2　零

零は

$$+\left(\frac{0}{2} + \frac{0}{2^2} + \frac{0}{2^3} + \cdots + \frac{0}{2^N}\right) \times 2^{e_{\min}} \tag{A.2}$$

と表される．

A.1.3　非規格化 2 進浮動小数点数

非規格化 2 進浮動小数点数は指数部が全て 0 であり，かつ仮数部が 0 出ない時に使用される．たとえば，単精度浮動小数点数において，2 進数で 00000000000100000000000000000000 を考えたとき，符号部は 0, 指数部は 00000000, 仮数部は 00100000000000000000000 である．この場合，符号は正，指数部は 2^{-126}, 仮数部は 2 進数で 0.00100000000000000000000 すなわち，2^{-3} なので $2^{-126} \times 2^{-3} = +2^{-129}$ である．

A.1.4　NaN

NaN は $\sqrt{7}$ や，∞/∞, $\infty + \infty$ などの演算によって得られる，特殊な値であり，正確な値を持たない．$\pm\infty$ はオーバーフローの結果，または値を零で割ったときに得られる．± 0 はアンダーフロー，または値を $\pm\infty$ で割ったときに得られる．NaN を除いた数の集合，すなわち，規格化浮動小数点数，零，非規格化 2 浮動小数点数の集合を $\mathbf{F}$ とする．実数 x が与えられたとき，x を挟む集合 $\mathbf{F}$ に含まれる二つの浮動小数点数の距離を $\mathrm{ulp}(x)$ で表す．

A.2　MPFR(追加)

A.2.1　ヘッダファイル（具体例）

```
// MPFR ライブラリのインクルードの例
#include<stdio.h>
#include<mpfr.h>

int main(void){

}
```

A.2.2　初期化関数（追加）

- **mpfr_get_default_prec**：現在設定されている MPFR のデフォルトの精度桁数（ビット数）を返す.
- **mpfr_set_prec**：指定された変数の精度桁数を変更し，その変数に NaN を代入する．引数には，変数と新しい精度桁数（ビット数）を指定する．この関数は，変数に格納されている値をクリアし，新しい精度で再初期化する．ただし，新しい精度桁数が既存のメモリ領域に収まる場合は，メモリ領域の再確保は行わない.

A.2.3　初等関数（追加）

A.3　研究で使用したヘッダファイルやライブラリの説明 (C 言語)

本研究で使用したプログラミング言語は C 言語，C++，MATLAB であるが，この節では，C 言語に関するヘッダファイルとライブラリの説明をする.

A.3.1　stdio.h

`#include<stdio.h>` は標準入出力ライブラリに関するヘッダファイルである.

A.3.2　stdlib.h

`#include<stdlib.h>` は，一般的な実用性を持つ 5 つの型といくつかの関数を宣言し，いくつかのマクロを定義している．今回の研究で重要な関数は疑似乱数列生成関数と記憶域管理関数である．疑似乱数列生成関数の中には rand 関数というものがあり，記憶域管理関数の中には free 関数と malloc 関数がある.

- rand 関数：0 以上 RAND_MAX 以下の範囲の疑似乱数整数列を計算する関数である．
- free 関数：free 関数は，ポインタが指す領域を解放し，その後の割り付けに使用できるようにする関数である．
- malloc 関数：size で指定されたサイズのオブジェクトのための領域を確保する．

以下に，stdlib.h ヘッダファイルで提供されるいくつかの関数の例を示す．

```
#include <stdlib.h>

int rand(void);
void free(void *ptr);
void *malloc(size_t size);
```

A.3.3 fenv.h

#include<fenv.h>は，C 言語における浮動小数点の環境操作に関連するヘッダファイルである．このヘッダファイルは，浮動小数点の例外や丸めモードを扱うための関数やマクロを提供する．浮動小数点の例外とは，計算中に発生する特殊な状況のことで，たとえば，オーバーフローやアンダーフロー，不正確な計算などが該当する．このヘッダファイルには，これらの例外を検出し，適切に対処するための機能が含まれている．また，丸めモードとは，浮動小数点数の計算結果を特定の方法で丸める設定のことである．たとえば，最も近い値に丸める，ゼロに向かって丸めるなどのモードがある．

- feclearexcept(int excepts)：指定された浮動小数点の例外フラグをクリアする．
- feraiseexcept(int excepts)：指定された浮動小数点の例外を発生させる．
- fegetround(void)：現在の丸めモードを取得する．
- fesetround(int mode)：丸めモードを設定する．

これらの関数を使用することで，プログラムは浮動小数点の計算をより厳密に制御し，特定の計算要件に合わせることができる．

A.3.4 math.h

#include<math.h>は数学関数と浮動小数点計算に関する多くの関数を提供するヘッダファイルである．このヘッダファイルは，三角関数，指数関数，対数関数，平方根関数など，様々な数学的計算を行うための関数を含んでいる．これにより，これらの計算を直接実装することなく，効率的かつ簡単に使用することができる．

- sin, cos, tan：三角関数で，引数としてラジアン単位の角度を取り，その三角関数の値を返す．
- exp：引数として与えられた数値の自然対数の底 (約 2.71828) を底とする指数関数の値を計算する．
- log：引数として与えられた数値の自然対数を計算する．
- sqrt：引数として与えられた数値の平方根を計算する．

以下に，math.h ヘッダファイルで提供されるいくつかの関数の例を示す．

```
#include <math.h>

double sin(double x);
double cos(double x);
double tan(double x);
double exp(double x);
double log(double x);
double sqrt(double x);
```

これらの関数を使用することで，数学的な計算を簡単に行うことができ，プログラムの実装が容易になる．特に，科学技術計算を行う際には，これらの関数が非常に有効である．

A.3.5 time.h

#include<time.h>は時間に関する様々な機能を提供するヘッダファイルである．このファイルには，時間の計測，変換，操作に関する関数や型定義が含まれている．以下に主要な機能をいくつか挙げる．

- time_t 型：この型は，システムのエポック（通常は 1970 年 1 月 1 日午前 0 時 UTC）からの経過秒数を表す．
- time 関数：現在の時間を time_t 型で返す．
- gmtime 関数：time_t 型の値を UTC に基づいた tm 構造体に変換する．
- localtime 関数：time_t 型の値をローカルタイムゾーンに基づいた tm 構造体に変換する．
- mktime 関数：tm 構造体の値を time_t 型に変換する．
- strftime 関数：tm 構造体に基づいて，日付や時間をフォーマットする．

以下に，time.h ヘッダファイルで提供される関数の例を示す.

```
#include <time.h>

time_t time(time_t *t);
struct tm *gmtime(const time_t *timep);
struct tm *localtime(const time_t *timep);
time_t mktime(struct tm *tm);
size_t strftime(char *s, size_t max,
const char *format, const struct tm *tm);
```

A.3.6 mi.c（ソースコード）

```
// 実数に関する構造体や関数

// rmulti 型の下限と上限のポインタを格納する，実数の区間を表す構造体を定義
typedef struct {
    rmulti* down; // 下限を表す rmulti 型のポインタ
    rmulti* up; // 上限を表す rmulti 型のポインタ
} mi_r;

// mi_r 型のポインタを動的に確保し，初期化する関数
mi_r* mi_rallocate() {
    mi_r* mir = (mi_r*)malloc(sizeof(mi_r)); // mi_r 型のメモリを確保
    mir->down = rallocate(); // 下限のメモリを確保し，初期化
    mir->up = rallocate(); // 上限のメモリを確保し，初期化
    return mir; // 確保した mi_r 型のポインタを返す
}

// mi_r 型のメモリを解放する関数
mi_r* mi_rfree(mi_r* x) {
    x->down = rmfree(x->down); // 下限のメモリを解放
    x->up = rmfree(x->up); // 上限のメモリを解放
    return NULL; // NULL を返してポインタを無効化
}

// mi_r 型の値を受け取り，mi_r 型の変数に格納する関数
void mir_mir_set(mi_r* z, mi_r* x) {
```

```
    mpfr_set(z->down, x->down, MPFR_RNDD); // x の下限を下向きに丸め, z の下限に格納
    mpfr_set(z->up, x->up, MPFR_RNDU); // x の上限を上向きに丸め, z の上限に格納
}

// 10 進数の文字列を受け取り，mi_r 型の変数に 10 進数の値として格納する関数
void mir_str_set(mi_r* z, char* x) {
    mpfr_set_str(z->down, x, 10, MPFR_RNDD); // 文字列 x を下向きに丸め, 10 進数の値として z の下限に格納
    mpfr_set_str(z->up, x, 10, MPFR_RNDU); // 文字列 x を上向きに丸め, 10 進数の値として z の上限に格納
}

// 10 進数の文字列を受け取り，mi_r 型の変数に点区間として格納する関数
void mir_str_set_dot(mi_r* z, char* x) {
    mpfr_set_str(z->down, x, 10, MPFR_RNDN); // 文字列 x を最近点に丸め, 10 進数の値として z の下限に格納
    mpfr_set_str(z->up, x, 10, MPFR_RNDN); // 文字列 x を最近点に丸め, 10 進数の値として z の上限に格納
}

// 2 つの実数の点区間を受け取り，加算して実数の区間で返す関数
void mir_rr_add(mi_r* z, rmulti* x, rmulti* y) {
    mpfr_add(z->down, x, y, MPFR_RNDD); // x と y を加算し，下向きに丸め，z の下限に格納
    mpfr_add(z->up, x, y, MPFR_RNDU); // x と y を加算し，上向きに丸め，z の上限に格納
}

// 2 つの実数の点区間を受け取り，減算して実数の区間で返す関数
void mir_rr_sub(mi_r* z, rmulti* x, rmulti* y) {
    mpfr_sub(z->down, x, y, MPFR_RNDD); // x から y を減算し，下向きに丸め，z の下限に格納
    mpfr_sub(z->up, x, y, MPFR_RNDU); // x から y を減算し，上向きに丸め，z の上限に格納
}
```

```
// 2つの実数の点区間を受け取り，乗算して実数の区間で返す関数
void mir_rr_mul(mi_r* z, rmulti* x, rmulti* y) {
    mpfr_mul(z->down, x, y, MPFR_RNDD); // x と y を乗算し，下向きに丸め，z の下限に格納
    mpfr_mul(z->up, x, y, MPFR_RNDU); // x と y を乗算し，上向きに丸め，z の上限に格納
}

// 2つの実数の点区間を受け取り，除算して実数の区間で返す関数
void mir_rr_div(mi_r* z, rmulti* x, rmulti* y) {
    mpfr_div(z->down, x, y, MPFR_RNDD); // x を y で除算し，下向きに丸め，z の下限に格納
    mpfr_div(z->up, x, y, MPFR_RNDU); // x を y で除算し，上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，平方根を実数の区間で返す関数
void mir_r_sqrt(mi_r* z, rmulti* x) {
    mpfr_sqrt(z->down, x, MPFR_RNDD); // x の平方根を下向きに丸め，z の下限に格納
    mpfr_sqrt(z->up, x, MPFR_RNDU); // x の平方根を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，逆平方根を実数の区間で返す関数
void mir_r_rec(mi_r* z, rmulti* x) {
    mpfr_rec_sqrt(z->down, x, MPFR_RNDD); // x の逆平方根を下向きに丸め，z の下限に格納
    mpfr_rec_sqrt(z->up, x, MPFR_RNDU); // x の逆平方根を上向きに丸め，z の上限に格納
}

// 2つの実数の点区間を受け取り，べき乗を実数の区間で返す関数
void mir_rr_pow(mi_r* z, rmulti* x, rmulti* y) {
    mpfr_pow(z->down, x, y, MPFR_RNDD); // x の y 乗を下向きに丸め，z の下限に格納
```

```
    mpfr_pow(z->up, x, y, MPFR_RNDU); // x の y 乗を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，自然対数を実数の区間で返す関数
void mir_r_log(mi_r* z, rmulti* x) {
    mpfr_log(z->down, x, MPFR_RNDD); // x の自然対数を下向きに丸め，z の下限に格納
    mpfr_log(z->up, x, MPFR_RNDU); // x の自然対数を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，2 を底とする対数を実数の区間で返す関数
void mir_r_log2(mi_r* z, rmulti* x) {
    mpfr_log2(z->down, x, MPFR_RNDD); // x の 2 を底とする対数を下向きに丸め，z の下限に格納
    mpfr_log2(z->up, x, MPFR_RNDU); // x の 2 を底とする対数を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，10 を底とする対数を実数の区間で返す関数
void mir_r_log10(mi_r* z, rmulti* x) {
    mpfr_log10(z->down, x, MPFR_RNDD); // x の 10 を底とする対数を下向きに丸め，z の下限に格納
    mpfr_log10(z->up, x, MPFR_RNDU); // x の 10 を底とする対数を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，e のべき乗を実数の区間で返す関数
void mir_r_exp(mi_r* z, rmulti* x) {
    mpfr_exp(z->down, x, MPFR_RNDD); // x の e のべき乗を下向きに丸め，z の下限に格納
    mpfr_exp(z->up, x, MPFR_RNDU); // x の e のべき乗を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，2 のべき乗を実数の区間で返す関数
```

```
void mir_r_exp2(mi_r* z, rmulti* x) {
    mpfr_exp2(z->down, x, MPFR_RNDD); // x の 2 のべき乗を下向きに丸め，z の下限に格納
    mpfr_exp2(z->up, x, MPFR_RNDU); // x の 2 のべき乗を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，2 のべき乗を実数の区間で返す関数
void mir_r_exp10(mi_r* z, rmulti* x) {
    mpfr_exp10(z->down, x, MPFR_RNDD); // x の 10 のべき乗を下向きに丸め，z の下限に格納
    mpfr_exp10(z->up, x, MPFR_RNDU); // x の 10 のべき乗を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，余弦を実数の区間で返す関数
void mir_r_cos(mi_r* z, rmulti* x) {
    mpfr_cos(z->down, x, MPFR_RNDD); // x の余弦を下向きに丸め，z の下限に格納
    mpfr_cos(z->up, x, MPFR_RNDU); // x の余弦を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，正弦を実数の区間で返す関数
void mir_r_sin(mi_r* z, rmulti* x) {
    mpfr_sin(z->down, x, MPFR_RNDD); // x の正弦を下向きに丸め，z の下限に格納
    mpfr_sin(z->up, x, MPFR_RNDU); // x の正弦を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，正接を実数の区間で返す関数
void mir_r_tan(mi_r* z, rmulti* x) {
    mpfr_tan(z->down, x, MPFR_RNDD); // x の正接を下向きに丸め，z の下限に格納
    mpfr_tan(z->up, x, MPFR_RNDU);   // x の正接を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，正割（secant）を実数の区間で返す関数
void mir_r_sec(mi_r* z, rmulti* x) {
    mpfr_sec(z->down, x, MPFR_RNDD); // x の正割を下向きに丸め，z の下限に格納
    mpfr_sec(z->up, x, MPFR_RNDU); // x の正割を上向きに丸め，z の上限に格納
```

```
}

// 実数の点区間を受け取り，余割（cosecant）を実数の区間で返す関数
void mir_r_csc(mi_r* z, rmulti* x) {
    mpfr_csc(z->down, x, MPFR_RNDD); // x の余割を下向きに丸め，z の下限に格納
    mpfr_csc(z->up, x, MPFR_RNDU); // x の余割を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，余接（cotangent）を実数の区間で返す関数
void mir_r_cot(mi_r* z, rmulti* x) {
    mpfr_cot(z->down, x, MPFR_RNDD); // x の余接を下向きに丸め，z の下限に格納
    mpfr_cot(z->up, x, MPFR_RNDU); // x の余接を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，逆余弦（arccosine）を実数の区間で返す関数
void mir_r_acos(mi_r* z, rmulti* x) {
    mpfr_acos(z->down, x, MPFR_RNDD); // x の逆余弦を下向きに丸め，z の下限に
格納
    mpfr_acos(z->up, x, MPFR_RNDU); // x の逆余弦を上向きに丸め，z の上限に格
納
}

// 実数の点区間を受け取り，逆正弦（arcsine）を実数の区間で返す関数
void mir_r_asin(mi_r* z, rmulti* x) {
    mpfr_asin(z->down, x, MPFR_RNDD); // x の逆正弦を下向きに丸め，z の下限に
格納
    mpfr_asin(z->up, x, MPFR_RNDU); // x の逆正弦を上向きに丸め，z の上限に格
納
}

// 実数の点区間を受け取り，逆正接（arctangent）を実数の区間で返す関数
void mir_r_atan(mi_r* z, rmulti* x) {
    mpfr_atan(z->down, x, MPFR_RNDD); // x の逆正接を下向きに丸め，z の下限に
格納
    mpfr_atan(z->up, x, MPFR_RNDU); // x の逆正接を上向きに丸め，z の上限に格
納
}
```

```
// 実数の点区間を受け取り，双曲線余弦（hyperbolic cosine）を実数の区間で返す関数
void mir_r_cosh(mi_r* z, rmulti* x) {
    mpfr_cosh(z->down, x, MPFR_RNDD); // x の双曲線余弦を下向きに丸め，z の下限に格納
    mpfr_cosh(z->up, x, MPFR_RNDU); // x の双曲線余弦を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，双曲線正弦（hyperbolic sine）を実数の区間で返す関数
void mir_r_sinh(mi_r* z, rmulti* x) {
    mpfr_sinh(z->down, x, MPFR_RNDD); // x の双曲線正弦を下向きに丸め，z の下限に格納
    mpfr_sinh(z->up, x, MPFR_RNDU); // x の双曲線正弦を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，双曲線正接（hyperbolic tangent）を実数の区間で返す関数
void mir_r_tanh(mi_r* z, rmulti* x) {
    mpfr_tanh(z->down, x, MPFR_RNDD); // x の双曲線正接を下向きに丸め，z の下限に格納
    mpfr_tanh(z->up, x, MPFR_RNDU); // x の双曲線正接を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，双曲線正割（hyperbolic secant）を実数の区間で返す関数
void mir_r_sech(mi_r* z, rmulti* x) {
    mpfr_sech(z->down, x, MPFR_RNDD); // x の双曲線正割を下向きに丸め，z の下限に格納
    mpfr_sech(z->up, x, MPFR_RNDU); // x の双曲線正割を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，双曲線余割（hyperbolic cosecant）を実数の区間で返す
```

関数
```
void mir_r_csch(mi_r* z, rmulti* x) {
    mpfr_csch(z->down, x, MPFR_RNDD); // x の双曲線余割を下向きに丸め，z の下限に格納
    mpfr_csch(z->up, x, MPFR_RNDU); // x の双曲線余割を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，双曲線余接（hyperbolic cotangent）を実数の区間で返す関数
void mir_r_coth(mi_r* z, rmulti* x) {
    mpfr_coth(z->down, x, MPFR_RNDD); // x の双曲線余接を下向きに丸め，z の下限に格納
    mpfr_coth(z->up, x, MPFR_RNDU); // x の双曲線余接を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，逆双曲線余弦（hyperbolic arccosine）を実数の区間で返す関数
void mir_r_acosh(mi_r* z, rmulti* x) {
    mpfr_acosh(z->down, x, MPFR_RNDD); // x の逆双曲線余弦を下向きに丸め，z の下限に格納
    mpfr_acosh(z->up, x, MPFR_RNDU); // x の逆双曲線余弦を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，逆双曲線正弦（hyperbolic arcsine）を実数の区間で返す関数
void mir_r_asinh(mi_r* z, rmulti* x) {
    mpfr_asinh(z->down, x, MPFR_RNDD); // x の逆双曲線正弦を下向きに丸め，z の下限に格納
    mpfr_asinh(z->up, x, MPFR_RNDU); // x の逆双曲線正弦を上向きに丸め，z の上限に格納
}

// 実数の点区間を受け取り，逆双曲線接線（hyperbolic arctangent）を実数の区間で返す関数
```

```
void mir_r_atanh(mi_r* z, rmulti* x) {
    mpfr_atanh(z->down, x, MPFR_RNDD); // x の逆双曲線接線を下向きに丸め, z の下限に格納
    mpfr_atanh(z->up, x, MPFR_RNDU); // x の逆双曲線接線を上向きに丸め, z の上限に格納
}

// 2 つの実数の区間を受け取り, 加算して実数の区間で返す関数
void mir_mirmir_add(mi_r* z, mi_r* x, mi_r* y) {
    mpfr_add(z->down, x->down, y->down, MPFR_RNDD); // x の下限と y の下限を加算し, 下向きに丸め, z の下限に格納
    mpfr_add(z->up, x->up, y->up, MPFR_RNDU); // x の上限と y の上限を加算し, 上向きに丸め, z の上限に格納
}

// 2 つの実数の区間を受け取り, 減算して実数の区間で返す関数
void mir_mirmir_sub(mi_r* z, mi_r* x, mi_r* y) {
    mpfr_sub(z->down, x->down, y->up, MPFR_RNDD); // x の下限から y の上限を減算し, 下向きに丸め, z の下限に格納
    mpfr_sub(z->up, x->up, y->down, MPFR_RNDU); // x の上限から y の下限を減算し, 上向きに丸め, z の上限に格納
}

// 2 つの実数の区間を受け取り, 乗算して実数の区間で返す関数
void mir_mirmir_mul(mi_r* z, mi_r* x, mi_r* y) {
    // 関数内で使用するポインタ変数の宣言
    rmulti *z_min, *z_max, *a;

    // rmulti 型のメモリを確保し, 初期化
    z_min = rallocate(); // 一時的な最小値用のメモリを確保し, 初期化
    z_max = rallocate(); // 一時的な最大値用のメモリを確保し, 初期化
    a = rallocate(); // 一時的な値を保持するメモリを確保し, 初期化

    mpfr_mul(z_min, x->down, y->down, MPFR_RNDD); // x の下限と y の下限を乗算し, 下向きに丸めて一時的な最小値に格納
    mpfr_mul(a, x->down, y->up, MPFR_RNDD); // x の下限と y の上限を乗算し, 下向きに丸めて一時的な変数 a に格納
```

```
    if(mpfr_cmp(a, z_min) < 0) {
        mpfr_set(z_min, a, MPFR_RNDD);
    } // 最小値を更新
    mpfr_mul(a, x->up, y->down, MPFR_RNDD); // x の上限と y の下限を乗算し，下向きに丸めて一時的な変数 a に格納
    if(mpfr_cmp(a, z_min) < 0) {
        mpfr_set(z_min, a, MPFR_RNDD);
    } // 最小値を更新
    mpfr_mul(a, x->up, y->up, MPFR_RNDD); // x の上限と y の上限を乗算し，下向きに丸めて一時的な変数 a に格納
    if(mpfr_cmp(a, z_min) < 0) {
        mpfr_set(z_min, a, MPFR_RNDD);
    } // 最小値を更新
    mpfr_set(z->down, z_min, MPFR_RNDD); // 最終的な下限を z の下限に格納

    // 同様の操作を上向きに丸めて行い，最終的な上限を z の上限に格納
    mpfr_mul(z_max, x->down, y->down, MPFR_RNDU);
    mpfr_mul(a, x->down, y->up, MPFR_RNDU);
    if(mpfr_cmp(a, z_max) > 0) {
        mpfr_set(z_max, a, MPFR_RNDU);
    }
    mpfr_mul(a, x->up, y->down, MPFR_RNDU);
    if(mpfr_cmp(a, z_max) > 0) {
        mpfr_set(z_max, a, MPFR_RNDU);
    }
    mpfr_mul(a, x->up, y->up, MPFR_RNDU);
    if(mpfr_cmp(a, z_max) < 0) {
        mpfr_set(z_max, a, MPFR_RNDU);
    }
    mpfr_set(z->up, z_max, MPFR_RNDU);

    // 使用した一時的なメモリを解放
    z_min = rmfree(z_min);
    z_max = rmfree(z_max);
    a = rmfree(a);
}
```

```
// 2つの実数の区間を受け取り，除算して実数の区間で返す関数
void mir_mirmir_div(mi_r* z, mi_r* x, mi_r* y) {
    // 関数内で使用するポインタ変数の宣言
    rmulti *z_min, *z_max, *a;

    // rmulti 型のメモリを確保し，初期化
    z_min = rallocate(); // 一時的な最小値用のメモリを確保し，初期化
    z_max = rallocate(); // 一時的な最大値用のメモリを確保し，初期化
    a = rallocate(); // 一時的な値を保持するメモリを確保し，初期化

    mpfr_div(z_min, x->down, y->down, MPFR_RNDD); // x の下限を y の下限で除算し，下向きに丸めて一時的な最小値に格納
    mpfr_div(a, x->down, y->up, MPFR_RNDD); // x の下限を y の上限で除算し，下向きに丸めて一時的な変数 a に格納
    if(mpfr_cmp(a, z_min) < 0) {
        mpfr_set(z_min, a, MPFR_RNDD);
    } // 最小値を更新
    mpfr_div(a, x->up, y->down, MPFR_RNDD); // x の上限を y の下限で除算し，下向きに丸めて一時的な変数 a に格納
    if(mpfr_cmp(a, z_min) < 0) {
        mpfr_set(z_min, a, MPFR_RNDD);
    } // 最小値を更新
    mpfr_div(a, x->up, y->up, MPFR_RNDD); // x の上限を y の上限で除算し，下向きに丸めて一時的な変数 a に格納
    if(mpfr_cmp(a, z_min) < 0) {
        mpfr_set(z_min, a, MPFR_RNDD);
    } // 最小値を更新
    mpfr_set(z->down, z_min, MPFR_RNDD); // 最終的な下限を z の下限に格納

    // 同様の操作を上向きに丸めて行い，最終的な上限を z の上限に格納
    mpfr_div(z_max, x->down, y->down, MPFR_RNDU);
    mpfr_div(a, x->down, y->up, MPFR_RNDU);
    if(mpfr_cmp(a, z_max) > 0) {
        mpfr_set(z_max, a, MPFR_RNDU);
    }
    mpfr_div(a, x->up, y->down, MPFR_RNDU);
    if(mpfr_cmp(a, z_max) > 0) {
```

```
        mpfr_set(z_max, a, MPFR_RNDU);
    }
    mpfr_div(a, x->up, y->up, MPFR_RNDU);
    if(mpfr_cmp(a, z_max) < 0) {
        mpfr_set(z_max, a, MPFR_RNDU);
    }
    mpfr_set(z->up, z_max, MPFR_RNDU);

    // 使用した一時的なメモリを解放
    z_min = rmfree(z_min);
    z_max = rmfree(z_max);
    a = rmfree(a);
}

// 複素数に関する構造体や関数

// mi_r 型の実部と虚部のポインタを格納する，複素数の区間を表す構造体を定義
typedef struct {
    mi_r* r; // 実部を表す mi_r 型のポインタ
    mi_r* i; // 虚部を表す mi_r 型のポインタ
} mi_c;

// mi_c 型のメモリを解放する関数
mi_c* mi_cfree(mi_c* x) {
    x->r = mi_rfree(x->r); // 実部のメモリを解放
    x->i = mi_rfree(x->i); // 虚部のメモリを解放
    return NULL; // NULL を返してポインタを無効化
}

// mi_c 型のポインタを動的に確保し，初期化する関数
mi_c* mi_callocate() {
    mi_c* mic = (mi_c*)malloc(sizeof(mi_c)); // mi_c 型のメモリを確保
    mic->r = mi_rallocate(); // 実部のメモリを確保し，初期化
    mic->i = mi_rallocate(); // 虚部のメモリを確保し．初期化
    return mic; // 確保した mi_c 型のポインタを返す
}
```

```
// mi_c 型の値を受け取り，mi_c 型の変数に格納する関数
void mic_mic_set(mi_c* z, mi_c* x) {
    mir_mir_set(z->r, x->r); // x の実部を z の実部に格納
    mir_mir_set(z->i, x->i); // x の虚部を z の虚部に格納
}

// 2 つの 10 進数の文字列を受け取り，mi_c 型の変数に 10 進数の値として格納する関数
void mic_strstr_set(mi_c* z, char* x, char* y) {
    mir_str_set(z->r, x); // 文字列 x を 10 進数の値として z の実部に格納
    mir_str_set(z->i, y); // 文字列 y を 10 進数の値として z の虚部に格納
}

// 2 つの 10 進数の文字列を受け取り，mi_c 型の変数に点区間として格納する関数
void mic_strstr_set_dot(mi_c* z, char* x, char* y) {
    mir_str_set_dot(z->r, x); // 文字列 x を最近点に丸め，10 進数の値として z の
実部に格納
    mir_str_set_dot(z->i, y); // 文字列 y を最近点に丸め，10 進数の値として z の
虚部に格納
}

// 2 つの複素数の区間を受け取り，加算して複素数の区間で返す関数
void mic_micmic_add(mi_c* z, mi_c* x, mi_c* y) {
    mir_mirmir_add(z->r, x->r, y->r); // x の実部の区間と y の実部の区間を加算
し，z の実部の区間に格納
    mir_mirmir_add(z->i, x->i, y->i); // x の虚部の区間と y の虚部の区間を加算
し，z の虚部の区間に格納
}

// 2 つの複素数の区間を受け取り，減算して複素数の区間で返す関数
void mic_micmic_sub(mi_c* z, mi_c* x, mi_c* y) {
    mir_mirmir_sub(z->r, x->r, y->r); // x の実部の区間から y の実部の区間を減
算し，z の実部の区間に格納
    mir_mirmir_sub(z->i, x->i, y->i); // x の虚部の区間から y の虚部の区間を減
算し，z の虚部の区間に格納
}

// 2 つの複素数の区間を受け取り，乗算して複素数の区間で返す関数
```

```
void mic_micmic_mul(mi_c* z, mi_c* x, mi_c* y) {
    // 関数内で使用するポインタ変数の宣言
    mi_r *a, *b;

    // mi_r 型のメモリを確保し，初期化
    a = mi_rallocate(); // a のメモリを確保し，初期化
    b = mi_rallocate(); // b のメモリを確保し，初期化

    // 複素数の乗算の実部計算
    mir_mirmir_mul(a, x->r, y->r); // x の実部の区間と y の実部の区間を乗算し，
a に格納
    mir_mirmir_mul(b, x->i, y->i); // x の虚部の区間と y の虚部の区間を乗算し，
b に格納
    mir_mirmir_sub(z->r, a, b); // a から b を減算し，z の実部に格納

    // 使用したメモリを解放し，再確保し，初期化
    a = mi_rfree(a); // a のメモリを解放
    b = mi_rfree(b); // b のメモリを解放
    a = mi_rallocate(); // a のメモリを再確保し，初期化
    b = mi_rallocate(); // b のメモリを再確保し，初期化

    // 複素数の乗算の虚部計算
    mir_mirmir_mul(a, x->r, y->i); // x の実部の区間と y の虚部の区間を乗算し，
a に格納
    mir_mirmir_mul(b, x->i, y->r); // x の虚部の区間と y の実部の区間を乗算し，
b に格納
    mir_mirmir_add(z->i, a, b); // a と b を加算し，z の虚部に格納

    // 使用したメモリを解放
    a = mi_rfree(a);
    b = mi_rfree(b);
}

// 2 つの複素数の区間を受け取り，仮数部を n ビット増加させて，乗算して複素数の区
間で返す関数
void mic_micmic_mul_plusnbits(mi_c* z, mi_c* x, mi_c* y, int n) {
    mi_c *x0, *y0, *z0; // 精度を上げた複素数の区間を格納するためのポインタ変
```

数の宣言

```
    // 精度を設定
    set_default_prec(mpfr_get_prec(x->r->down) + n); // x の実部の下限に n ビット追加して，一時的にデフォルトの精度を設定

    // mi_c 型のメモリを確保し，初期化
    x0 = mi_callocate(); // x0 のメモリを確保し，初期化
    y0 = mi_callocate(); // y0 のメモリを確保し，初期化
    z0 = mi_callocate(); // z0 のメモリを確保し，初期化

    // mi_c 型の値を受け取り，mi_c 型の変数に格納
    mic_mic_set(x0, x); // 入力された複素数の区間 x を仮数部のビット数を増やした複素数の区間 x0 に格納
    mic_mic_set(y0, y); // 入力された複素数の区間 y を仮数部のビット数を増やした複素数の区間 y0 に格納

    // 複素数の乗算
    mic_micmic_mul(z0, x0, y0); // 複素数の区間 x0 と複素数の区間 y0 を乗算し，z0 に格納

    // 精度を元に戻す
    set_default_prec(mpfr_get_prec(x->r->down)); // x の実部の下限の精度と同じ精度に設定

    // mi_c 型の値を受け取り，mi_c 型の変数に格納
    mic_mic_set(z, z0); // 仮数部のビット数を増やした複素数の区間 z0 を入力された複素数の区間 z に格納

    // 使用したメモリを解放
    x0 = mi_cfree(x0); // x0 のメモリを解放
    y0 = mi_cfree(y0); // y0 のメモリを解放
    z0 = mi_cfree(z0); // z0 のメモリを解放
}

// 2 つの複素数の区間を受け取り，仮数部を n 倍して，乗算して複素数の区間で返す関数
void mic_micmic_mul_timesnbits(mi_c* z, mi_c* x, mi_c* y, int n) {
```

```
    mi_c *x0, *y0, *z0; // 精度を上げた複素数の区間を格納するためのポインタ変数の宣言

    // 精度を設定
    set_default_prec(mpfr_get_prec(x->r->down) * n); // x の実部の下限の精度に n を掛けて，一時的にデフォルトの精度を設定

    // mi_c 型のメモリを確保し，初期化
    x0 = mi_callocate(); // x0 のメモリを確保し，初期化
    y0 = mi_callocate(); // y0 のメモリを確保し，初期化
    z0 = mi_callocate(); // z0 のメモリを確保し，初期化

    // mi_c 型の値を受け取り，mi_c 型の変数に格納
    mic_mic_set(x0, x); // 入力された複素数の区間 x を仮数部のビット数を増やした複素数の区間 x0 に格納
    mic_mic_set(y0, y); // 入力された複素数の区間 y を仮数部のビット数を増やした複素数の区間 y0 に格納

    // 複素数の乗算
    mic_micmic_mul(z0, x0, y0); // 複素数の区間 x0 と複素数の区間 y0 を乗算し，z0 に格納

    // 精度を元に戻す
    set_default_prec(mpfr_get_prec(x->r->down)); // x の実部の下限の精度と同じ精度に設定

    // mi_c 型の値を受け取り，mi_c 型の変数に格納
    mic_mic_set(z, z0); // 仮数部のビット数を増やした複素数の区間 z0 を入力された複素数の区間 z に格納

    // 使用したメモリを解放
    x0 = mi_cfree(x0); // x0 のメモリを解放
    y0 = mi_cfree(y0); // y0 のメモリを解放
    z0 = mi_cfree(z0); // z0 のメモリを解放
}

// 2 つの複素数の区間を受け取り，区間幅が 1ulp になるように精度を上げて，乗算して
```

複素数の区間で返す関数

```
void mic_micmic_mul_1ulp(mi_c* z, mi_c* x, mi_c* y) {
    mi_c *x0, *y0, *z0; // 精度を上げた複素数の区間を格納するためのポインタ変数の宣言

    // 精度を設定
    set_default_prec((mpfr_get_prec(x->r->down) * 2) + 16); // x の実部の下限の精度を 2 倍して，さらに 16 ビット追加して，一時的にデフォルトの精度を設定

    // mi_c 型のメモリを確保し，初期化
    x0 = mi_callocate(); // x0 のメモリを確保し，初期化
    y0 = mi_callocate(); // y0 のメモリを確保し，初期化
    z0 = mi_callocate(); // z0 のメモリを確保し，初期化

    // mi_c 型の値を受け取り，mi_c 型の変数に格納
    mic_mic_set(x0, x); // 入力された複素数の区間 x を精度を上げた複素数の区間 x0 に格納
    mic_mic_set(y0, y); // 入力された複素数の区間 y を精度を上げた複素数の区間 y0 に格納

    // 複素数の乗算
    mic_micmic_mul(z0, x0, y0); // 精度を上げた複素数の区間 x0 と精度を上げた複素数の区間 y0 を乗算し，z0 に格納

    // 精度を元に戻す
    set_default_prec(mpfr_get_prec(x->r->down)); // x の実部の下限の精度と同じ精度に設定

    // mi_c 型の値を受け取り，mi_c 型の変数に格納
    mic_mic_set(z, z0); // 精度を上げた複素数の区間 z0 を入力された複素数の区間 z に格納

    // 使用したメモリを解放
    x0 = mi_cfree(x0); // x0 のメモリを解放
    y0 = mi_cfree(y0); // y0 のメモリを解放
    z0 = mi_cfree(z0); // z0 のメモリを解放
}
```

```
// 2つの複素数の区間を受け取り，除算して複素数の区間で返す関数
void mic_micmic_div(mi_c* z, mi_c* x, mi_c* y) {
    // 関数内で使用するポインタ変数の宣言
    mi_r *a, *b, *c, *d;

    // mi_r型のメモリを確保し，初期化
    a = mi_rallocate(); // aのメモリを確保し，初期化
    b = mi_rallocate(); // bのメモリを確保し，初期化
    c = mi_rallocate(); // cのメモリを確保し，初期化
    d = mi_rallocate(); // dのメモリを確保し，初期化

    // 実部の分子を計算
    mir_mirmir_mul(a, x->r, y->r); // xの実部の区間とyの実部の区間を乗算し，aに格納
    mir_mirmir_mul(b, x->i, y->i); // xの虚部の区間とyの虚部の区間を乗算し，bに格納
    mir_mirmir_add(c, a, b); // aとbを加算し，cに格納

    // 使用したメモリを解放し，再確保し，初期化
    a = mi_rfree(a); // aのメモリを解放
    b = mi_rfree(b); // bのメモリを解放
    a = mi_rallocate(); // aのメモリを再確保し，初期化
    b = mi_rallocate(); // bのメモリを再確保し，初期化

    // 実部の分母を計算
    mir_mirmir_mul(a, y->r, y->r); // yの実部の区間を2乗し，aに格納
    mir_mirmir_mul(b, y->i, y->i); // yの虚部の区間を2乗し，bに格納
    mir_mirmir_add(d, a, b); // aとbを加算し，dに格納

    // 実部の除算を計算
    mir_mirmir_div(z->r, c, d); // cをdで除算し，zの実部に格納

    // 使用したメモリを解放し，再確保し，初期化
    a = mi_rfree(a); // aのメモリを解放
    b = mi_rfree(b); // bのメモリを解放
    c = mi_rfree(c); // cのメモリを解放
```

```
    d = mi_rfree(d); // d のメモリを解放
    a = mi_rallocate(); // a のメモリを再確保し，初期化
    b = mi_rallocate(); // b のメモリを再確保し，初期化
    c = mi_rallocate(); // c のメモリを再確保し，初期化
    d = mi_rallocate(); // d のメモリを再確保し，初期化

    // 虚部の分子を計算
    mir_mirmir_mul(a, x->i, y->r); // x の虚部の区間と y の実部の区間を乗算し，
a に格納
    mir_mirmir_mul(b, x->r, y->i); // x の実部の区間と y の虚部の区間を乗算し，
b に格納
    mir_mirmir_sub(c, a, b); // a から b を減算し，c に格納

    // 虚部の分母を計算
    mir_mirmir_mul(a, y->r, y->r); // y の実部の区間を 2 乗し，a に格納
    mir_mirmir_mul(b, y->i, y->i); // y の虚部の区間を 2 乗し，b に格納
    mir_mirmir_add(d, a, b); // a と b を加算し，d に格納

    // 虚部の除算を実行
    mir_mirmir_div(z->i, c, d); // c を d で除算し，z の虚部に格納

    // 使用したメモリを解放
    a = mi_rfree(a);
    b = mi_rfree(b);
    c = mi_rfree(c);
    d = mi_rfree(d);
}

// 2 つの複素数の区間を受け取り，仮数部を n ビット増加させて，除算して複素数の区
間で返す関数
void mic_micmic_div_plusnbits(mi_c* z, mi_c* x, mi_c* y, int n) {
    mi_c *x0, *y0, *z0; // 精度を上げた複素数の区間を格納するためのポインタ変
数の宣言

    // 精度を設定
    set_default_prec(mpfr_get_prec(x->r->down) + n); // x の実部の下限に n ビ
ット追加して，一時的にデフォルトの精度を設定
```

```
    // mi_c 型のメモリを確保し，初期化
    x0 = mi_callocate(); // x0 のメモリを確保し，初期化
    y0 = mi_callocate(); // y0 のメモリを確保し，初期化
    z0 = mi_callocate(); // z0 のメモリを確保し，初期化

    // mi_c 型の値を受け取り，mi_c 型の変数に格納
    mic_mic_set(x0, x); // 入力された複素数の区間 x を仮数部のビット数を増やした複素数の区間 x0 に格納
    mic_mic_set(y0, y); // 入力された複素数の区間 y を仮数部のビット数を増やした複素数の区間 y0 に格納

    // 複素数の除算
    mic_micmic_div(z0, x0, y0); // 複素数の区間 x0 と複素数の区間 y0 を除算し，z0 に格納

    // 精度を元に戻す
    set_default_prec(mpfr_get_prec(x->r->down)); // x の実部の下限の精度と同じ精度に設定

    // mi_c 型の値を受け取り，mi_c 型の変数に格納
    mic_mic_set(z, z0); // 仮数部のビット数を増やした複素数の区間 z0 を入力された複素数の区間 z に格納

    // 使用したメモリを解放
    x0 = mi_cfree(x0); // x0 のメモリを解放
    y0 = mi_cfree(y0); // y0 のメモリを解放
    z0 = mi_cfree(z0); // z0 のメモリを解放
}

// 2 つの複素数の区間を受け取り，仮数部を n 倍して，除算して複素数の区間で返す関数
void mic_micmic_div_timesnbits(mi_c* z, mi_c* x, mi_c* y, int n) {
    mi_c *x0, *y0, *z0; // 精度を上げた複素数の区間を格納するためのポインタ変数の宣言

    // 精度を設定
    set_default_prec(mpfr_get_prec(x->r->down) * n); // x の実部の下限の精度
```

に n を掛けて，一時的にデフォルトの精度を設定

```
    // mi_c 型のメモリを確保し，初期化
    x0 = mi_callocate(); // x0 のメモリを確保し，初期化
    y0 = mi_callocate(); // y0 のメモリを確保し，初期化
    z0 = mi_callocate(); // z0 のメモリを確保し，初期化

    // mi_c 型の値を受け取り，mi_c 型の変数に格納
    mic_mic_set(x0, x); // 入力された複素数の区間 x を仮数部のビット数を n 倍した複素数の区間 x0 に格納
    mic_mic_set(y0, y); // 入力された複素数の区間 y を仮数部のビット数を n 倍した複素数の区間 y0 に格納

    // 複素数の除算
    mic_micmic_div(z0, x0, y0); // 複素数の区間 x0 と複素数の区間 y0 を除算し，z0 に格納

    // 精度を元に戻す
    set_default_prec(mpfr_get_prec(x->r->down)); // x の実部の下限の精度と同じ精度に設定

    // mi_c 型の値を受け取り，mi_c 型の変数に格納
    mic_mic_set(z, z0); // 仮数部のビット数を n 倍した複素数の区間 z0 を入力された複素数の区間 z に格納

    // 使用したメモリを解放
    x0 = mi_cfree(x0); // x0 のメモリを解放
    y0 = mi_cfree(y0); // y0 のメモリを解放
    z0 = mi_cfree(z0); // z0 のメモリを解放
}

// 2 つの複素数の区間を受け取り，区間幅が 1ulp になるように精度を上げて，除算して複素数の区間で返す関数
void mic_micmic_div_1ulp(mi_c* z, mi_c* x, mi_c* y) {
    mi_c *x0, *y0, *z0; // 精度を上げた複素数の区間を格納するためのポインタ変数の宣言
```

```
    // 精度を設定
    set_default_prec((mpfr_get_prec(x->r->down) * 2) + 16); // x の実部の下限の精度を 2 倍して，さらに 16 ビット追加して，一時的にデフォルトの精度を設定

    // mi_c 型のメモリを確保し，初期化
    x0 = mi_callocate(); // x0 のメモリを確保し，初期化
    y0 = mi_callocate(); // y0 のメモリを確保し，初期化
    z0 = mi_callocate(); // z0 のメモリを確保し，初期化

    // mi_c 型の値を受け取り，mi_c 型の変数に格納
    mic_mic_set(x0, x); // 入力された複素数の区間 x を精度を上げた複素数の区間 x0 に格納
    mic_mic_set(y0, y); // 入力された複素数の区間 y を精度を上げた複素数の区間 y0 に格納

    // 複素数の除算
    mic_micmic_div(z0, x0, y0); // 精度を上げた複素数の区間 x0 と精度を上げた複素数の区間 y0 を除算し，z0 に格納

    // 精度を元に戻す
    set_default_prec(mpfr_get_prec(x->r->down)); // x の実部の下限の精度と同じ精度に設定

    // mi_c 型の値を受け取り，mi_c 型の変数に格納
    mic_mic_set(z, z0); // 精度を上げた複素数の区間 z0 を入力された複素数の区間 z に格納

    // 使用したメモリを解放
    x0 = mi_cfree(x0); // x0 のメモリを解放
    y0 = mi_cfree(y0); // y0 のメモリを解放
    z0 = mi_cfree(z0); // z0 のメモリを解放
}
```

A.4　研究で使用したヘッダファイルやライブラリの説明（C++）

A.4.1　iostream

#include<iostream> は C++の標準入出力ストリームを扱うヘッダファイルである．このヘッダは主に以下の 4 つのオブジェクトを提供する．

- cin：標準入力ストリーム．キーボードからの入力を受け取るために使われる．
- cout：標準出力ストリーム．画面に出力を行うために使われる．
- cerr：標準エラー出力ストリーム．エラーメッセージを出力するために使われ，cout とは異なり，バッファリングされない．
- clog：ログ出力ストリーム．エラーメッセージや警告などのログ情報を出力するために使われる．

cin と cout は入力と出力のために最も頻繁に使用される．例えば，cout を使って画面に文字列を出力したり，cin を使ってユーザーから入力を受け取ることができる．以下に簡単な例を示す．

```
#include <iostream>

int main() {
    int number;
    std::cout << "数字を入力してください: ";
    std::cin >> number;
    std::cout << "入力された数字は " << number << " です。";
    return 0;
}
```

A.4.2　iomanip

#include<iomanip> は C++の標準ライブラリの一部で，出力ストリームの書式を操作するためのユーティリティとオブジェクトを提供するヘッダファイルである．このヘッダファイルには，出力の書式を細かく制御するための機能が多数含まれており，以下のようなオブジェクトと関数がある．

- setprecision：出力する浮動小数点数の精度を設定する．
- setw：出力するフィールドの幅を設定する．
- left，right：出力を左寄せまたは右寄せにする．

- fixed，scientific：浮動小数点数の表示方法を固定小数点形式または科学技術形式に設定する．

これらの機能を利用することで，出力の見栄えを向上させることができる．例えば，setw と setprecision を組み合わせて，特定の幅に合わせて浮動小数点数を出力することが可能である．以下に簡単な例を示す．

```
#include <iostream>
#include <iomanip>

int main() {
    double pi = 3.14159265;
    std::cout << "Pi with 3 decimal places: ";
    std::cout << std::setprecision(3) << pi << std::endl;
    std::cout << "Pi in a 10-character field: ";
    std::cout << std::setw(10) << pi << std::endl;
    return 0;
}
```

A.4.3 cmath

#include<cmath>は，C++で数学的な関数を扱うための標準ヘッダファイルである．このヘッダファイルには，三角関数，指数関数，対数関数，べき乗計算，平方根．およびその他の一般的な数学関数が含まれている．主な関数には次のようなものがある．

- std::sin，std::cos，std::tan：三角関数
- std::exp：指数関数
- std::log，std::log10：対数関数
- std::pow：べき乗計算
- std::sqrt：平方根の計算
- std::abs：絶対値の計算

これらの関数は，科学計算，エンジニアリング，データ分析など，さまざまな分野で広く使われている．例えば，std::sin 関数は角度の正弦を計算し，std::log 関数は数値の自然対数を計算する．以下に簡単な例を示す．

```
#include <iostream>
#include <cmath>

int main() {
    double angle = 45.0;
    double radians = angle * M_PI / 180.0; // 角度をラジアンに変換
    std::cout << "sin(" << angle << "度) = " << std::sin(radians) << std::endl;
    return 0;
}
```

A.4.4 chrono

#include<chrono>はC++で日付や時間を扱うためのヘッダファイルである．このヘッダファイルは時間の測定や操作に関連する様々なクラスや関数を提供する．主に以下のような機能がある．

- 時間の点または期間を表すためのクラス（例：time_point，duration）．
- 時間の測定と操作に使われる様々なクロック（例：system_clock，steady_clock）．
- 時間の単位（例：秒，ミリ秒，ナノ秒）．

chrono は，プログラムの実行時間の計測や，特定の時間にイベントをスケジューリングする際によく使用される．以下に簡単な例を示す．

```
#include <iostream>
#include <chrono>

int main() {
    auto start = std::chrono::high_resolution_clock::now();
    // ここに時間を測定したい処理を書く
    auto end = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double> elapsed = end - start;
    std::cout << "経過時間: " << elapsed.count() << "秒";
    return 0;
}
```

B　プログラミング言語や丸めモードの切り替え，機械区間演算への理解を深めるための数値実験（付録）

B.1　MATLAB による内積計算の機械区間演算

実験目的　簡単な内積計算を例にし，MATLAB での機械区間演算を実際に行う．区間演算とはどのようなものであり，デフォルトの精度ではどのような結果が得られるのか理解するためにこの実験を行う．

実験方法　テストケースとして，ベクトル $x = [1, 2, 3]$ と $y = [5, 6, 7]$ を使用する．次に，up コマンドを使用して丸めモードを上向きに設定する．その後，ベクトル x と y をそれぞれ 10 で割ったベクトル $x_u = [0.1, 0.2, 0.3]$ と $y_u = [0.5, 0.6, 0.7]$ を生成する．最後に，これらのベクトルの内積を計算し，結果を s_u に格納する．コマンドウィンドウでの操作は以下の通りである．

```
>> x=[1,2,3];
>> y=[5,6,7];
>> up;
>> xu=x/10;
>> yu=y/10;
>> su=xu*yu';
```

down コマンドを使用して丸めモードを下向きに設定する，同様に，ベクトル x と y を 10 で割ったベクトル $x_l = [0.1, 0.2, 0.3]$ と $y_l = [0.5, 0.6, 0.7]$ を生成し，これらのベクトルの内積を計算し，結果を s_l に格納する．コマンドウィンドウでの操作は以下の通りである．

```
>> down;
>> xl=x/10;
>> yl=y/10;
>> sl=xl*yl';
>> disp(sprintf('[%25.20f,%25.20f]',sl,su))
```

実験結果　最終的な結果は，次のようになった:

$$[\ 0.37999999999999989341,\ 0.38000000000000017097\ ]$$

考察　上記の結果からわかるように，丸め誤差が存在した．計算の中間結果の丸めにより，s_l の値は理論的な値 0.38 とは異なる結果となった．上向きの場合 (s_u) では，0.38 よりも大きい値になり，下向きの場合 (s_l) では，0.38 よりも小さい値になった．このような丸め誤差は，計算精度に影響を及ぼす．また，有効桁数 16 桁ほどが信用できると計算結果から推測できる．10 進数で 16 桁ということは 2 進数では $\log_2 10^{16}$ が約 53 であることから，仮数部が 52bits の倍精度がデフォルトであることが予想できる．

B.2　単精度と倍精度

単精度は浮動小数点数において，符号ビット 1bit，指数部 8bits，仮数部 23bits，というのはすでに述べた．これに対し，倍精度は符号ビット 1bit，指数部 11bits，仮数部 52bits，である．しかし，これだけを比較しても，具体的にどんな差があるのかよくわからないので, 実際に数値を計算し, それらの差について検討していく．本節では, C++と MATLAB を使用して単精度と倍精度の演算速度を比較する実験について述べる．C 言語ではなく C++と MATLAB を使用する理由に関しては，この実験を行ったときの自分が C 言語をあまり使っていなかったからである．この実験の目的は, 単精度と倍精度の演算における精度と速度の違いを明らかにすることである．

B.2.1　$\mathcal{O}\left(10^6\right)$ の加算

実験目的　本実験では，C++と MATLAB の 2 つのプログラミング言語を使用して，単精度と倍精度の数値演算の精度と速度を比較することを目的としている.

実験方法　それぞれの言語で，単精度（float 型または single 型）と倍精度（double 型）の 11.11 という数値を $1,000,000$ 回加算し，その結果と実行時間を計測した．この計測を 10 回繰り返し，平均実行時間を求めた. 以下に，C++と MATLAB で使用したソースコードを示す．C++のコードは単精度で，MATLAB のコードは倍精度である．C++の倍精度は下記のコードの float を double に変えて，MATLAB の単精度は double を single に変えればよい.

```
// C++のコード

#include <iostream> // 入出力ストリームのインクルード
#include <chrono> // 時間計測用のライブラリをインクルード
#include <iomanip> // 入出力のマニピュレータをインクルード

int main() {
    // 単精度の変数を初期化
```

```
    float result = 0.0f;
    float a = 11.11f;

    auto start = std::chrono::high_resolution_clock::now(); // 高精度タイマーを開始

    // 単精度の数値を1,000,000回加算
    for (int i = 0; i < 1000000; i++) {
        result += a;
    }

    auto end = std::chrono::high_resolution_clock::now(); // 高精度タイマーを停止

    std::chrono::duration<double> diff = end - start; // 実行時間を計算

    std::cout << std::fixed << std::setprecision(8); // 出力の形式を固定小数点とし，小数点以下8桁で表示
    std::cout << "Result: " << result << std::endl; // 計算結果を出力

    std::cout << std::scientific << std::setprecision(4); // 出力の形式を科学技術表記とし，小数点以下4桁で表示
    std::cout << "Execution time: "
    << diff.count() << " s" << std::endl; // 実行時間を秒単位で出力

    return 0; // プログラムを正常終了
}
```

```
% MATLABのコード

% 倍精度の変数を初期化
result = double(0.0);
a = double(11.11);

tic; % 高精度タイマーを開始

% 倍精度の数値を1,000,000回加算
for i = 1:1e6
```

```
    result = result + a;
end

elapsedTime = toc; % 高精度タイマーを停止

fprintf('Result: %.8f\n', result); % 計算結果を出力

fprintf('Execution time: %.4e s\n', elapsedTime); % 実行時間を出力
```

実験結果　実験結果を表 1 に示す.

		C++	MATLAB
単精度	計算結果	11023200.00000000	11023200.00000000
	真値との誤差 [%]	0.7812	0.7812
	平均実行時間 [ms]	4.4942	1.0724
倍精度	計算結果	11109999.99996448	11109999.99996448
	真値との誤差 [%]	0.0000000003204	0.0000000003204
	平均実行時間 [ms]	4.4754	1.1198

表 1: C++と MATLAB による単精度と倍精度の計算結果と平均実行時間（$\mathcal{O}(10^6)$ の加算）

考察　表 1 から，単精度と倍精度の演算における精度と速度の違いが明らかになった．まず，精度について考察する．単精度の計算結果は真値との誤差が約 0.78% となり，倍精度の計算結果は真値との誤差が約 0.00000000032% となった．これは，倍精度が単精度よりも精度が高いことを示している．次に，速度について考察する．C++は，倍精度の演算速度が単精度の演算速度よりもわずかに速かった．それに対し，MATLAB は，単精度の演算速度が倍精度の演算速度よりもわずかに速かった．しかし，その差は非常に小さく, 実用上は無視できる程度である．精度が重要な場合は倍精度を, 速度が重要な場合は単精度を使用することが一般には推奨される．しかし, 現代のコンピュータでは, 倍精度の演算速度も十分に速いため, 精度を優先することが多い．また、MATLAB は C++よりも演算速度が速かったが, これは MATLAB が数値計算に特化した言語であるためと考えられる．

B.2.2　$\mathcal{O}(1)$ の乗算

実験目的　先ほどの実験では，$\mathcal{O}(10^6)$ の加算を行い，精度と速度の比較をしたが，次は $\mathcal{O}(1)$ の乗算を行い，精度と速度の比較をする．つまり，計算量のオーダーと演算を変えて実験を行うということである．

実験方法　次は, それぞれの言語で, 単精度 (float 型または single 型) と倍精度 (double 型) の 11.11 という数値を乗算し, その結果と実行時間を計測した．以下に, C++と MATLAB で使用したソースコードを示す．

```
// C++のコード

#include <iostream> // 入出力ストリームのインクルード
#include <chrono> // 時間計測用のライブラリをインクルード
#include <iomanip> // 入出力のマニピュレータをインクルード
int main() {
    double a = 11.11; // 倍精度浮動小数点数 a を 11.11 で初期化
    double b = 11.11; // 倍精度浮動小数点数 b を 11.11 で初期化

    auto start = std::chrono::high_resolution_clock::now(); // 高精度タイマーの開始時間を記録

    double result = a * b; // a と b の積を result に格納

    auto end = std::chrono::high_resolution_clock::now(); // 高精度タイマーの終了時間を記録
```

```
    std::chrono::duration<double> diff = end - start; // startとendの差(実行時間)を計算

    std::cout << std::fixed << std::setprecision(8); // 出力の形式を固定小数点とし，小数点以下8桁で表示
    std::cout << "Result: " << result << std::endl; // 結果(result)を出力
    std::cout << std::scientific << std::setprecision(4); // 出力の形式を科学技術表記とし，小数点以下4桁で表示
    std::cout << "Execution time: "
    << diff.count() << " s" << std::endl; // 実行時間を秒単位で出力

    return 0; // プログラムを正常終了
}

% MATLABのコード

% 単精度の変数を初期化
a = single(11.11);
b = single(11.11);

tic; % 高精度タイマーを開始

result = a * b; % 単精度の数値を乗算

elapsedTime = toc; % 高精度タイマーを停止

fprintf('Result: %.8f\n', result); % 計算結果を出力

fprintf('Execution time: %.4e s\n', elapsedTime); % 実行時間を出力
```

実験結果　実験結果を表 2 に示す.

		C++	MATLAB
単精度	計算結果	123.43209076	123.43209076
	真値との誤差 [%]	0.000007486	0.000007486
	平均実行時間 [ms]	0.000004160	0.001079
倍精度	計算結果	123.43210000	123.43210000
	真値との誤差 [%]	0.0	0.0
	平均実行時間 [ms]	0.000005420	0.001096

表 2: C++と MATLAB による単精度と倍精度の計算結果と平均実行時間（$\mathcal{O}(1)$ の乗算）

考察　表 2 から，単精度と倍精度の演算における精度と速度の違いが明らかになった．単精度の計算結果は真値との誤差が約 0.000007486% となり, 倍精度の計算結果は真値との誤差が 0.0% となった．これは, 倍精度が単精度よりも精度が高いことを示している．また、C++と MATLAB の実行時間を比較すると, C++の実行時間が短い. 精度が重要な場合は倍精度を, 速度が重要な場合は単精度を使用することが一般には推奨される. しかし, 現代のコンピュータでは, 倍精度の演算速度も十分に速いため, 精度を優先することが多い. また,MATLAB は C++よりも演算速度が速かったが, これは MATLAB が数値計算に特化した言語であるためと考えられる.

B.2.3　$\mathcal{O}(10^6)$ の区分求積

実験目的　本実験では, C++と MATLAB の 2 つのプログラミング言語を使用して, 単精度と倍精度の数値積分の精度と速度を比較することを目的としている.

実験方法　それぞれの言語で，単精度（float 型または single 型）と倍精度（double 型）の x^3 という関数を 1,000,000 回積分し，その結果と実行時間を計測した．この計測を 10 回繰り返し，平均実行時間を求めた．以下に，C++と MATLAB で使用したソースコードを示す.

```
// C++のコード

#include <iostream> // 入出力ストリームのインクルード
#include <chrono> // 時間計測用のライブラリをインクルード
#include <cmath> // 数学関数のインクルード
#include <iomanip> // 入出力のマニピュレータをインクルード

// 関数: f(x) = x^3
float f(float x) {
    return std::pow(x, 3); // x の 3 乗を返す
}

int main() {
    float a = 0.0, b = 1.0; // 積分区間の下限 a と上限 b をそれぞれ 0.0 と 1.0 で初期化
    int n = 1e6; // 計算点の数 n を 100 万で初期化

    float h = (b - a) / n; // 刻み幅 h を計算
```

```
    float integral = 0.0; // 積分の結果を格納する変数 integral を 0.0 で初期化

    auto start = std::chrono::high_resolution_clock::now(); // 高精度タイマーの開始時間を記録

    // 区分求積法による積分計算
    for (int i = 0; i < n; i++) {
        float x = a + i * h;
        integral += f(x) * h;
    }

    auto stop = std::chrono::high_resolution_clock::now(); // 高精度タイマーの終了時間を記録

    auto duration =
    std::chrono::duration_cast<std::chrono::microseconds>
    (stop - start); // start と stop の差 (実行時間) を計算

    std::cout << std::fixed << std::setprecision(10); // 出力の形式を固定小数点とし，小数点以下 10 桁で表示
    std::cout << "Result: " << integral << std::endl; // 結果 (integral) を出力
    std::cout << std::scientific << std::setprecision(4); // 出力の形式を科学技術表記とし，小数点以下 4 桁で表示
    std::cout << "Execution time: " <<
    duration.count() / 1e6 << " seconds" << std::endl; // 実行時間を秒単位で出力

    return 0; // プログラムを正常終了
}
% MATLAB のコード

% 関数: f(x) = x^3
f = @(x) x.^3; % 関数 f を x の 3 乗として定義

% 積分区間の下限 a と上限 b をそれぞれ 0.0 と 1.0 で設定
a = 0.0;
```

```
b = 1.0;

n = 1e6; % 計算点の数 n を 100 万で設定

h = (b - a) / n; % 刻み幅 h を計算

integral = 0.0; % 積分の結果を格納する変数 integral を 0.0 で初期化

tic; % 高精度タイマーを開始

% 区分求積法による積分計算
for i = 1:n
    x = a + (i - 1) * h;
    integral = integral + f(x) * h;
end

elapsedTime = toc; % 高精度タイマーを停止

fprintf('Result: %.10f\n', integral); % 結果を小数点以下 10 桁で出力
fprintf('Execution time: %.4e seconds\n', elapsedTime); % 実行時間を秒単位で出力
```

実験結果　実験結果を表 3 に示す.

		C++	MATLAB
単精度	計算結果	0.2500020564	0.2500020564
	真値との誤差 [%]	0.0008226	0.0008226
	平均実行時間 [ms]	14.391	113.264
倍精度	計算結果	0.2499995000	0.2499995000
	真値との誤差 [%]	0.0002	0.0002
	平均実行時間 [ms]	14.265	110.152

表 3: C++と MATLAB による単精度と倍精度の計算結果と平均実行時間（$\mathcal{O}(10^6)$ の区分求積）

考察　表 3 から，単精度と倍精度の数値積分の精度と速度を比較することができる．まず，計算結果を見ると，C++と MATLAB の両方のプログラムで，単精度および倍精度の場合において，計算結果は非常に近い値となった．真値との誤差も非常に小さく，単精度の場合は 0.0008226%，倍精度の場合は 0.0002% となっている．次に，実行時間について考察する．表 3 を見ると，MATLAB の実行時間が C++に比べてかなり長いことが分かる．この結果から，C++の方が MATLAB に比べて数値積分の計算速度が高いことが示唆される．C++はコンパイル言語であり，MATLAB はインタプリタ言語であるため，C++の方が効率的なコードの実行が可能である．しかし，第 A.5.1 節の実験と比較すると，MATLAB の方が高速であった．この原因はいろいろ考えられる．まずは，区分求積では関数を使っているが，加算では関数を使っていないことが理由の一つだと考えられる．C++の関数呼び出しが，MATLAB よりも高速だった可能性がある．ループの最適化においては MATLAB の方が高速だが，関数呼び出しに対する最適化が難しかったのかもしれない．なお，本実験では x^3 という比較的単純な関数を使用しているが，より複雑な関数や高次元の積分の場合においても，C++の方が MATLAB よりも高速な計算が期待できる可能性がある．

B.3　倍精度の内積計算と丸め誤差

実験目的　本実験では，丸め誤差におけるベクトルの内積の精度，計算速度，また，それぞれの丸め方における数値の差を C 言語，MATLAB で比較していく．デフォルト，最近点丸め，下向き丸め，上向き丸めの 4 つで比較する．それぞれの丸め方による違いや，言語による違いを明らかにする．

実験方法　以下に，C 言語，MATLAB で使用したソースコードを示す．

```
// C 言語のコード

#include <stdio.h> // 標準入出力ライブラリをインクルード
#include <stdlib.h> // 標準ライブラリをインクルード
#include <time.h> // 時間関連のライブラリをインクルード
#include <fenv.h> // 浮動小数点環境のライブラリをインクルード
#define SIZE 10 // ベクトルのサイズを定義
#define RUNS 100000000 // 実行回数を定義

double vectorA[SIZE] = {34.88, -19.59, -35.99, -68.27, 62.03,
-94.03, -23.40, 33.90, -48.42, 84.51}; // ベクトル A を定義
double vectorB[SIZE] = {-59.21, -64.66, 27.00, -24.50, -86.04,
```

```
6.43, -91.92, -11.56, 70.51, 77.32}; // ベクトルBを定義

// ベクトルの内積を計算する関数を定義
double dot_product(double a[SIZE], double b[SIZE]) {
    double sum = 0.0; // 合計値を初期化
    for (int i = 0; i < SIZE; i++) {
        sum += a[i] * b[i]; // ベクトルの要素同士を掛けて合計に加える
    }
    return sum; // 内積の結果を返す
}

int main() {
    clock_t start, end; // 開始時間と終了時間を定義
    double elapsed_time1 = 0.0, elapsed_time2 = 0.0; // 経過時間を定義
    double Default, Near; // 内積の結果を格納する変数を定義

    // Default
    start = clock(); // 計算開始時間を記録
    for (int run = 0; run < RUNS; run++) {
        Default = dot_product(vectorA, vectorB); // デフォルトの丸めモードで
内積を計算
    }
    end = clock(); // 計算終了時間を記録
    elapsed_time1 = ((double) (end - start)) / CLOCKS_PER_SEC; // 経過時間
を計算

    // Near
    fesetround(FE_TONEAREST); // 丸めモードを最も近い値に設定
    start = clock(); // 計算開始時間を記録
    for (int run = 0; run < RUNS; run++) {
        Near = dot_product(vectorA, vectorB); // 最も近い値の丸めモードで内
積を計算
    }
    end = clock(); // 計算終了時間を記録
    elapsed_time2 = ((double) (end - start)) / CLOCKS_PER_SEC; // 経過時間
を計算
```

```
    // Lower bound
    fesetround(FE_DOWNWARD); // 丸めモードを下向きに設定
    start = clock(); // 計算開始時間を記録
    for (int run = 0; run < RUNS; run++) {
        Lower_bound = dot_product(vectorA, vectorB); // 下向きの丸めモードで
内積を計算
    }
    end = clock(); // 計算終了時間を記録
    elapsed_time3 = ((double) (end - start)) / CLOCKS_PER_SEC; // 経過時間
を計算

    // Upper bound
    fesetround(FE_UPWARD); // 丸めモードを上向きに設定
    start = clock(); // 計算開始時間を記録
    for (int run = 0; run < RUNS; run++) {
        Upper_bound = dot_product(vectorA, vectorB); // 上向きの丸めモードで
内積を計算
    }
    end = clock(); // 計算終了時間を記録
    elapsed_time4 = ((double) (end - start)) / CLOCKS_PER_SEC; // 経過時間
を計算

    printf("Default    : %.25f\n", Default); // デフォルトの丸めモードでの内
積の結果を表示
    printf("Near       : %.25f\n", Near); // 最も近い値の丸めモードでの内積
の結果を表示
    printf("Lower bound: %.25f\n", Lower_bound); // 下向きの丸めモードでの内
積の結果を表示
    printf("Upper bound: %.25f\n", Upper_bound); // 上向きの丸めモードでの内
積の結果を表示
    printf("Average elapsed time1:
    %.10f ms\n", (elapsed_time1 / RUNS) * 1000); // デフォルトの丸めモードで
の計算時間を表示
    printf("Average elapsed time2:
    %.10f ms\n", (elapsed_time2 / RUNS) * 1000); // 最も近い値の丸めモードで
の計算時間を表示
    printf("Average elapsed time3:
```

```
    %.10f ms\n", (elapsed_time3 / RUNS) * 1000); // 下向きの丸めモードでの計
算時間を表示
    printf("Average elapsed time4:
    %.10f ms\n", (elapsed_time4 / RUNS) * 1000); // 上向きの丸めモードでの計
算時間を表示

    return 0; // プログラムを終了
}
% MATLAB のコード

vectorA =
[34.88, -19.59, -35.99, -68.27, 62.03,
     -94.03, -23.40, 33.90, -48.42, 84.51]; % ベクトル A を定義

vectorB =
[-59.21, -64.66, 27.00, -24.50, -86.04,
     6.43, -91.92, -11.56, 70.51, 77.32]; % ベクトル B を定義

vectorA = double(vectorA); % ベクトル A を double 型に変換

vectorB = double(vectorB); % ベクトル B を double 型に変換

RUNS = 100000000; % 実行回数を定義

tic; % 計算開始時間を記録

% デフォルトの丸めモードで内積を計算
for run = 1:RUNS
    default = sum(vectorA .* vectorB);
end

elapsed_time1 = toc; % 経過時間を計算

setround(0); % 丸めモードを最も近い値に設定

tic; % 計算開始時間を記録
```

```
% 最も近い値の丸めモードで内積を計算
for run = 1:RUNS
    near = sum(vectorA .* vectorB);
end

elapsed_time2 = toc; % 経過時間を計算

setround(-1); % 丸めモードを下方向に設定

tic; % 計算開始時間を記録

% 下方向の丸めモードで内積を計算
for run = 1:RUNS
    lower_bound = sum(vectorA .* vectorB);
end

elapsed_time3 = toc; % 経過時間を計算

setround(1); % 丸めモードを上方向に設定

tic; % 計算開始時間を記録

% 上方向の丸めモードで内積を計算
for run = 1:RUNS
    upper_bound = sum(vectorA .* vectorB);
end

elapsed_time4 = toc; % 経過時間を計算

disp(['Default    : ', sprintf('%.25f',default)]); % デフォルトの丸めモードでの内積の結果を表示

disp(['Near       : ', sprintf('%.25f',near)]); % 最も近い値の丸めモードでの内積の結果を表示

disp(['Lower bound: ', sprintf('%.25f',lower_bound)]);
% 下方向の丸めモードでの内積の結果を表示
```

```
disp(['Upper bound: ', sprintf('%.25f',upper_bound)]); % 上方向の丸めモード
での内積の結果を表示

disp(['Average elapsed time1: ',
     sprintf('%.10f', (elapsed_time1 / RUNS) * 1000), ' ms']); % デフォルト
の丸めモードでの計算時間を表示

disp(['Average elapsed time2: ',
     sprintf('%.10f', (elapsed_time2 / RUNS) * 1000), ' ms']); % 最も近い値
の丸めモードでの計算時間を表示

disp(['Average elapsed time3: ',
     sprintf('%.10f', (elapsed_time3 / RUNS) * 1000), ' ms']); % 下方向の丸
めモードでの計算時間を表示

disp(['Average elapsed time4: ',
     sprintf('%.10f', (elapsed_time4 / RUNS) * 1000), ' ms']); % 上方向の丸
めモードでの計算時間を表示
```

実験結果 (1)　ベクトル A とベクトル B の要素を以下に示す.

$$\text{vectorA} = [34.88, -19.59, -35.99, -68.27, 62.03, -94.03, -23.40, 33.90, -48.42, 84.51]$$

$$\text{vectorB} = [-59.21, -64.66, 27.00, -24.50, -86.04, 6.43, -91.92, -11.56, 70.51, 77.32]$$

上記のベクトルのときの実験結果を表 4 に示す.

		C 言語
計算結果	最近点丸め	$-1160.08150000000018244463717564$
	下向き丸め	$-1160.08150000000029613147489726$
	上向き丸め	$-1160.08149999999981864675646647$
	最近点と下向きの差	$-1.1368683772162 \times 10^{-12}$
	最近点と上向きの差	$3.6379788070917 \times 10^{-12}$
	上向きと下向きの差	$4.7748471843079 \times 10^{-12}$
平均実行時間 [ms]	最近点丸め	0.0000350184
	下向き丸め	0.0000347843
	上向き丸め	0.0000349017
		MATLAB
計算結果	最近点丸め	$-1160.08150000000020518200471997$
	下向き丸め	$-1160.08150000000038708094507455$
	上向き丸め	$-1160.08149999999975043465383350$
	最近点と下向きの差	$-1.8189894035458 \times 10^{-12}$
	最近点と上向きの差	$4.5474735088647 \times 10^{-12}$
	上向きと下向きの差	$6.3664629124105 \times 10^{-12}$
平均実行時間 [ms]	最近点丸め	0.0000576217
	下向き丸め	0.0000562240
	上向き丸め	0.0000564383

表 4: C 言語と MATLAB によるそれぞれの丸め方での内積計算の計算結果と平均実行時間（1）

実験結果（2）　ベクトル A とベクトル B の要素を以下に示す.

$$\text{vectorA} = [-58.51, 19.07, 87.96, -16.82, -59.37, -14.44, -92.16, -83.91, 59.54, 0.27]$$

$$\text{vectorB} = [47.85, -32.31, -41.55, -79.71, 76.46, 76.16, 26.25, -80.73, 12.43, 87.53]$$

上記のベクトルのときの実験結果を表 5 に示す.

		C 言語
計算結果	最近点丸め	$-6250.481999999999708961695432$
	下向き丸め	$-6250.482000000000451836967840 79$
	上向き丸め	$-6250.481999999992649385553598$
	最近点と下向きの差	$-4.54747350886470 \times 10^{-12}$
	最近点と上向きの差	$7.2759576141834 \times 10^{-12}$
	上向きと下向きの差	$1.18234311230481 \times 10^{-11}$
平均実行時間 [ms]	最近点丸め	0.0000347807
	下向き丸め	0.0000343967
	上向き丸め	0.0000342681
		MATLAB
計算結果	最近点丸め	$-6250.481999999999708961695432$
	下向き丸め	$-6250.482000000000451836967840 79$
	上向き丸め	$-6250.481999999992649385553598$
	最近点と下向きの差	$-4.54747350886470 \times 10^{-12}$
	最近点と上向きの差	$7.2759576141834 \times 10^{-12}$
	上向きと下向きの差	$1.18234311230481 \times 10^{-11}$
平均実行時間 [ms]	最近点丸め	0.0000574047
	下向き丸め	0.0000561914
	上向き丸め	0.0000578450

表 5: C 言語と MATLAB によるそれぞれの丸め方での内積計算の計算結果と平均実行時間（2）

実験結果（3）　ベクトル A とベクトル B の要素を以下に示す.

$$\mathrm{vectorA} = [-46.35, 2.04, 33.26, 5.00, -51.30, -66.95, 79.29, 53.44, 75.44, -65.90]$$

$$\mathrm{vectorB} = [-36.53, 67.67, -56.77, -36.51, -41.04, -23.22, 59.30, -3.06, -8.20, -35.74]$$

上記のベクトルのときの実験結果を表 6 に示す.

		C 言語
計算結果	最近点丸め	9695.45169999999996187398210168
	下向き丸め	9695.45169999999977997504174710
	上向き丸め	9695.45170000000032567186281086
	最近点と下向きの差	$-1.81898940354580 \times 10^{-12}$
	最近点と上向きの差	$3.63797880709180 \times 10^{-12}$
	上向きと下向きの差	$5.45696821063760 \times 10^{-12}$
平均実行時間 [ms]	最近点丸め	0.0000346329
	下向き丸め	0.0000344090
	上向き丸め	0.0000342720
		MATLAB
計算結果	最近点丸め	9695.45170000000014377292245627
	下向き丸め	9695.45169999999977997504174710
	上向き丸め	9695.45170000000032567186281086
	最近点と下向きの差	$-3.63797880709170 \times 10^{-12}$
	最近点と上向きの差	$1.81898940354590 \times 10^{-12}$
	上向きと下向きの差	$5.45696821063760 \times 10^{-12}$
平均実行時間 [ms]	最近点丸め	0.0000568081
	下向き丸め	0.0000561438
	上向き丸め	0.0000559168

表 6: C 言語と MATLAB によるそれぞれの丸め方での内積計算の計算結果と平均実行時間（3）

考察　本実験では，C 言語と MATLAB で実装した浮動小数点数の丸めモードによる計算結果の違いについて調査した．その結果，同じベクトルに対しても，丸めモードによって計算結果が同じになる場合と異なる場合があることが確認された．この現象は，浮動小数点数の丸めモードが計算結果に影響を与えることに起因する．浮動小数点数の計算では，有限の精度により丸め誤差が生じる．そして，丸めモードはこの丸め誤差の処理方法を決定する．したがって，丸めモードによって，同じ計算でも微妙に異なる結果が得られる．また，ベクトルによっては，C 言語と MATLAB で同じ計算結果になることもあれば，異なることもある．さらには，一部の丸めモードだけ同じ結果になることもあるということが実験結果からわかる．以上の考察から，浮動小数点数の丸めモードが計算結果に影響を与え，その影響は計算の内容や使用する数値によって変わることが理解できる．

C　C 言語による多倍長精度の機械区間演算 (付録)

C.1　点区間同士による多倍長精度の四則演算や初等関数のソースコード (実数)

```
#include<isys.h> // 独自のヘッダーファイルをインクルード
#include "mi.c" // 独自のソースファイルをインクルード

int main() {
    // 使用する変数の宣言
    rmulti *a, *b, *diff_up_down, *diff; // rmulti 型のポインタ変数の宣言
    mi_r *c; // mi_r 型のポインタ変数の宣言
    int diff_exp_prec; // 整数型の変数の宣言

    // 精度を 1 から 1000 まで動かして，ループを実行
    for(int i = 1; i < 1001; i++) {

        set_default_prec(i); // デフォルトの精度を設定

        // メモリの確保
        a = rallocate(); // rmulti 型の変数のメモリを確保
        b = rallocate();
        c = mi_rallocate(); // mi_r 型の変数のメモリを確保
        diff_up_down = rallocate();
        diff = rallocate();
```

```
        // 基本演算関数
        mpfr_ui_pow_ui (a, 3, 628, MPFR_RNDN); // 3の628乗を最近点に丸め, a
に格納
        mpfr_sqrt_ui (b, 5, MPFR_RNDN); // 5の平方根を最近点に丸め, bに格納

        // 点区間同士の機械区間演算
        mir_rr_add(c, a, b); // 点区間aとbの和を区間cに格納

        // 相対誤差を計算
        mpfr_sub(diff_up_down, c->up, c->down, MPFR_RNDN);

        diff_exp_prec = mpfr_get_exp(c->up) - mpfr_get_prec(c->up);

        rdiv_2exp(diff, diff_up_down, diff_exp_prec);

        // 結果を標準出力
        printf(output_file, "precision = %d\n", i); // 精度を出力
        mpfr_printf(output_file,
         "(up-down) / 2^(exp-prec) = %50.10Rf\t\n", diff); // 相対誤差を出力


        // メモリの解放
        a = rmfree(a); // rmulti型の変数のメモリを解放
        b = rmfree(b);
        c = mi_rfree(c); // mi_r型の変数のメモリを解放
        diff_up_down = rmfree(diff_up_down);
        diff = rmfree(diff);
    }

    return 0; // プログラムの終了
}
```

C.2　精度に対する相対誤差の表とグラフ（実数）

C.2.1　加算（$x + y$）

第 4.1 節の多倍長精度の加算（実数）の実験結果を図 2 と表 7 に示す.

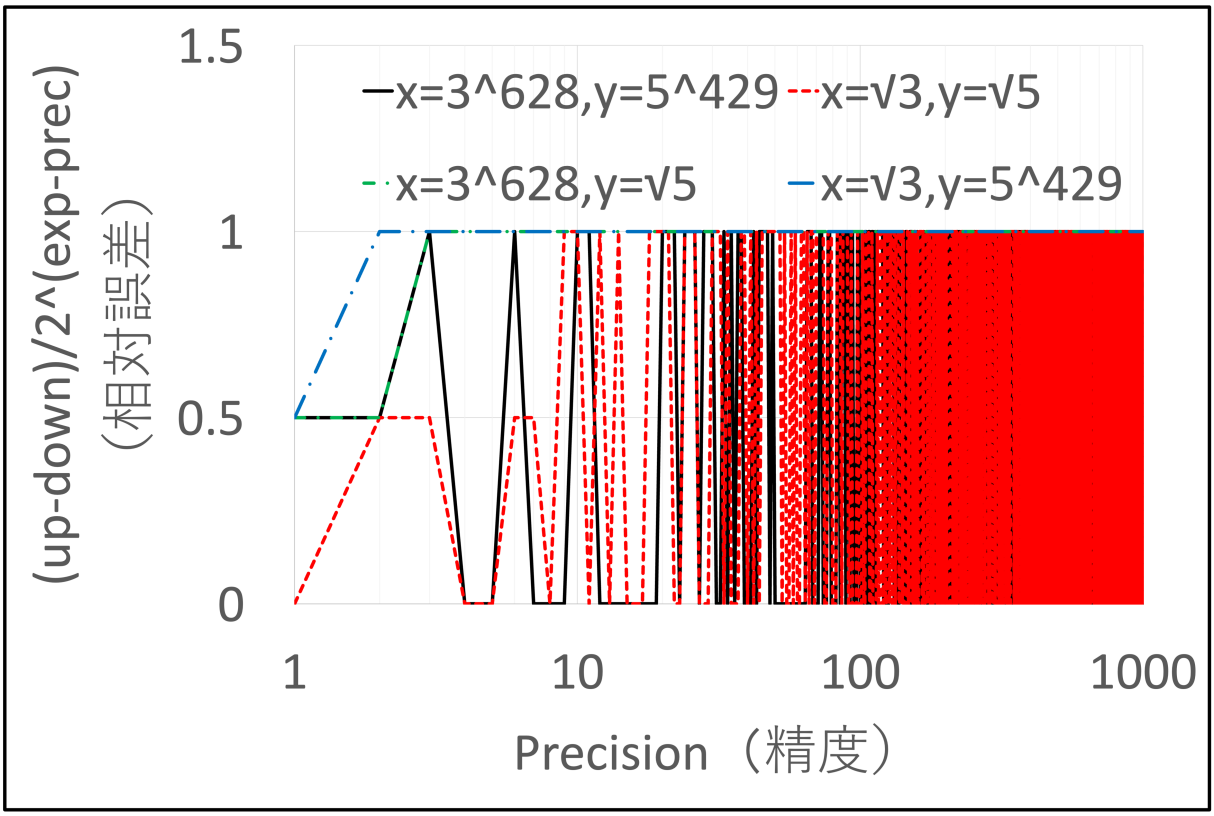

図 2: 精度に対する相対誤差の片対数グラフ（実数の加算）

x \ y	L^{300}_{+151}	S^{301}_{-150}	$-L^{300}_{+151}$	$-S^{301}_{-150}$	$\sqrt{3}$
L^{300}_{+150}	1.0	1.0	1.0	1.0	1.0
S^{300}_{-150}	1.0	1.0	1.0	0.0	1.0
$-L^{300}_{+150}$	1.0	1.0	1.0	1.0	1.0
$-S^{300}_{-150}$	1.0	0.0	1.0	1.0	1.0
$\sqrt{2}$	1.0	1.0	1.0	1.0	1.0

表 7: 精度が 1 から 1000 のときの相対誤差の最大値（実数の加算）

C.2.2 減算（$x - y$）

第 4.1 節の多倍長精度の減算（実数）の実験結果を図 3 と表 8 に示す.

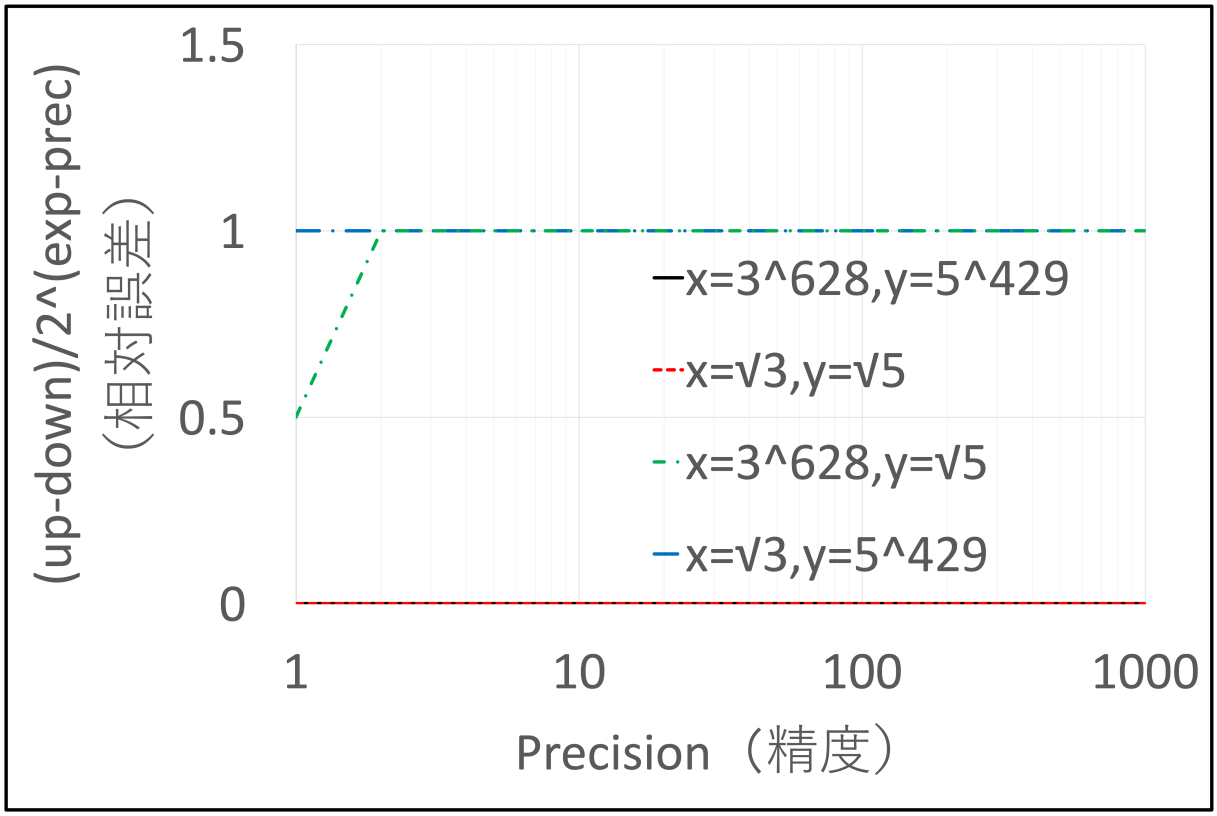

図 3: 精度に対する相対誤差の片対数グラフ（実数の減算）

x \ y	L^{300}_{+151}	S^{301}_{-150}	$-L^{300}_{+151}$	$-S^{301}_{-150}$	$\sqrt{3}$
L^{300}_{+150}	1.0	1.0	1.0	1.0	1.0
S^{300}_{-150}	1.0	0.0	1.0	1.0	1.0
$-L^{300}_{+150}$	1.0	1.0	1.0	1.0	1.0
$-S^{300}_{-150}$	1.0	1.0	1.0	0.0	1.0
$\sqrt{2}$	1.0	1.0	1.0	1.0	0.0

表 8: 精度が 1 から 1000 のときの相対誤差の最大値（実数の減算）

C.2.3 乗算（$x \times y$）

第 4.1 節の多倍長精度の乗算（実数）の実験結果を図 4 と表 9 に示す.

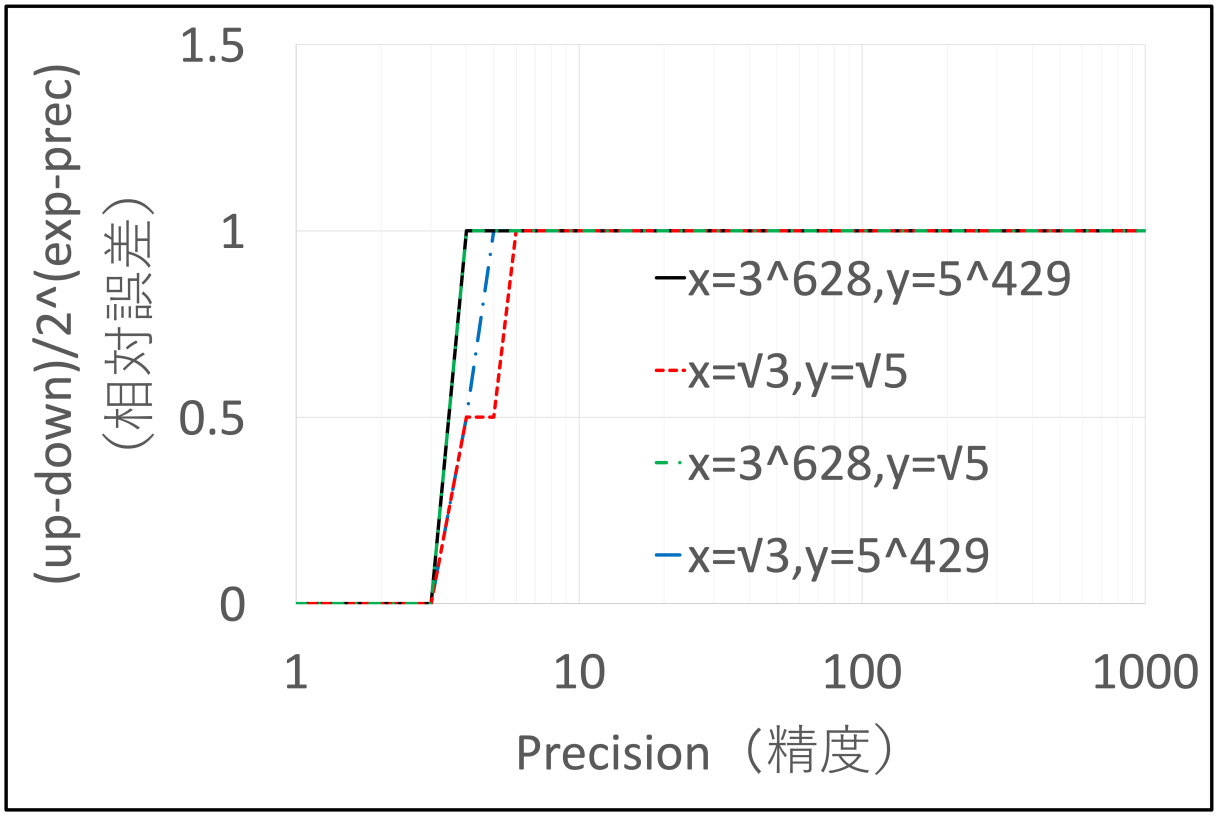

図 4: 精度に対する相対誤差の片対数グラフ（実数の乗算）

x \ y	L_{+76}^{300}	S_{-75}^{301}	$-L_{+76}^{300}$	$-S_{-75}^{301}$	$\sqrt{3}$
L_{+75}^{300}	1.0	1.0	1.0	1.0	1.0
S_{-75}^{300}	1.0	1.0	1.0	1.0	1.0
$-L_{+75}^{300}$	1.0	1.0	1.0	1.0	1.0
$-S_{-75}^{300}$	1.0	1.0	1.0	1.0	1.0
$\sqrt{2}$	1.0	1.0	1.0	1.0	1.0

表 9: 精度が 1 から 1000 のときの相対誤差の最大値（実数の乗算）

C.2.4 除算（x/y）

第 4.1 節の多倍長精度の除算（実数）の実験結果を図 5 と表 10 に示す.

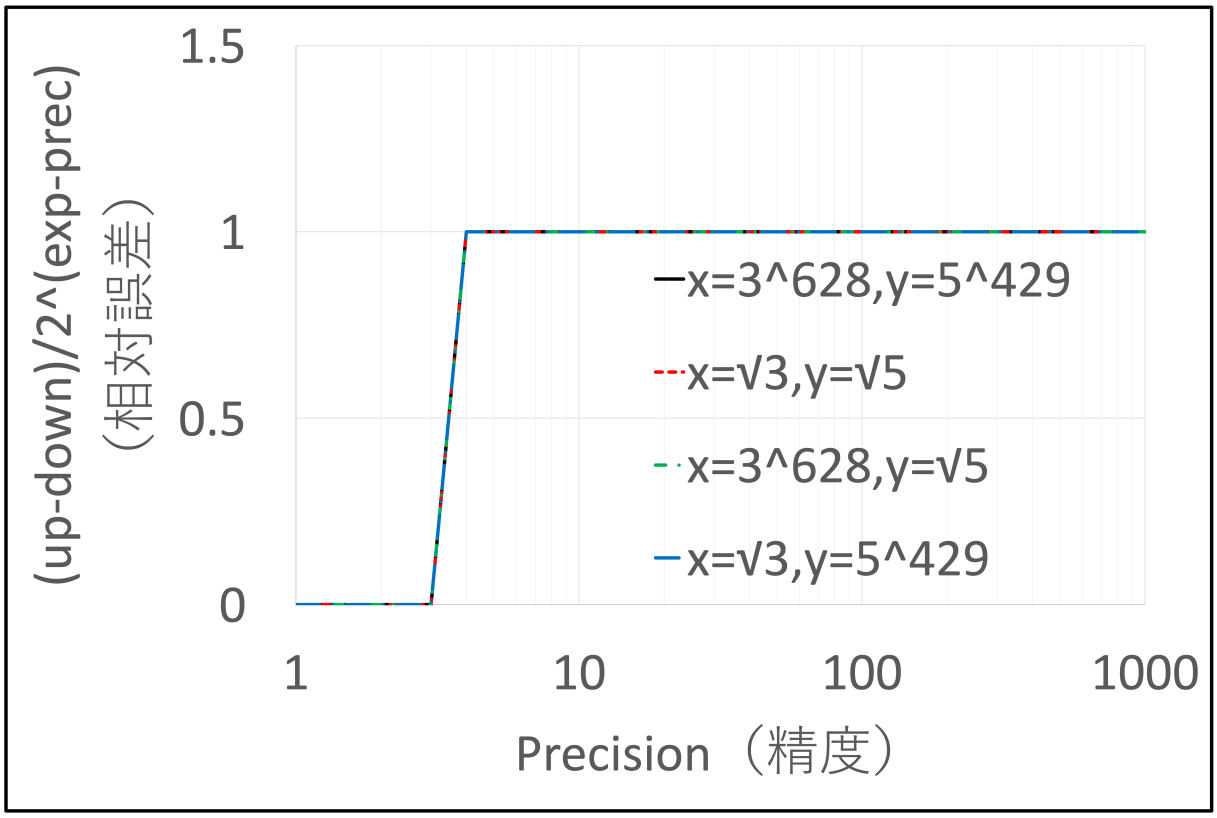

図 5: 精度に対する相対誤差の片対数グラフ（実数の除算）

x \ y	L_{+76}^{300}	S_{-75}^{301}	$-L_{+76}^{300}$	$-S_{-75}^{301}$	$\sqrt{3}$
L_{+75}^{300}	1.0	1.0	1.0	1.0	1.0
S_{-75}^{300}	1.0	1.0	1.0	1.0	1.0
$-L_{+75}^{300}$	1.0	1.0	1.0	1.0	1.0
$-S_{-75}^{300}$	1.0	1.0	1.0	1.0	1.0
$\sqrt{2}$	1.0	1.0	1.0	1.0	1.0

表 10: 精度が 1 から 1000 のときの相対誤差の最大値（実数の除算）

C.2.5　平方根（$\sqrt{x}$）

第 4.2 節の多倍長精度の平方根（実数）の実験結果を図 6 と表 11 に示す.

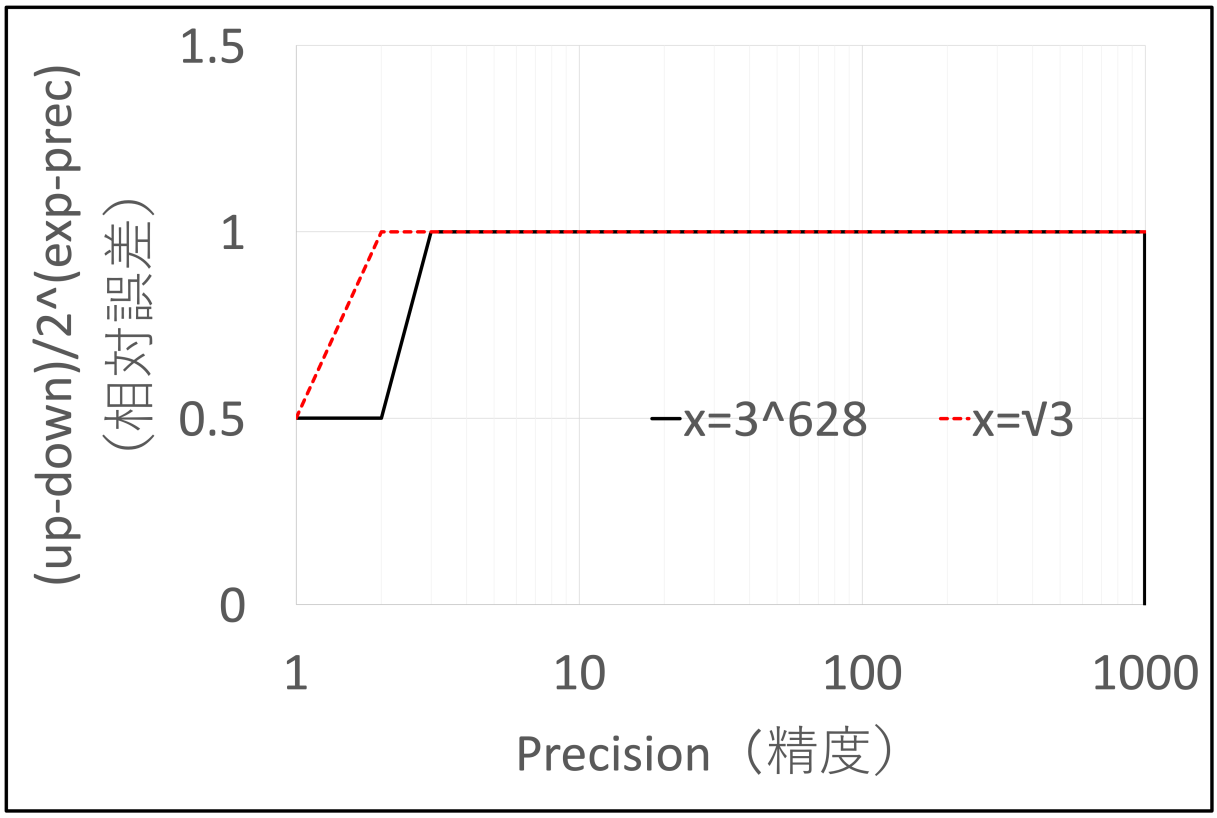

図 6: 精度に対する相対誤差の片対数グラフ（実数の平方根）

x	L^{300}_{+150}	S^{300}_{-150}	$\sqrt{2}$
相対誤差	1.0	1.0	1.0

表 11: 精度が 1 から 1000 のときの相対誤差の最大値（実数の平方根）

C.2.6　平方根の逆数（$1/\sqrt{x}$）

第 4.2 節の多倍長精度の平方根の逆数（実数）の実験結果を図 7 と表 12 に示す.

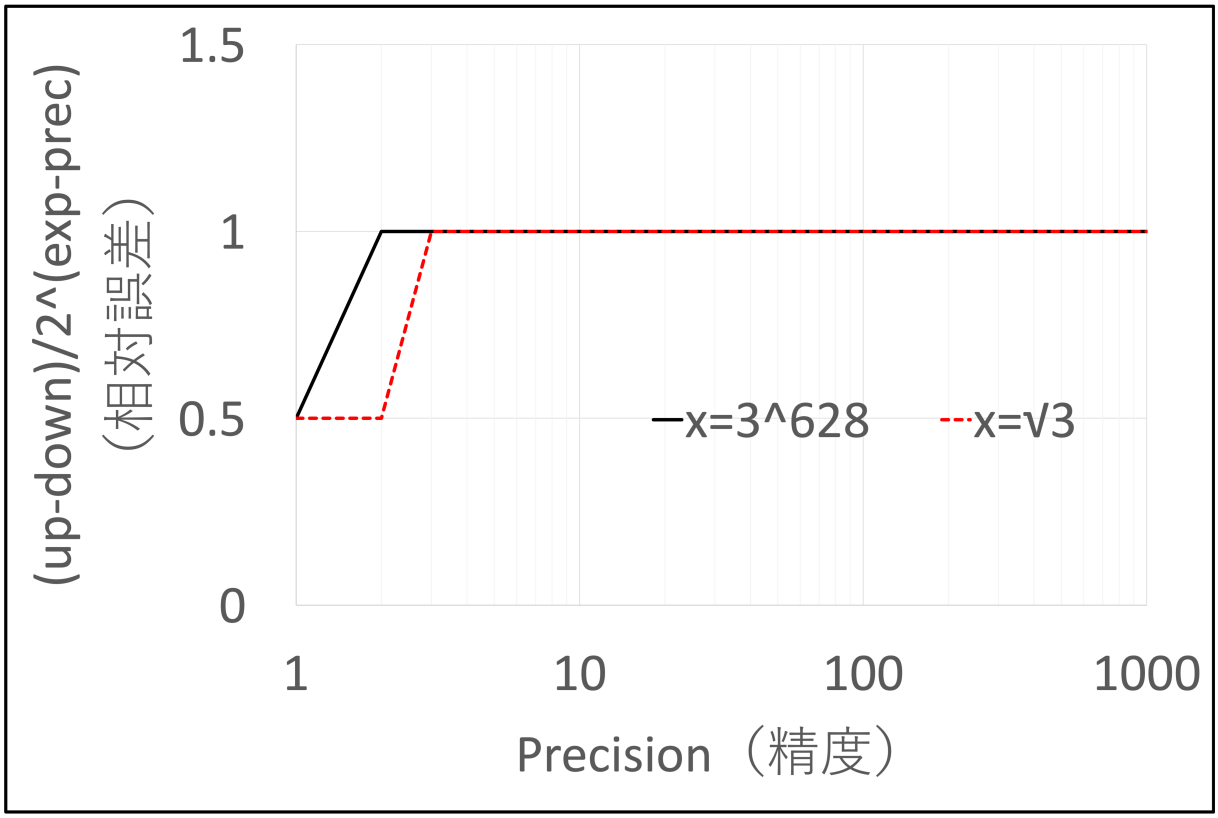

図 7: 精度に対する相対誤差の片対数グラフ（実数の平方根の逆数）

x	L^{300}_{+150}	S^{300}_{-150}	$\sqrt{2}$
相対誤差	1.0	1.0	1.0

表 12: 精度が 1 から 1000 のときの相対誤差の最大値（実数の平方根の逆数）

C.2.7　指数関数（x^y）

第 4.2 節の多倍長精度の指数関数（実数）の実験結果を図 8 と表 13 に示す．図 8 のグラフにおいて，$x = 3^{628}, y = 5^{429}$ のときと，$x = \sqrt{3}, y = 5^{429}$ のときは segmentation fault により記録不可であった．また，表 13 において，$\times$ と書いてある部分は segmentation fault により記録不可であった．

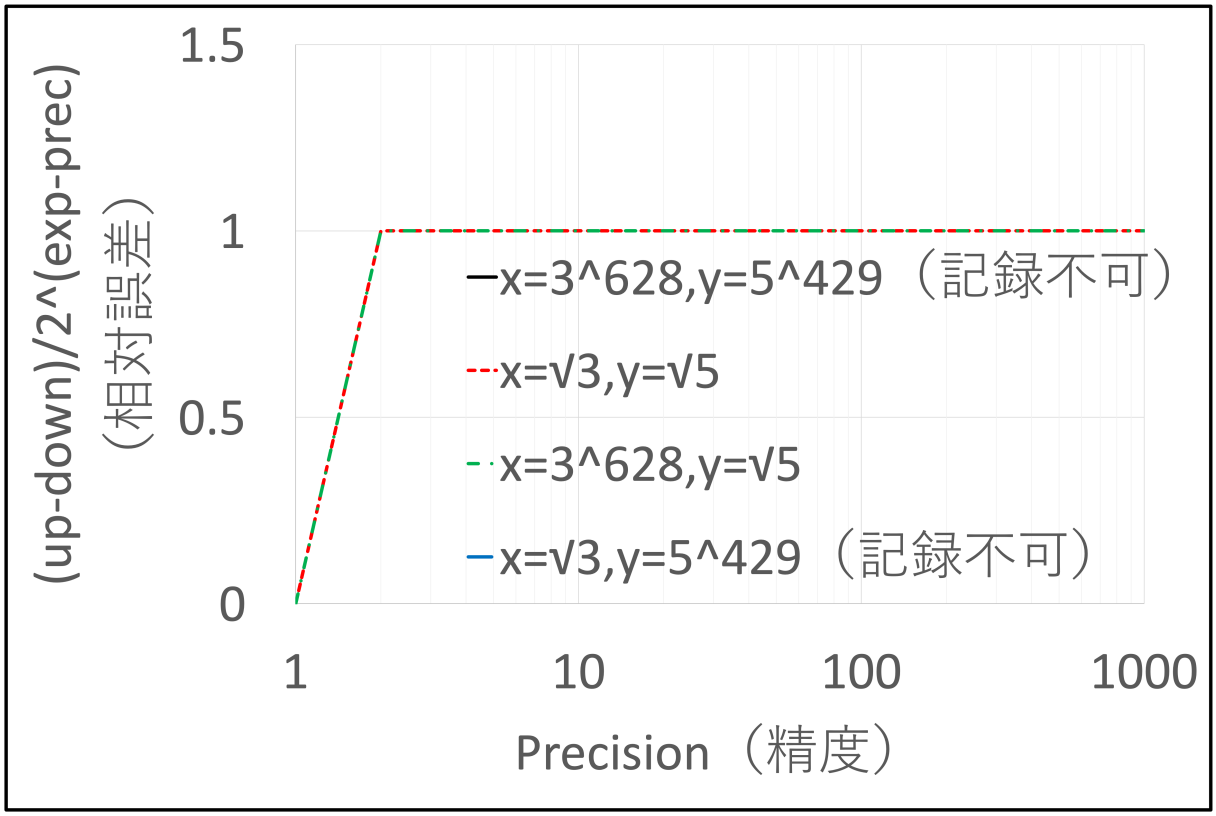

図 8: 精度に対する相対誤差の片対数グラフ（実数の指数関数）

x \ y	L^{300}_{+76}	S^{301}_{-75}	$-L^{300}_{+76}$	$-S^{301}_{-75}$	$\sqrt{3}$
L^{300}_{+75}	$\times$	1.0	5.4×10^{300}	1.0	1.0
S^{300}_{-75}	1.0	1.0	$\times$	1.0	1.0
$\sqrt{2}$	$\times$	1.0	5.4×10^{300}	1.0	1.0

表 13: 精度が 1 から 1000 のときの相対誤差の最大値（実数の指数関数）

C.2.8　自然対数（$\ln x$）

第 4.2 節の多倍長精度の自然対数（実数）の実験結果を図 9 と表 14 に示す.

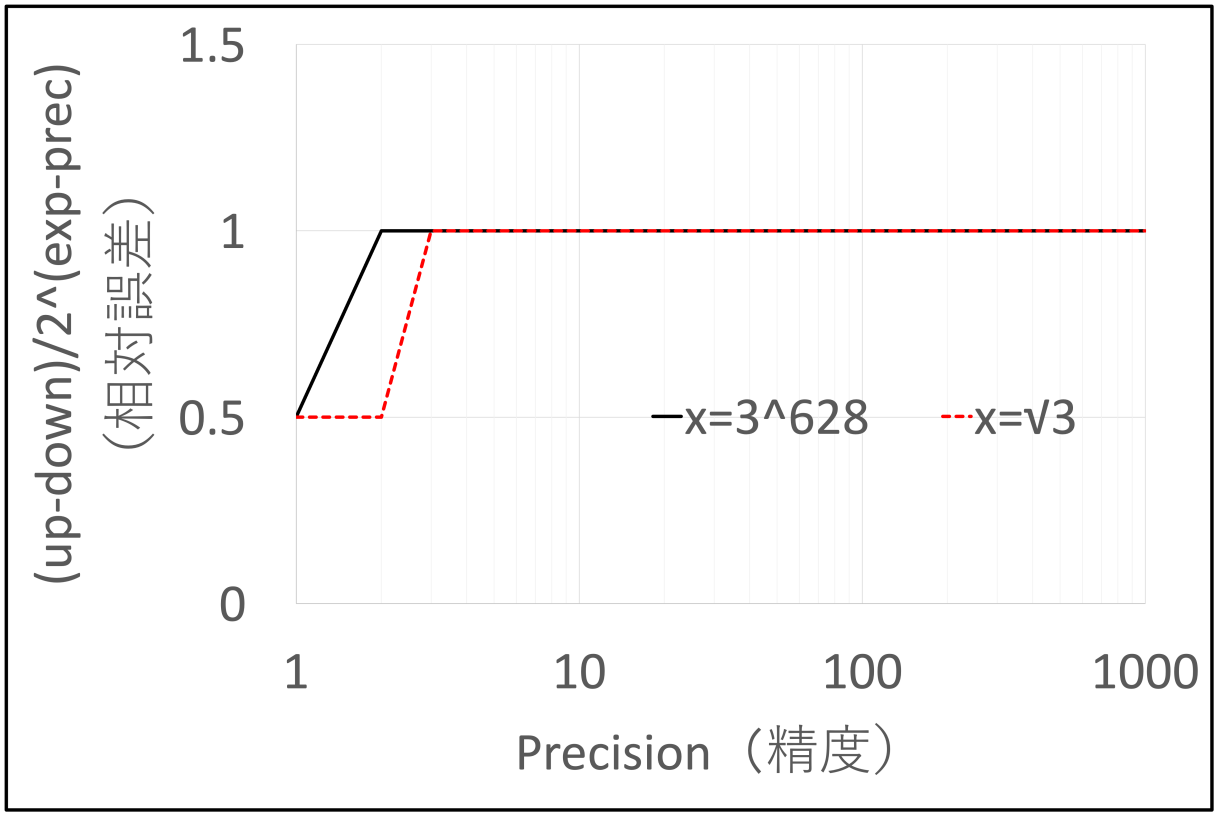

図 9: 精度に対する相対誤差の片対数グラフ（実数の自然対数）

x	L^{300}_{+150}	S^{300}_{-150}	$\sqrt{2}$
相対誤差	1.0	1.0	1.0

表 14: 精度が 1 から 1000 のときの相対誤差の最大値（実数の自然対数）

C.2.9　底が 2 の対数（$\log_2 x$）

第 4.2 節の多倍長精度の底が 2 の対数（実数）の実験結果を図 10 と表 15 に示す.

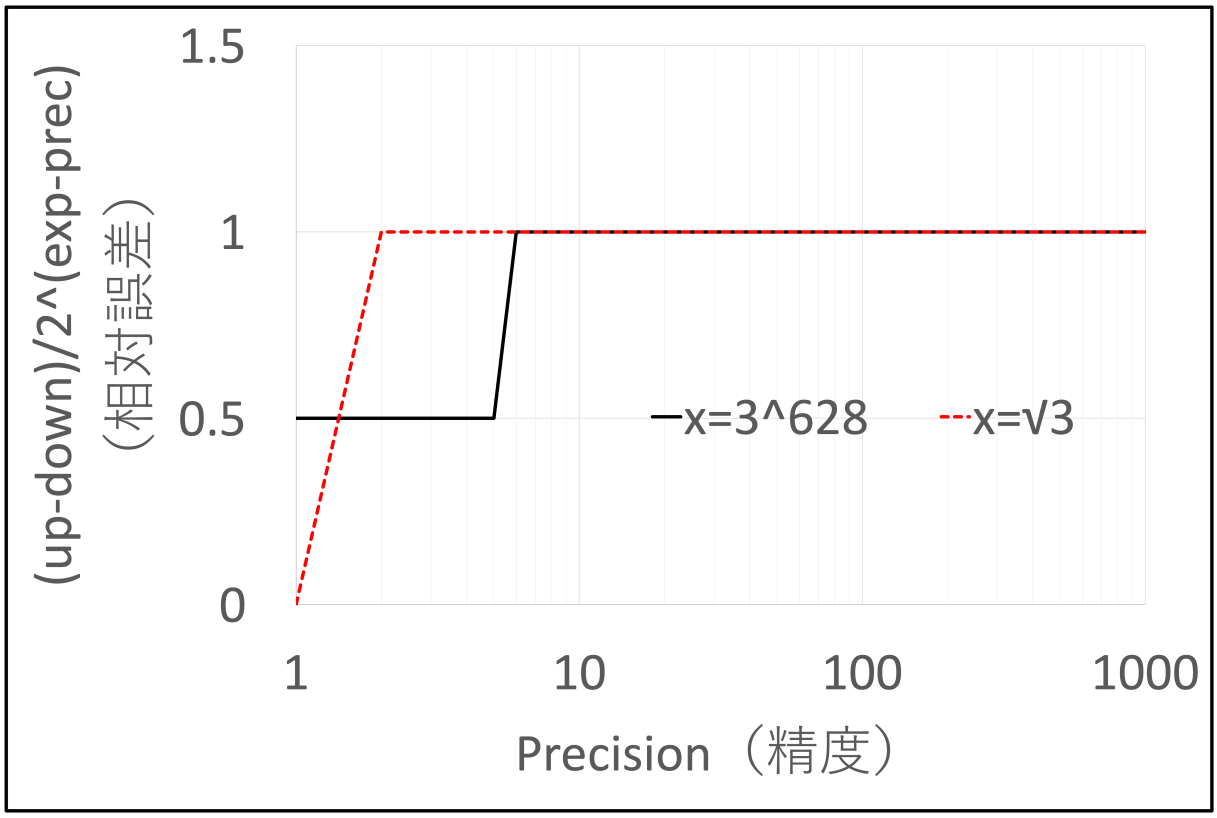

図 10: 精度に対する相対誤差の片対数グラフ（実数の底が 2 の対数）

x	L^{300}_{+150}	S^{300}_{-150}	$\sqrt{2}$
相対誤差	1.0	1.0	1.0

表 15: 精度が 1 から 1000 のときの相対誤差の最大値（実数の底が 2 の対数）

C.2.10　底が 10 の対数（$\log_{10} x$）

第 4.2 節の多倍長精度の底が 10 の対数（実数）の実験結果を図 11 と表 16 に示す.

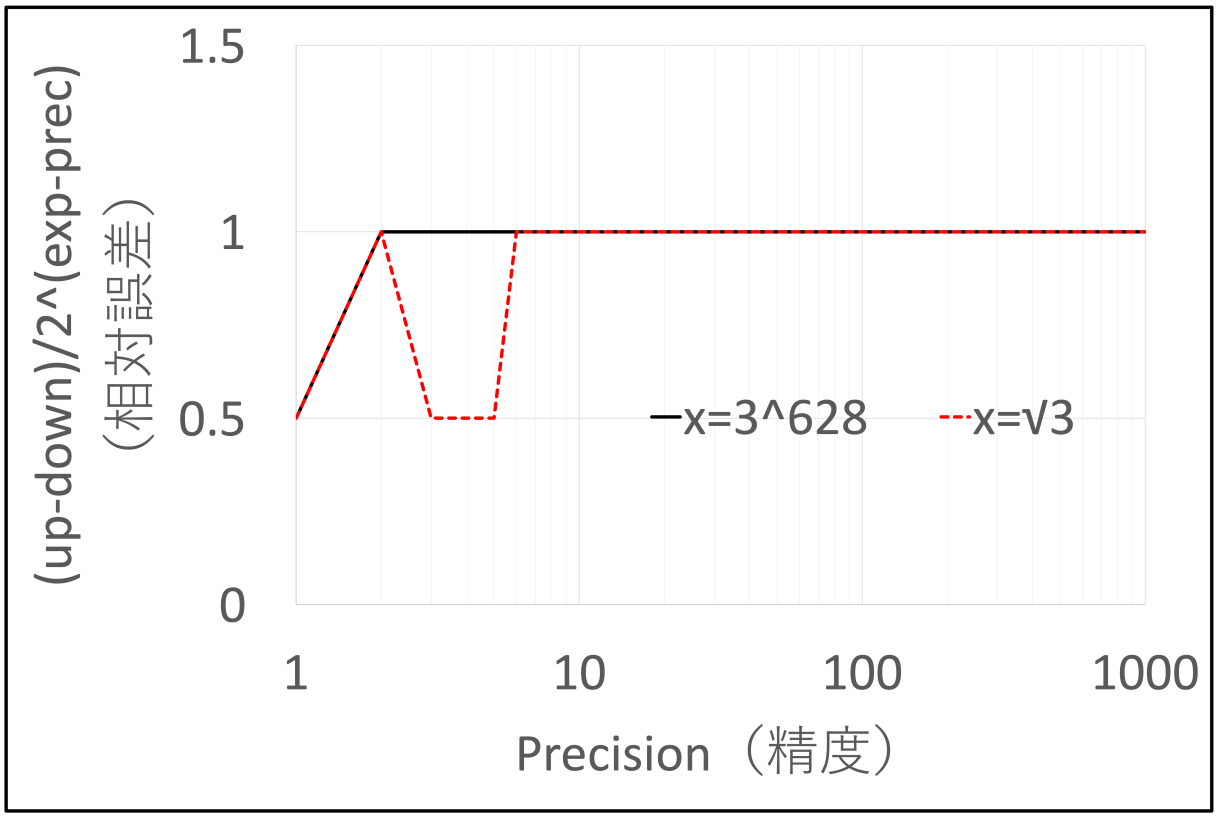

図 11: 精度に対する相対誤差の片対数グラフ（実数の底が 10 の対数）

x	L^{300}_{+150}	S^{300}_{-150}	$\sqrt{2}$
相対誤差	1.0	1.0	1.0

表 16: 精度が 1 から 1000 のときの相対誤差の最大値（実数の底が 10 の対数）

C.2.11 e のべき乗（e^x）

第 4.2 節の多倍長精度の e のべき乗（実数）の実験結果を図 12 と表 17 に示す．図 12 のグラフにおいて，$x = 3^{628}$ のときは segmentation fault により記録不可であった．また，表 17 において，$\times$ と書いてある部分は segmentation fault により記録不可であった．

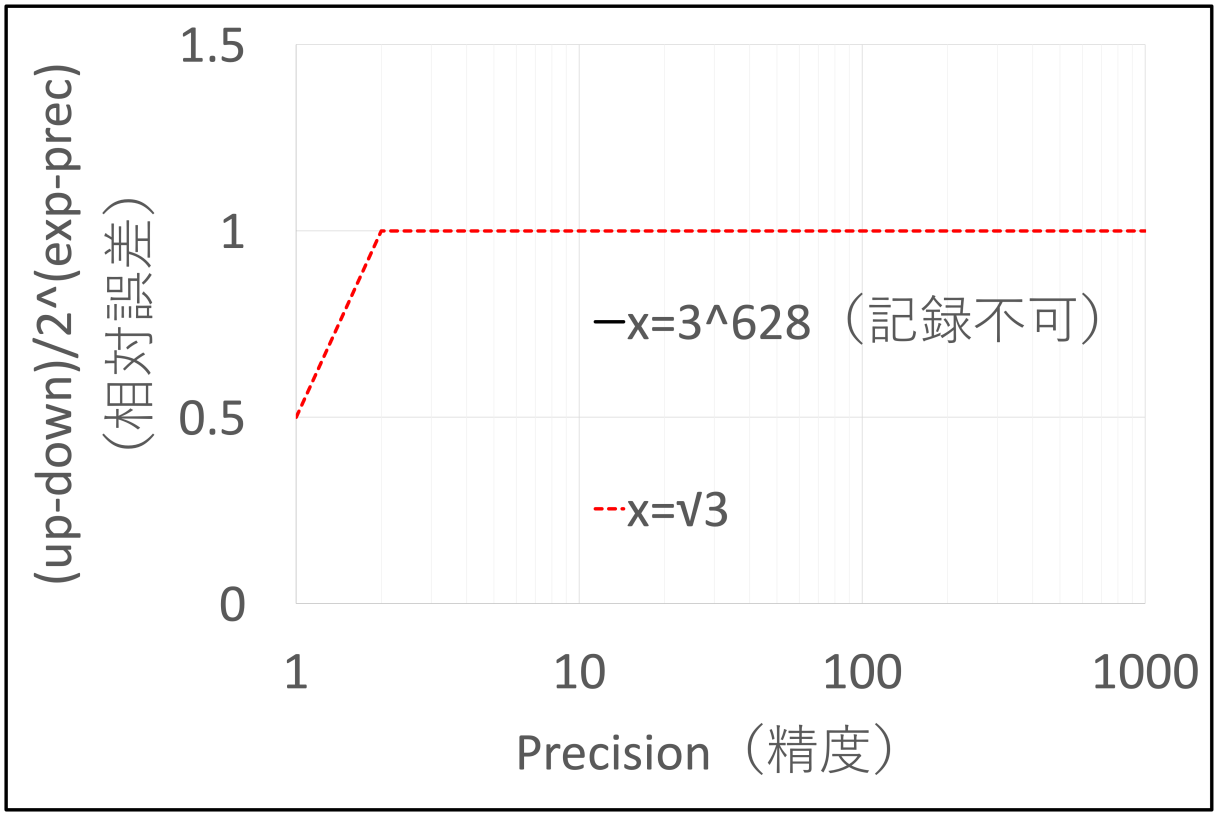

図 12: 精度に対する相対誤差の片対数グラフ（実数の e のべき乗）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	$\times$	1.0	5.4×10^{300}	1.0	1.0

表 17: 精度が 1 から 1000 のときの相対誤差の最大値（実数の e のべき乗）

C.2.12　2 のべき乗（2^x）

第 4.2 節の多倍長精度の 2 のべき乗（実数）の実験結果を図 13 と表 18 に示す．図 13 のグラフにおいて，$x = 3^{628}$ のときは segmentation fault により記録不可であった．また，表 18 において，$\times$ と書いてある部分は segmentation fault により記録不可であった．

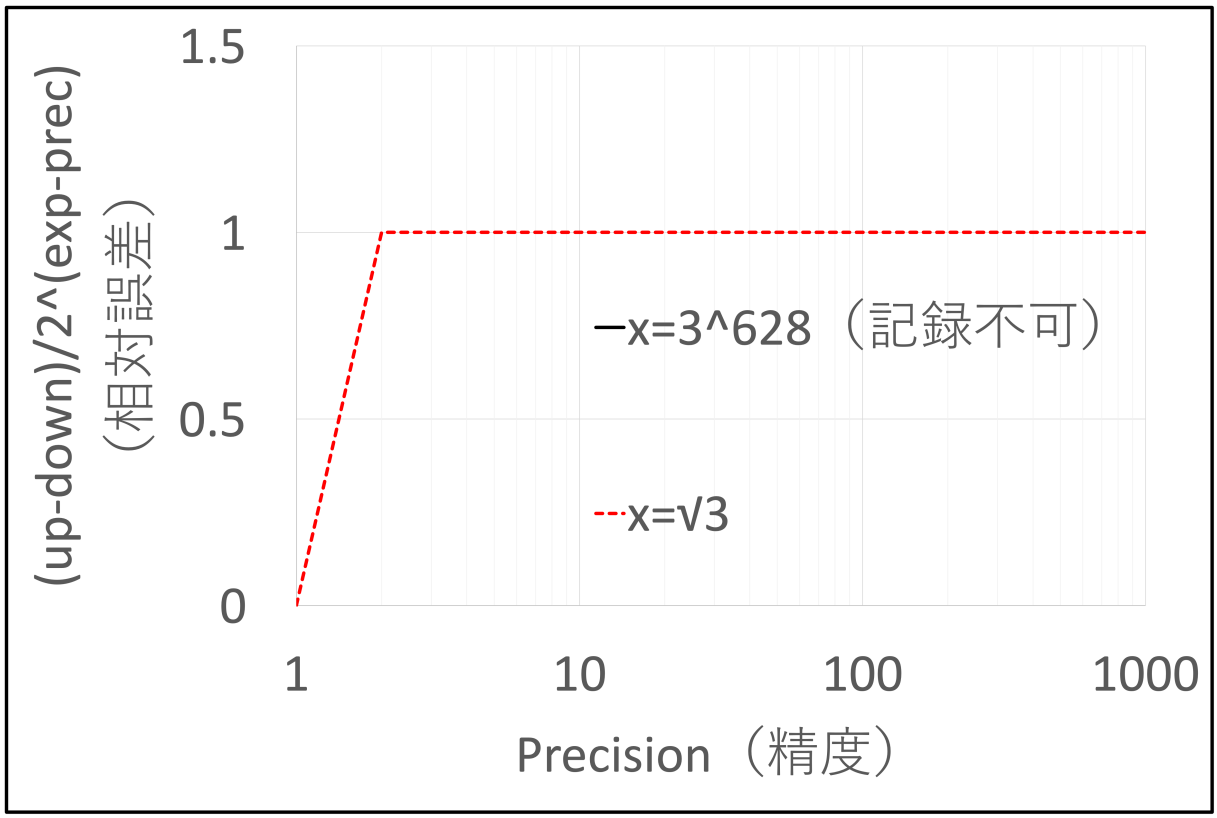

図 13: 精度に対する相対誤差の片対数グラフ（実数の 2 のべき乗）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	$\times$	1.0	5.4×10^{300}	1.0	1.0

表 18: 精度が 1 から 1000 のときの相対誤差の最大値（実数の 2 のべき乗）

C.2.13　10 のべき乗（10^x）

第 4.2 節の多倍長精度の 10 のべき乗（実数）の実験結果を図 14 と表 19 に示す．図 14 のグラフにおいて，$x = 3^{628}$ のときは segmentation fault により記録不可であった．また，表 19 において，× と書いてある部分は segmentation fault により記録不可であった．

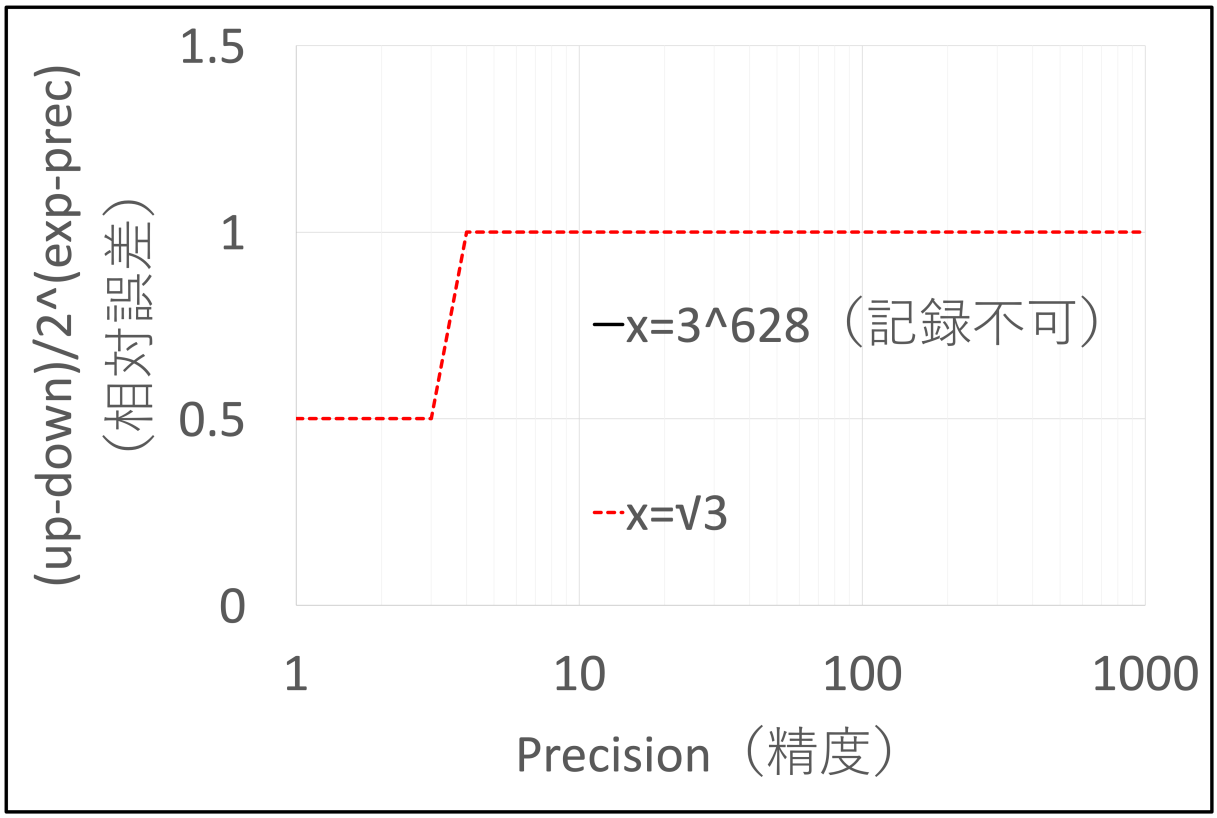

図 14: 精度に対する相対誤差の片対数グラフ（実数の 10 のべき乗）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	×	1.0	5.4×10^{300}	1.0	1.0

表 19: 精度が 1 から 1000 のときの相対誤差の最大値（実数の 10 のべき乗）

C.2.14　余弦（$\cos x$）

第 4.2 節の多倍長精度の余弦（実数）の実験結果を図 15 と表 20 に示す.

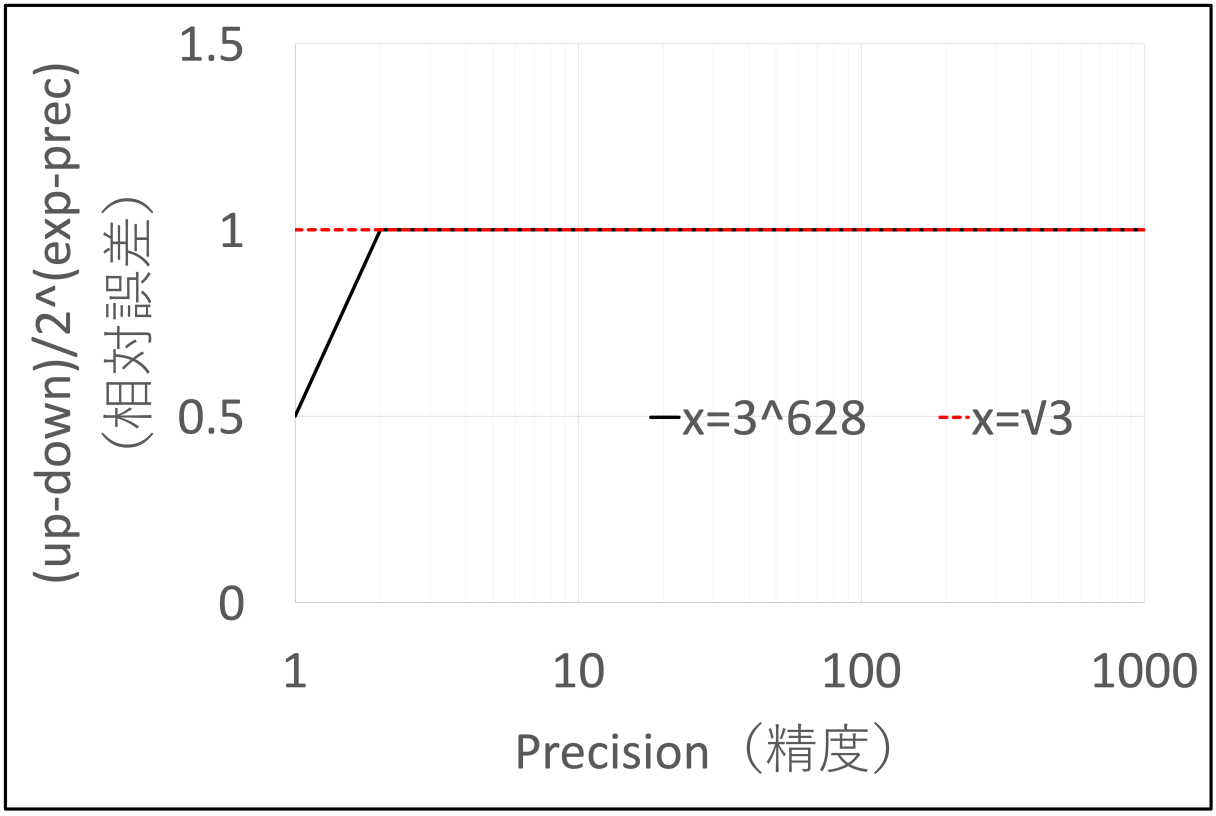

図 15: 精度に対する相対誤差の片対数グラフ（実数の余弦）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	1.0	1.0	1.0	1.0	1.0

表 20: 精度が 1 から 1000 のときの相対誤差の最大値（実数の余弦）

C.2.15　正弦（$\sin x$）

第 4.2 節の多倍長精度の正弦（実数）の実験結果を図 16 と表 21 に示す.

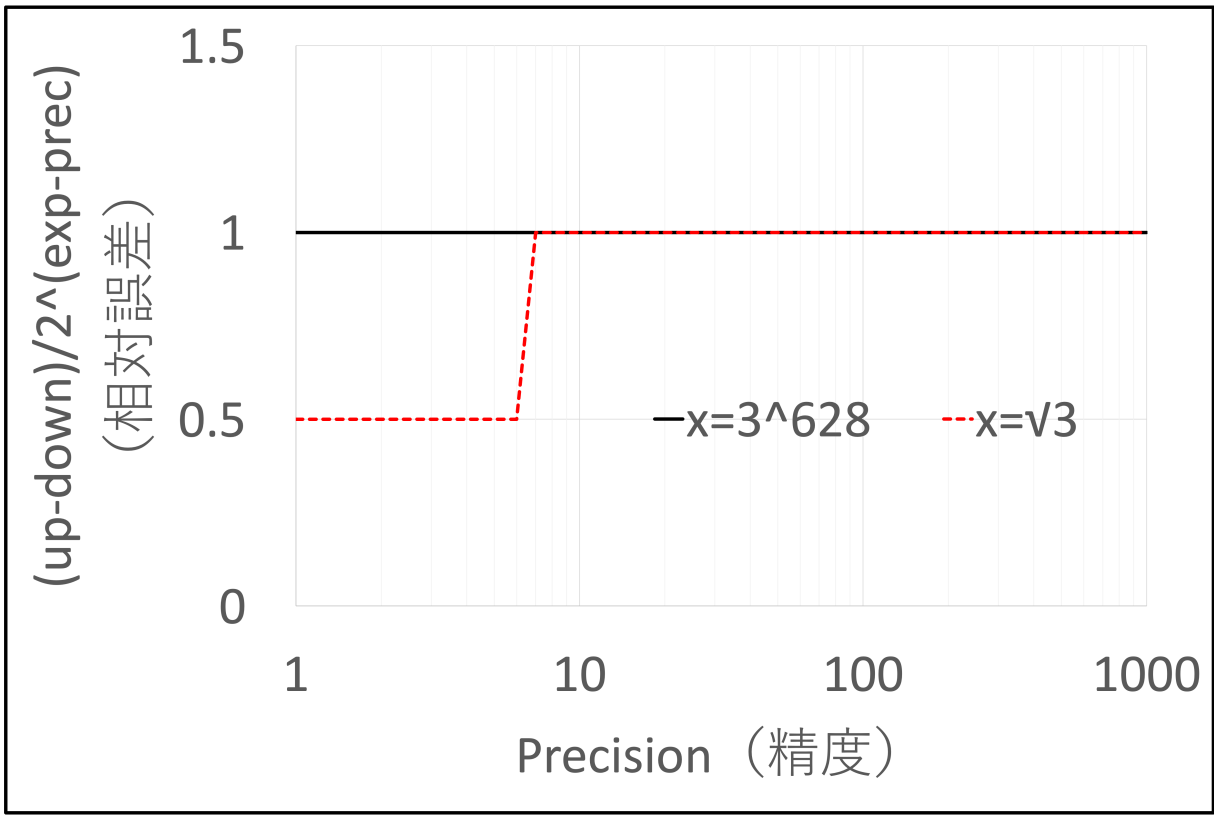

図 16: 精度に対する相対誤差の片対数グラフ（実数の正弦）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	1.0	1.0	1.0	1.0	1.0

表 21: 精度が 1 から 1000 のときの相対誤差の最大値（実数の正弦）

C.2.16　正接（$\tan x$）

第 4.2 節の多倍長精度の正接（実数）の実験結果を図 17 と表 22 に示す.

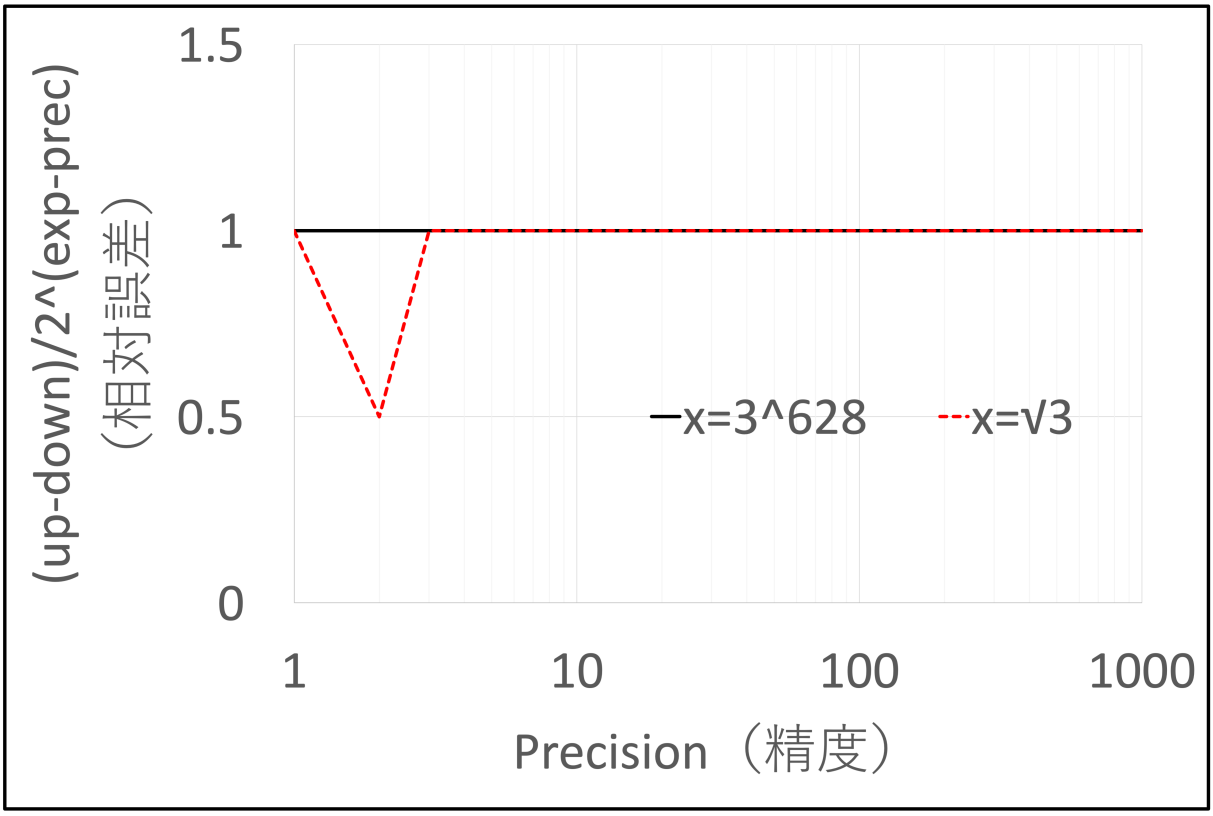

図 17: 精度に対する相対誤差の片対数グラフ（実数の正接）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	1.0	1.0	1.0	1.0	1.0

表 22: 精度が 1 から 1000 のときの相対誤差の最大値（実数の正接）

C.2.17　正割（$\sec x$）

第 4.2 節の多倍長精度の正割（実数）の実験結果を図 18 と表 23 に示す.

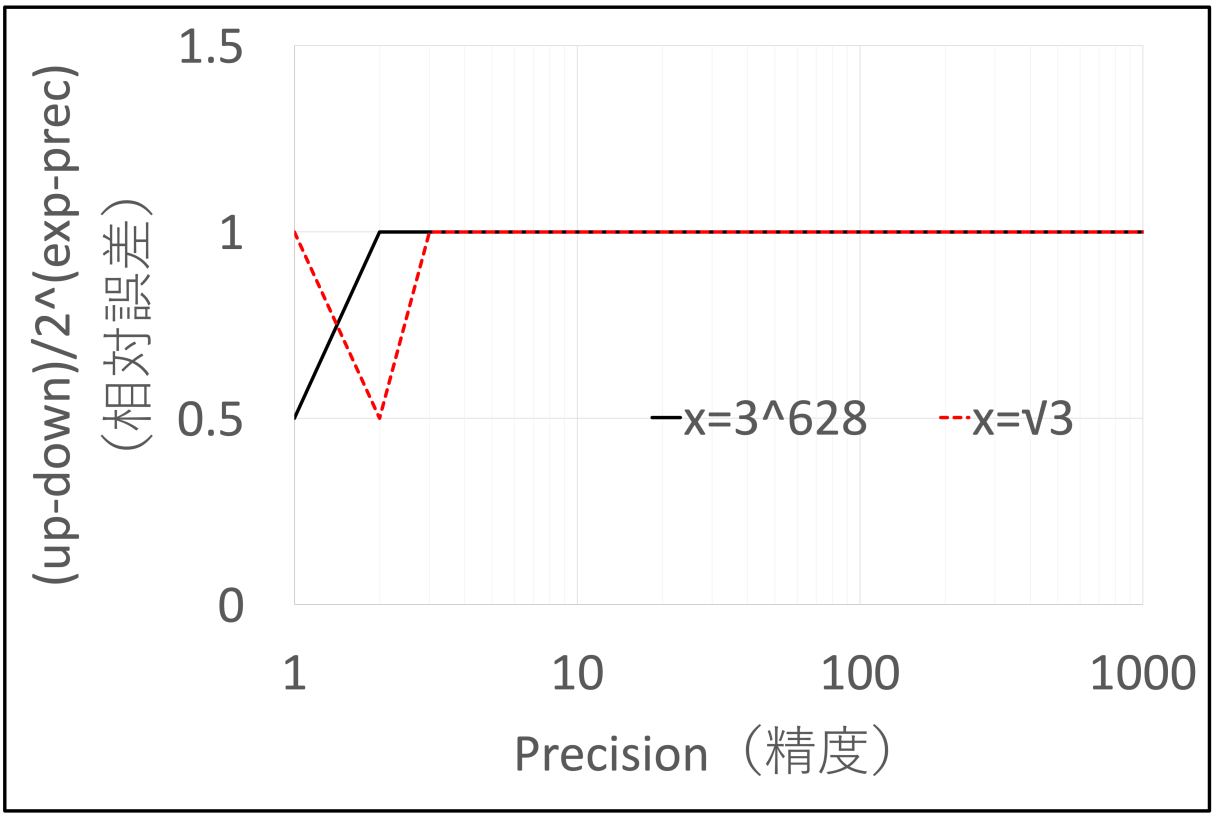

図 18: 精度に対する相対誤差の片対数グラフ（実数の正割）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	1.0	1.0	1.0	1.0	1.0

表 23: 精度が 1 から 1000 のときの相対誤差の最大値（実数の正割）

C.2.18　余割（$\csc x$）

第 4.2 節の多倍長精度の余割（実数）の実験結果を図 19 と表 24 に示す.

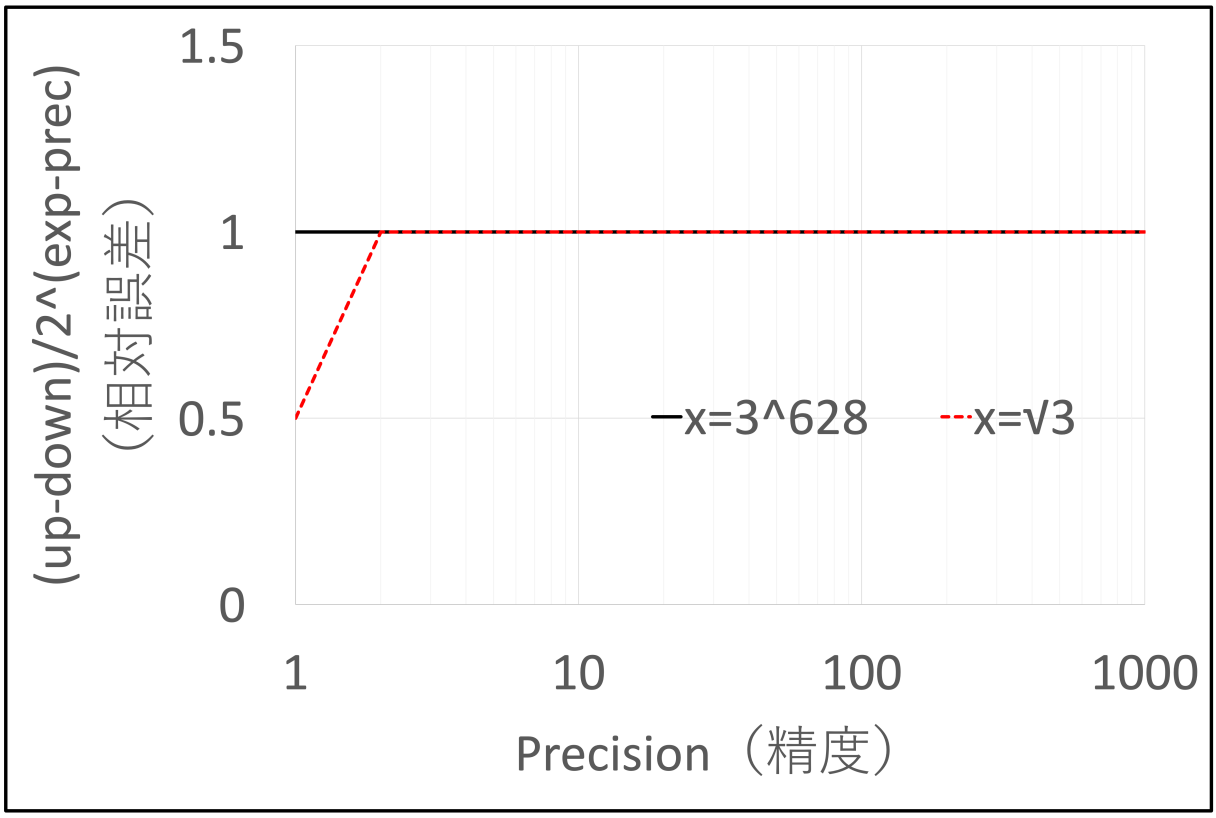

図 19: 精度に対する相対誤差の片対数グラフ（実数の余割）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	1.0	1.0	1.0	1.0	1.0

表 24: 精度が 1 から 1000 のときの相対誤差の最大値（実数の余割）

C.2.19　余接（$\cot x$）

第 4.2 節の多倍長精度の余接（実数）の実験結果を図 20 と表 25 に示す.

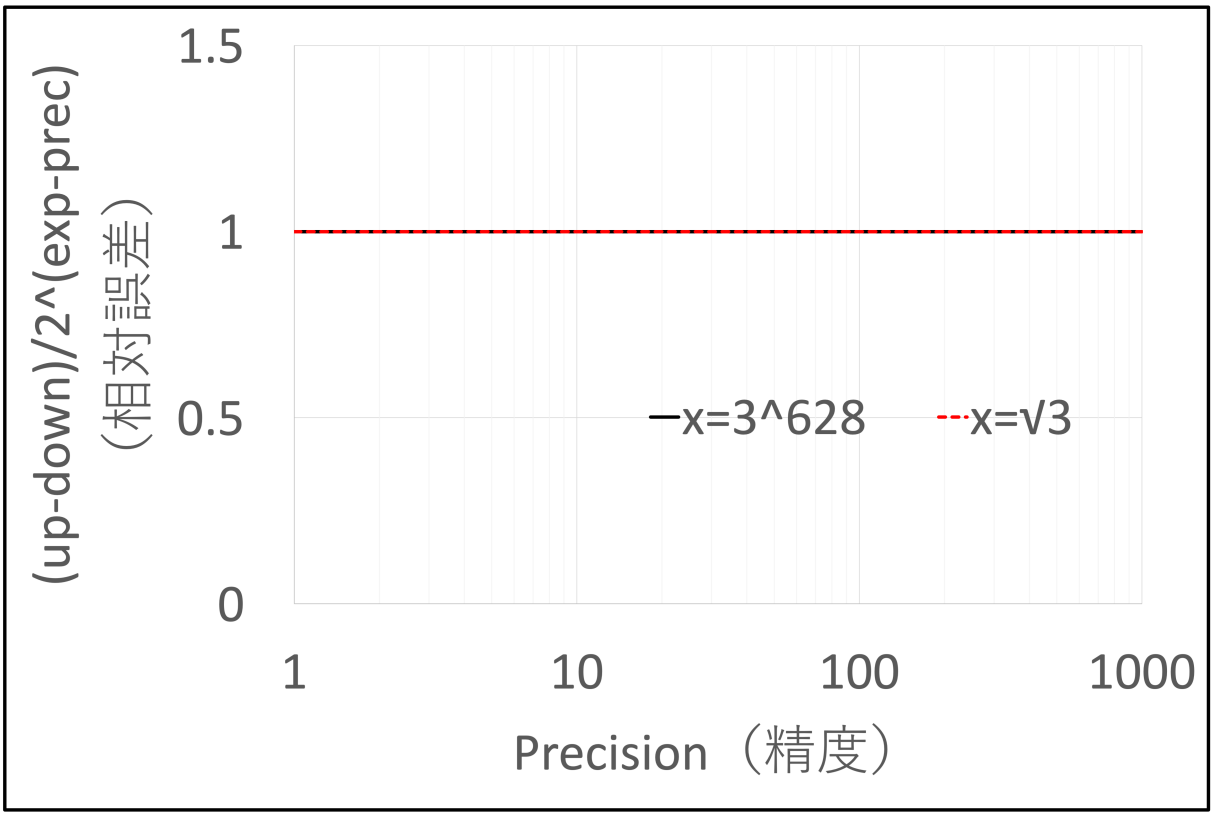

図 20: 精度に対する相対誤差の片対数グラフ（実数の余接）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	1.0	1.0	1.0	1.0	1.0

表 25: 精度が 1 から 1000 のときの相対誤差の最大値（実数の余接）

C.2.20　余弦の逆関数（$\arccos x$）

第 4.2 節の多倍長精度の余弦の逆関数（実数）の実験結果を表 26 に示す.

x	S_{-150}^{300}	$-S_{-150}^{300}$
相対誤差	1.0	1.0

表 26: 精度が 1 から 1000 のときの相対誤差の最大値（実数の余弦の逆関数）

C.2.21　正弦の逆関数（$\arcsin x$）

第 4.2 節の多倍長精度の正弦の逆関数（実数）の実験結果を表 27 に示す.

x	S_{-150}^{300}	$-S_{-150}^{300}$
相対誤差	1.0	1.0

表 27: 精度が 1 から 1000 のときの相対誤差の最大値（実数の正弦の逆関数）

C.2.22　正接の逆関数（$\arctan x$）

第 4.2 節の多倍長精度の正接の逆関数（実数）の実験結果を図 21 と表 28 に示す.

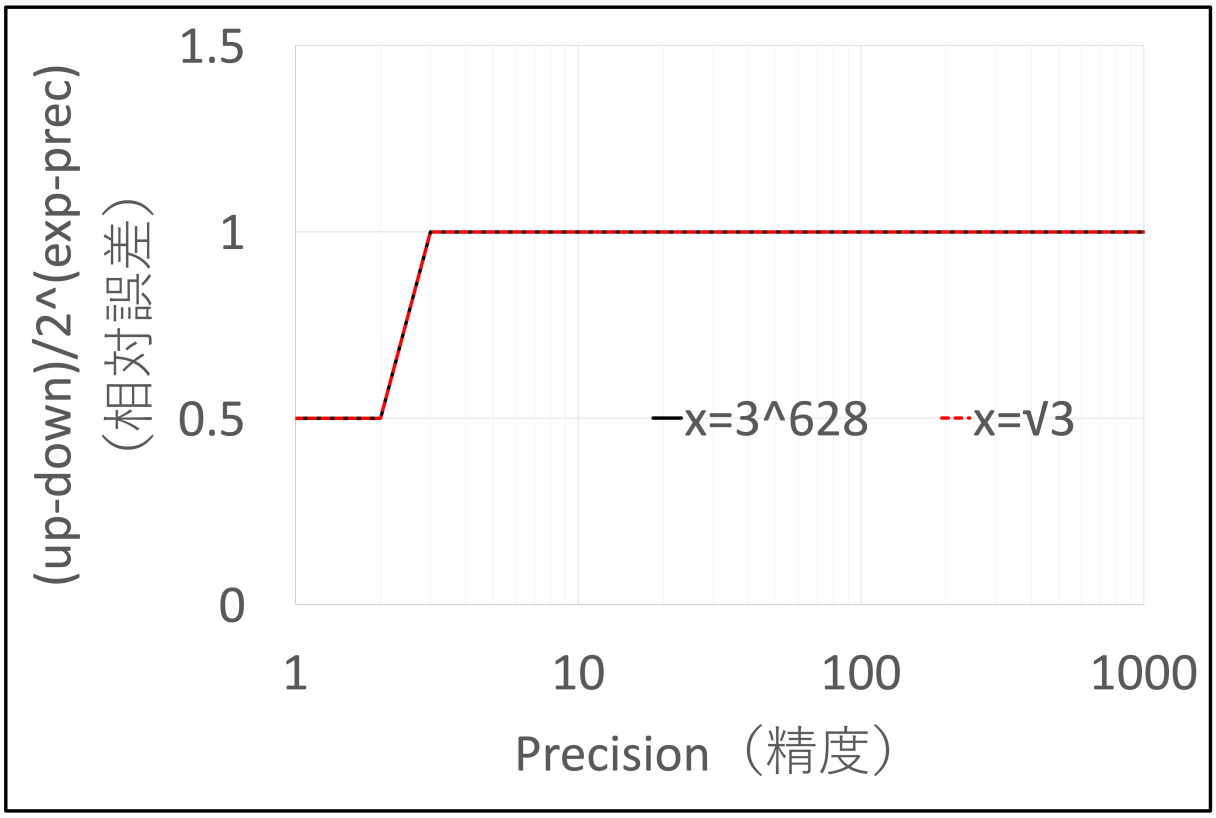

図 21: 精度に対する相対誤差の片対数グラフ（実数の正接の逆関数）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	1.0	1.0	1.0	1.0	1.0

表 28: 精度が 1 から 1000 のときの相対誤差の最大値（実数の正接の逆関数）

C.2.23　双曲線余弦（$\cosh x$）

第 4.2 節の多倍長精度の双曲線余弦（実数）の実験結果を図 22 と表 29 に示す．図 22 のグラフにおいて，$x = 3^{628}$ のときは segmentation fault により記録不可であった．また，表 29 において，$\times$ と書いてある部分は segmentation fault により記録不可であった．

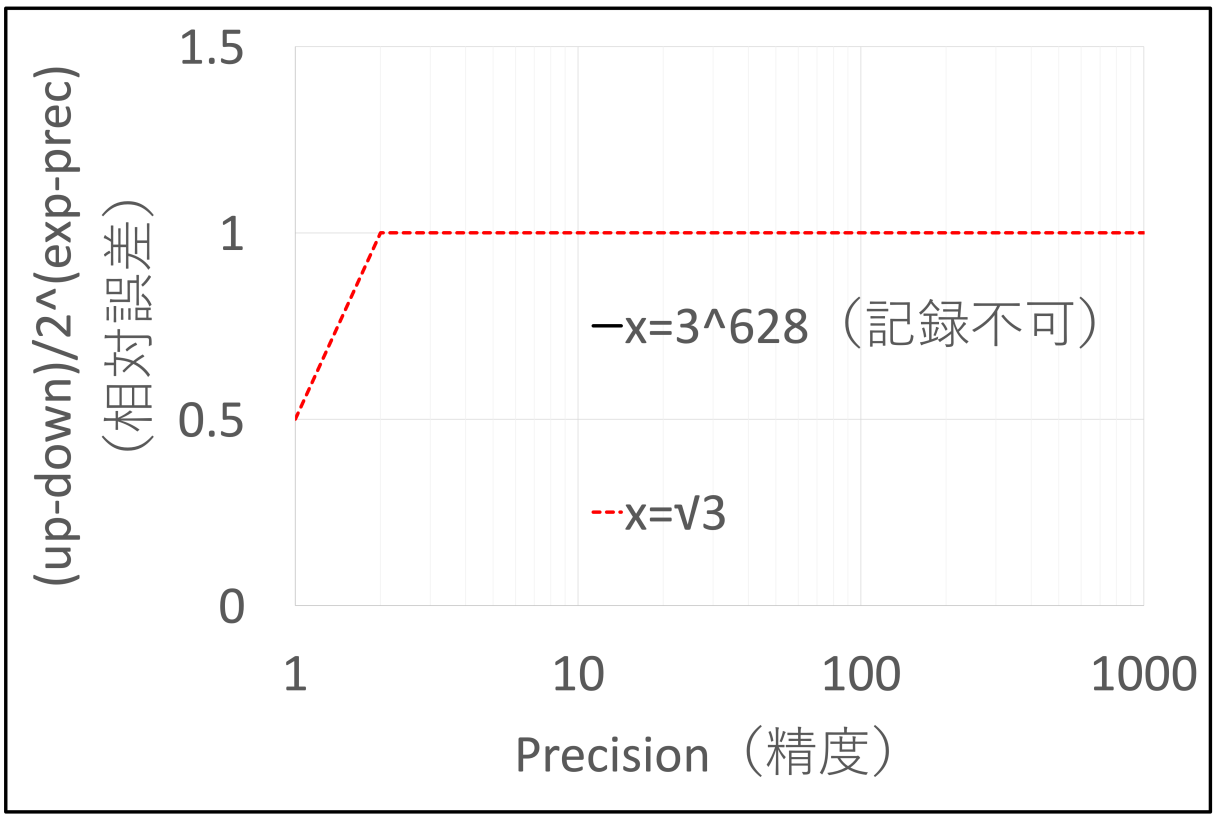

図 22: 精度に対する相対誤差の片対数グラフ（実数の双曲線余弦）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	$\times$	1.0	$\times$	1.0	1.0

表 29: 精度が 1 から 1000 のときの相対誤差の最大値（実数の双曲線余弦）

C.2.24　双曲線正弦（$\sinh x$）

第 4.2 節の多倍長精度の双曲線正弦（実数）の実験結果を図 23 と表 30 に示す．図 23 のグラフにおいて，$x = 3^{628}$ のときは segmentation fault により記録不可であった．また，表 30 において，$\times$ と書いてある部分は segmentation fault により記録不可であった．

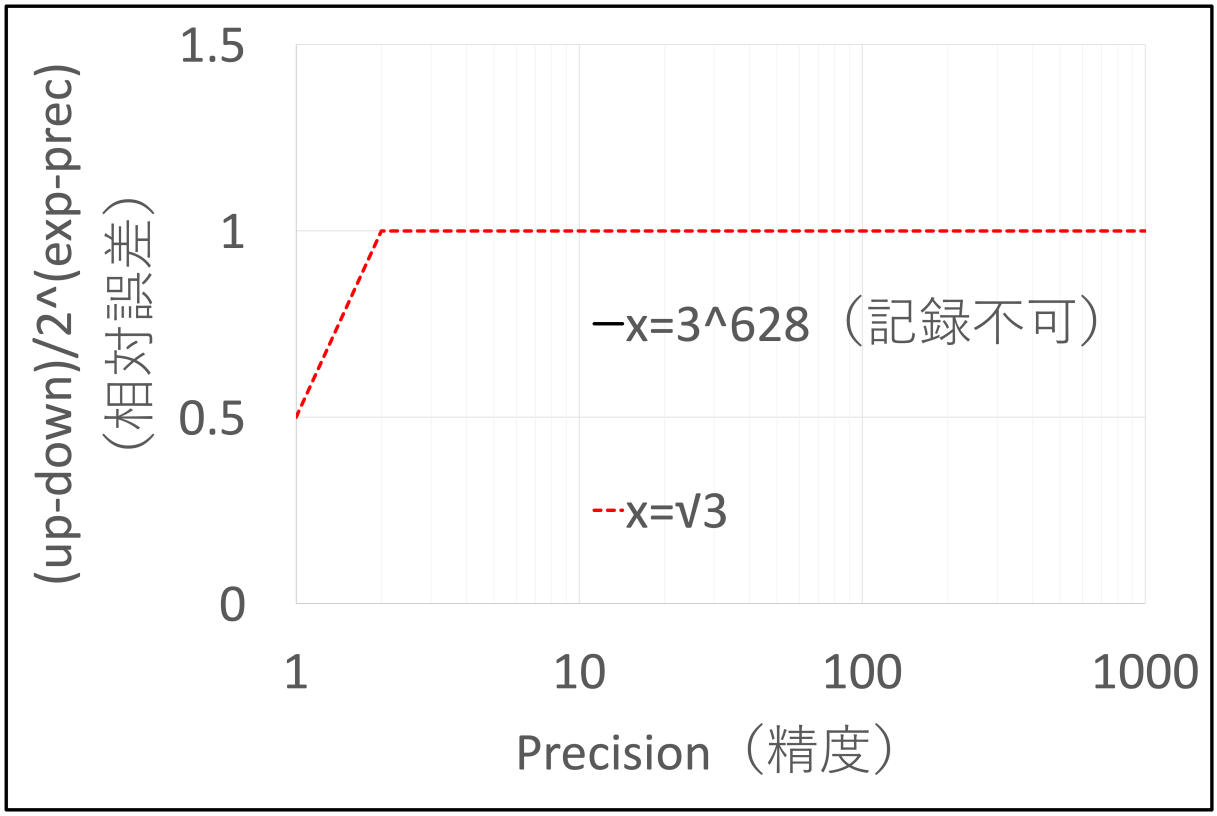

図 23: 精度に対する相対誤差の片対数グラフ（実数の双曲線正弦）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	$\times$	1.0	$\times$	1.0	1.0

表 30: 精度が 1 から 1000 のときの相対誤差の最大値（実数の双曲線正弦）

C.2.25　双曲線正接（$\tanh x$）

第 4.2 節の多倍長精度の双曲線正接（実数）の実験結果を図 24 と表 31 に示す.

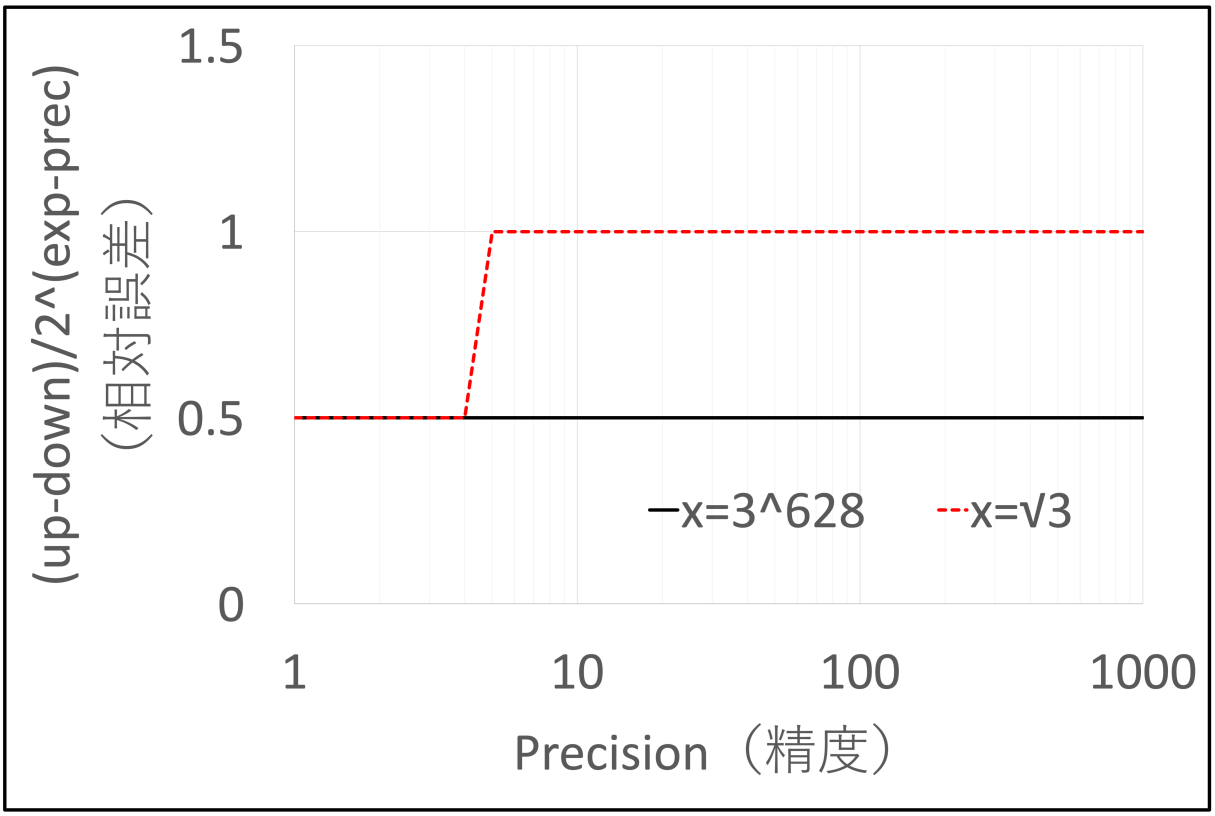

図 24: 精度に対する相対誤差の片対数グラフ（実数の双曲線正接）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	1.0	1.0	1.0	1.0	1.0

表 31: 精度が 1 から 1000 のときの相対誤差の最大値（実数の双曲線正接）

C.2.26　双曲線正割（$\mathrm{sech}\, x$）

第 4.2 節の多倍長精度の双曲線正割（実数）の実験結果を図 25，26 及び表 32 に示す.

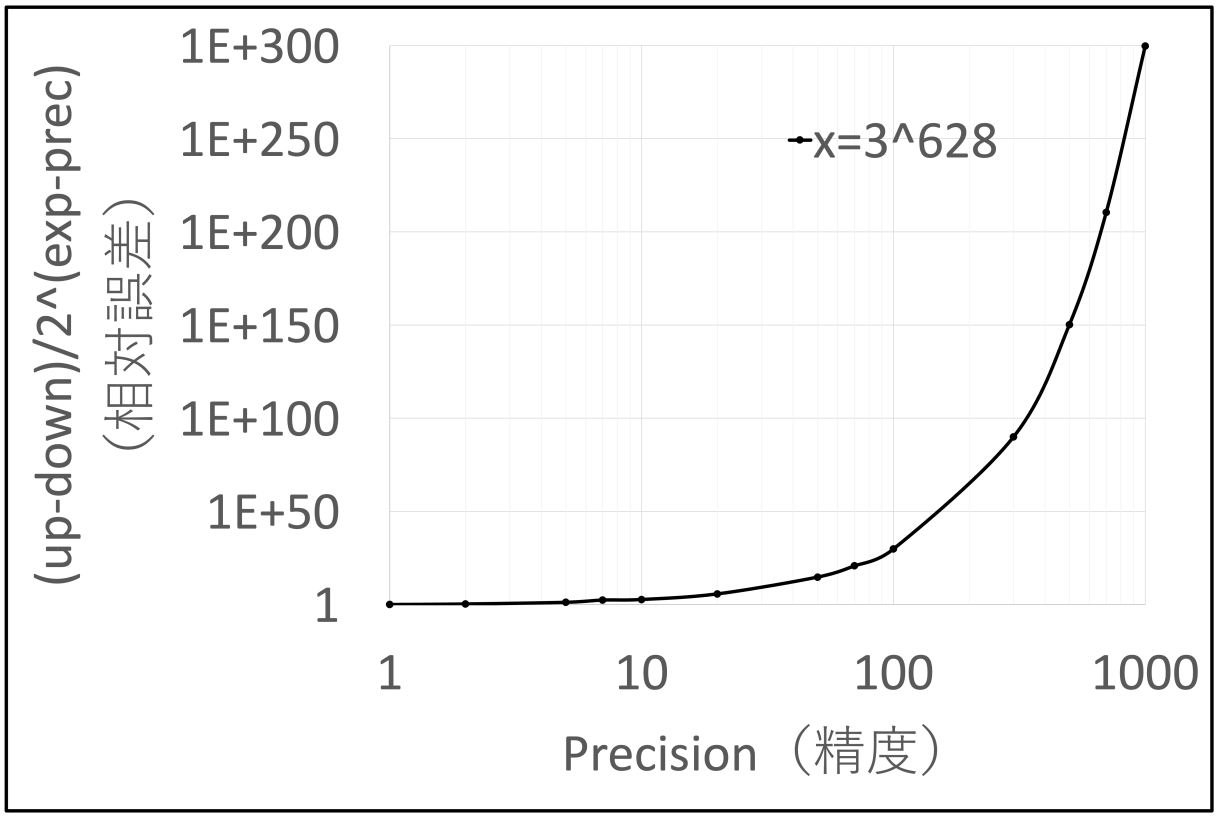

図 25: 精度に対する相対誤差の両対数グラフ（実数の双曲線正割）

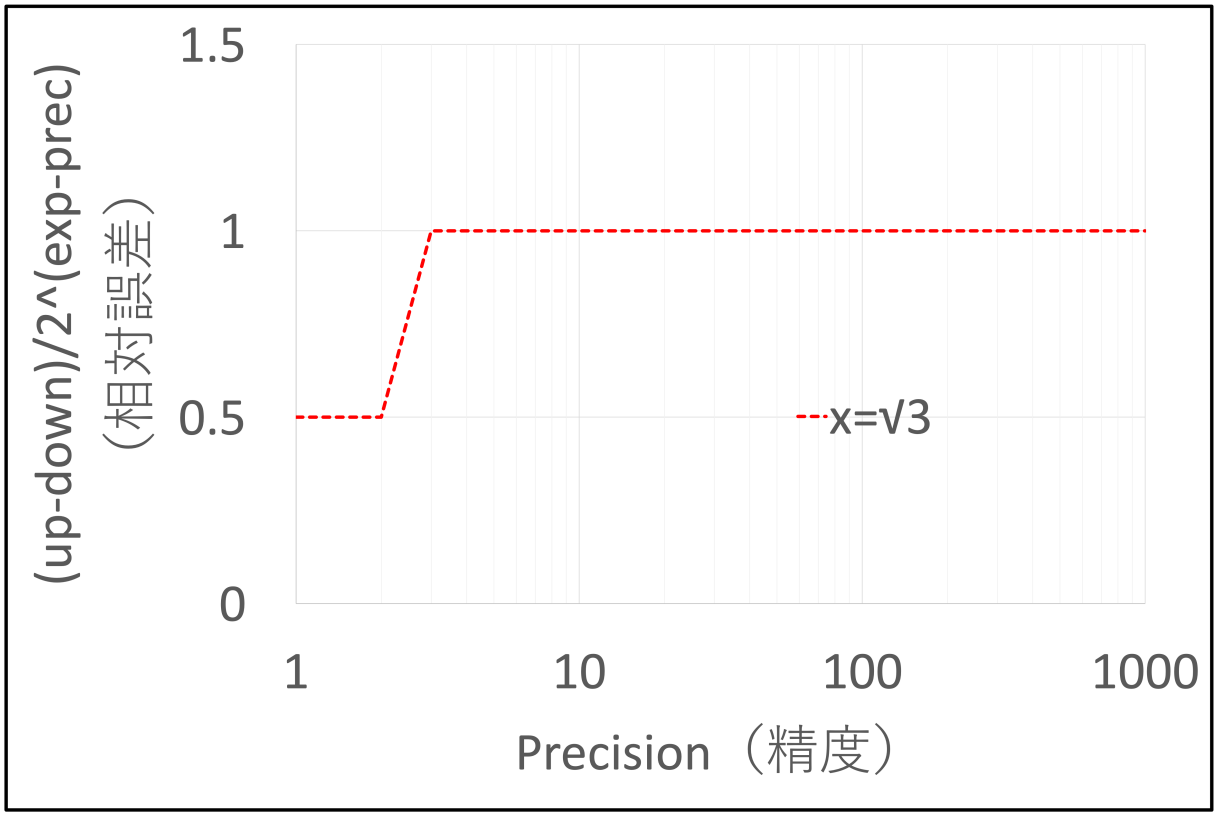

図 26: 精度に対する相対誤差の片対数グラフ（実数の双曲線正割）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	5.4×10^{300}	1.0	5.4×10^{300}	1.0	1.0

表 32: 精度が 1 から 1000 のときの相対誤差の最大値（実数の双曲線正割）

C.2.27　双曲線余割（csch x）

第 4.2 節の多倍長精度の双曲線余割（実数）の実験結果を図 27，28 及び表 33 に示す.

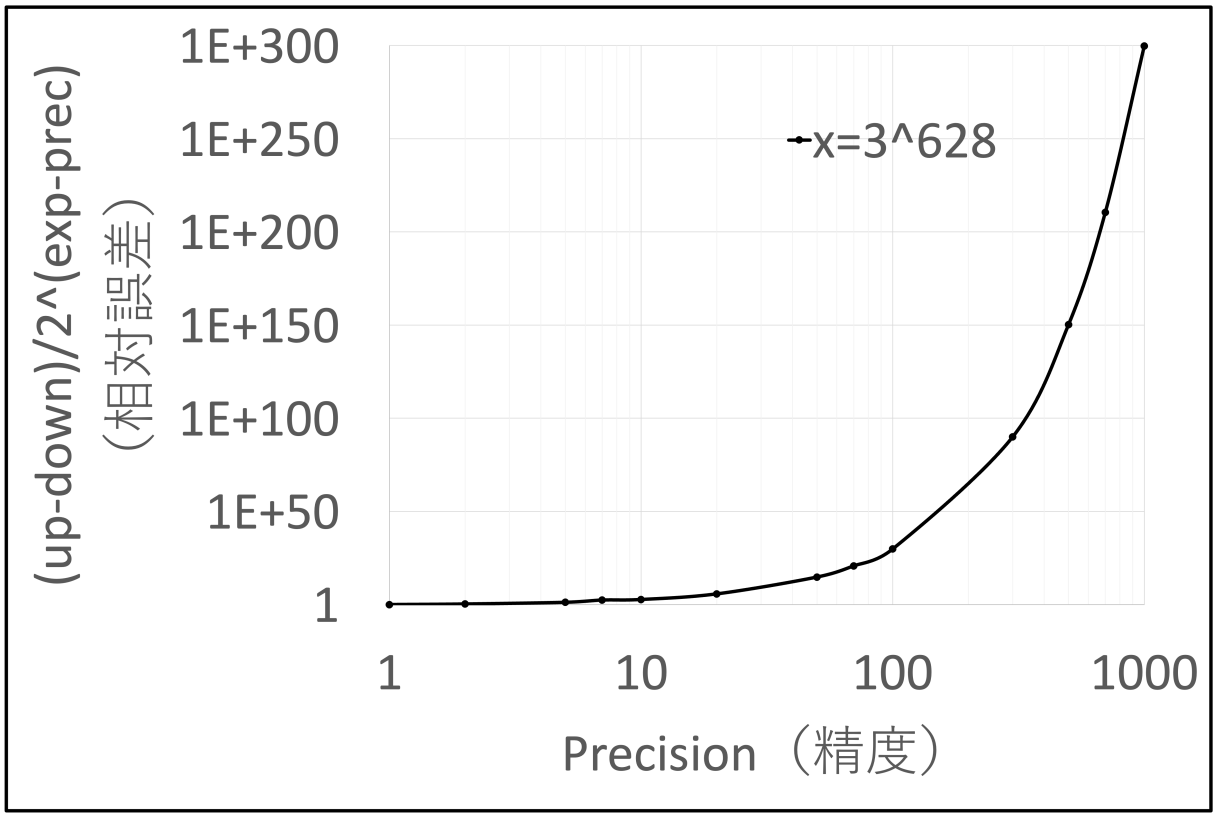

図 27: 精度に対する相対誤差の両対数グラフ（実数の双曲線余割）

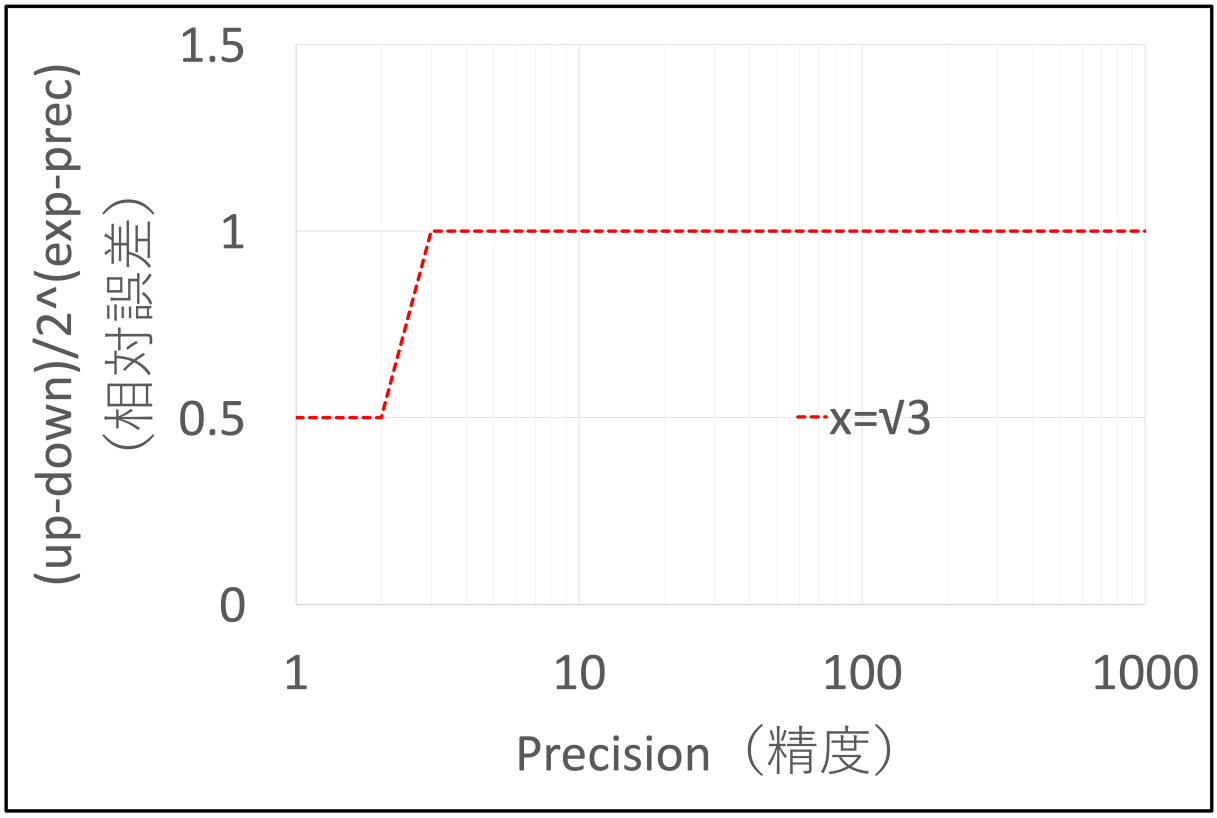

図 28: 精度に対する相対誤差の片対数グラフ（実数の双曲線余割）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	5.4×10^{300}	1.0	0.0	1.0	1.0

表 33: 精度が 1 から 1000 のときの相対誤差の最大値（実数の双曲線余割）

C.2.28　双曲線余接（$\coth x$）

第 4.2 節の多倍長精度の双曲線余接（実数）の実験結果を図 29 と表 34 に示す.

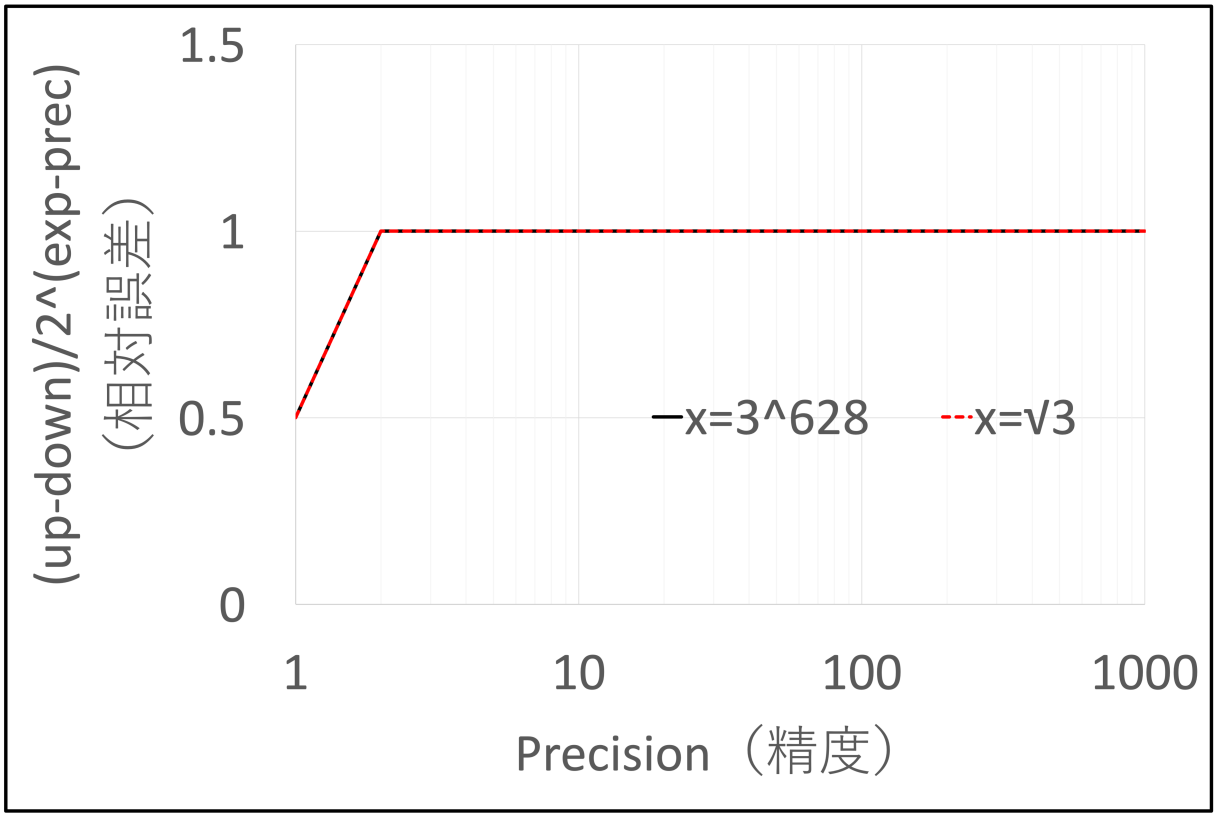

図 29: 精度に対する相対誤差の片対数グラフ（実数の双曲線余接）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	1.0	1.0	1.0	1.0	1.0

表 34: 精度が 1 から 1000 のときの相対誤差の最大値（実数の双曲線余接）

C.2.29　双曲線余弦の逆関数（$\operatorname{arccosh} x$）

第 4.2 節の多倍長精度の双曲線余弦の逆関数（実数）の実験結果を図 30 と表 35 に示す.

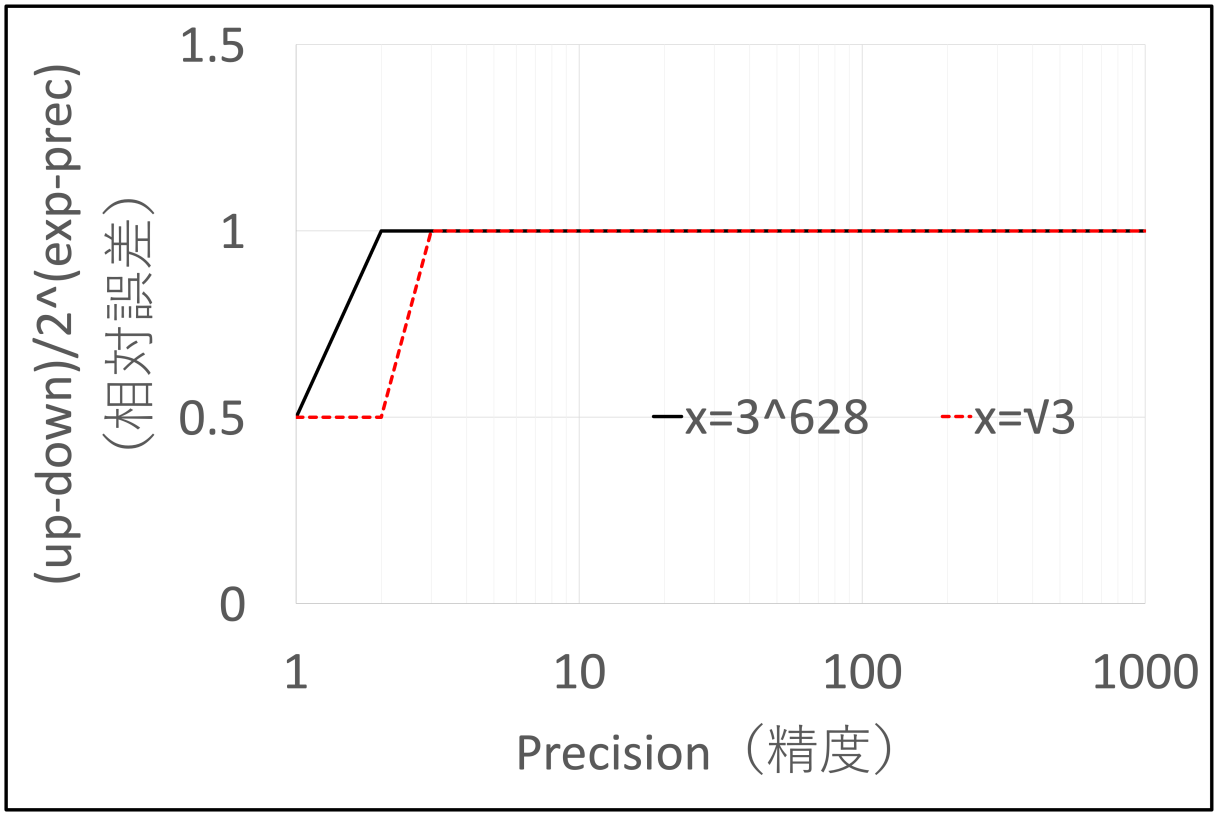

図 30: 精度に対する相対誤差の片対数グラフ（実数の双曲線余弦の逆関数）

x	L^{300}_{+150}	S^{300}_{-150}	$\sqrt{2}$
相対誤差	1.0	×	1.0

表 35: 精度が 1 から 1000 のときの相対誤差の最大値（実数の双曲線余弦の逆関数）

C.2.30　双曲線正弦の逆関数（$\operatorname{arcsinh} x$）

第 4.2 節の多倍長精度の双曲線余弦の逆関数（実数）の実験結果を図 31 と表 36 に示す.

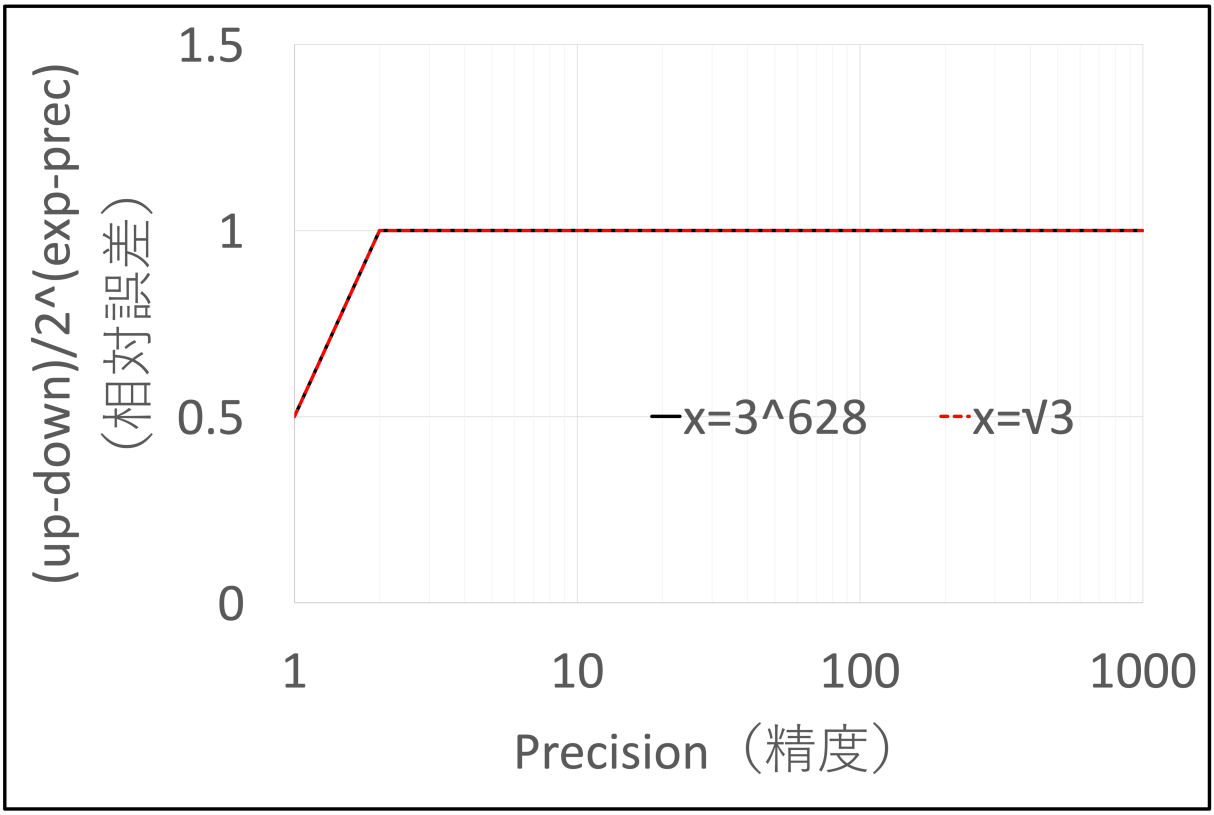

図 31: 精度に対する相対誤差の片対数グラフ（実数の双曲線余弦の逆関数）

x	L^{300}_{+150}	S^{300}_{-150}	$-L^{300}_{+150}$	$-S^{300}_{-150}$	$\sqrt{2}$
相対誤差	1.0	1.0	1.0	1.0	1.0

表 36: 精度が 1 から 1000 のときの相対誤差の最大値（実数の双曲線正弦の逆関数）

C.2.31　双曲線正接の逆関数（$\operatorname{arctanh} x$）

第 4.2 節の多倍長精度の双曲線余弦の逆関数（実数）の実験結果を表 37 に示す.

x	S_{-150}^{300}	$-S_{-150}^{300}$
相対誤差	1.0	1.0

表 37: 精度が 1 から 1000 のときの相対誤差の最大値（実数の双曲線正接の逆関数）

C.3　点区間同士による多倍長精度の四則演算のソースコード（複素数）

```
#include<isys.h> // 独自のヘッダーファイルをインクルード
#include "mi.c" // 独自のソースファイルをインクルード

int main() {
    // 使用する変数の宣言
    int diff_exp_prec_r, diff_exp_prec_i; // 整数型の変数の宣言（実部と虚部の精度）
    cmulti *diff_up_down, *diff; // cmulti 型のポインタ変数の宣言
    mi_c *a, *b, *c; // mi_c 型のポインタ変数の宣言

    // 精度を 1 から 1000 まで動かして，ループを実行
    for(int i = 1; i < 1001; i++) {

        set_default_prec(i); // デフォルトの精度を設定

        // メモリの確保
        a = mi_callocate(); // mi_c 型の変数のメモリを確保
        b = mi_callocate();
        c = mi_callocate();
        diff_up_down = callocate(); // cmulti 型の変数のメモリを確保
        diff = callocate();

        // 初期値の設定
        mic_strstr_set_dot(a, "1", "2"); // 文字列の点区間 1+2i を複素数の区間 a に格納
        mic_strstr_set_dot(b, "3", "4"); // 文字列の点区間 3+4i を複素数の区間 b に格納

        // 点区間同士の機械区間演算
        mic_micmic_add(c, a, b); // 複素数の点区間 a と b の和を区間 c に格納

        // 相対誤差を計算（実部と虚部）
        mpfr_sub(diff_up_down->r, c->r->up, c->r->down, MPFR_RNDN);
        mpfr_sub(diff_up_down->i, c->i->up, c->i->down, MPFR_RNDN);
```

```
        diff_exp_prec_r =
         mpfr_get_exp(c->r->down) - mpfr_get_prec(c->r->down);
        diff_exp_prec_i =
         mpfr_get_exp(c->i->down) - mpfr_get_prec(c->i->down);

        rdiv_2exp(diff->r, diff_up_down->r, diff_exp_prec_r);
        rdiv_2exp(diff->i, diff_up_down->i, diff_exp_prec_i);

        // 結果を標準出力
        printf("prec      = %d\n\n", i); // 精度を出力
        mpfr_printf("z_r: (up-down) / 2^(exp - prec) =
         %380.350Rf\t\n\n", diff->r); // 実部の相対誤差を出力
        mpfr_printf("z_i: (up-down) / 2^(exp - prec) =
         %380.350Rf\t\n\n", diff->i); // 虚部の相対誤差を出力

        // メモリの解放
        a = mi_cfree(a); // mi_c 型の変数のメモリを解放
        b = mi_cfree(b);
        c = mi_cfree(c);
        diff_up_down = cmfree(diff_up_down); // cmulti 型の変数のメモリを解
放
        diff = cmfree(diff);
    }

    return 0; // プログラムの終了
}
```

C.4　精度に対する相対誤差の表とグラフ（複素数）

C.4.1　加算（$(x_r + ix_i) + (y_r + iy_i)$）

第 4.3 節の多倍長精度の加算（複素数）の実験結果を図 32 と表 38 に示す.

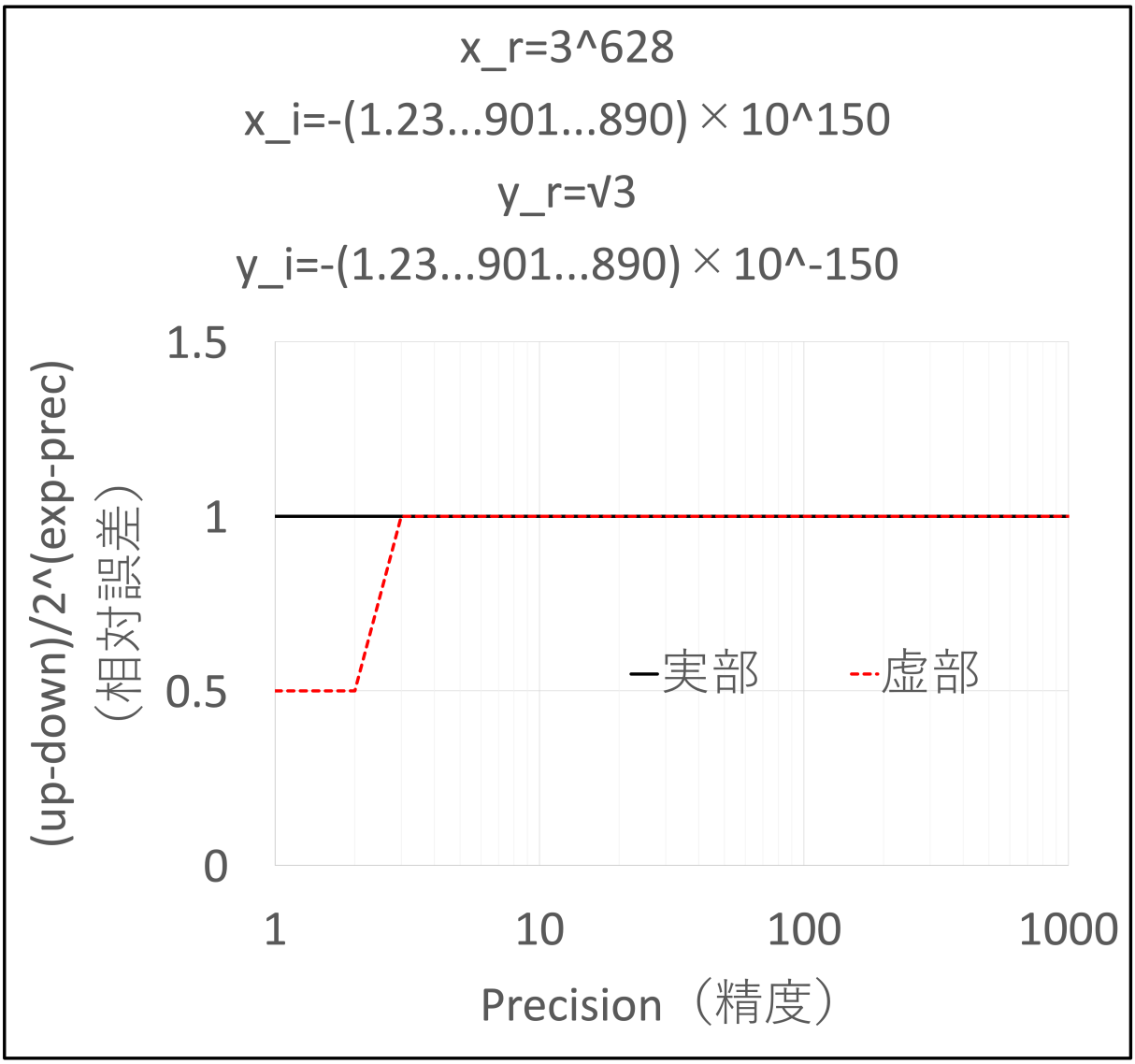

図 32: 精度に対する相対誤差の片対数グラフ（複素数の加算）

x_r,x_i \ y_r,y_i	L^{300}_{+152},L^{300}_{+153}	L^{300}_{+152},S^{302}_{-150}	S^{302}_{-150},L^{300}_{+152}	S^{302}_{-150},S^{303}_{-150}
L^{300}_{+150},L^{300}_{+151}	1.0	1.0	1.0	1.0
L^{300}_{+150},S^{300}_{-150}	1.0	1.0	1.0	1.0
S^{300}_{-150},L^{300}_{+150}	1.0	1.0	1.0	1.0
S^{300}_{-150},S^{301}_{-150}	1.0	1.0	1.0	1.0

表 38: 精度が 1 から 1000 のときの相対誤差の最大値 (複素数の加算)

C.4.2 減算（$(x_r + ix_i) - (y_r + iy_i)$）

第 4.3 節の多倍長精度の減算（複素数）の実験結果を図 33 と表 39 に示す.

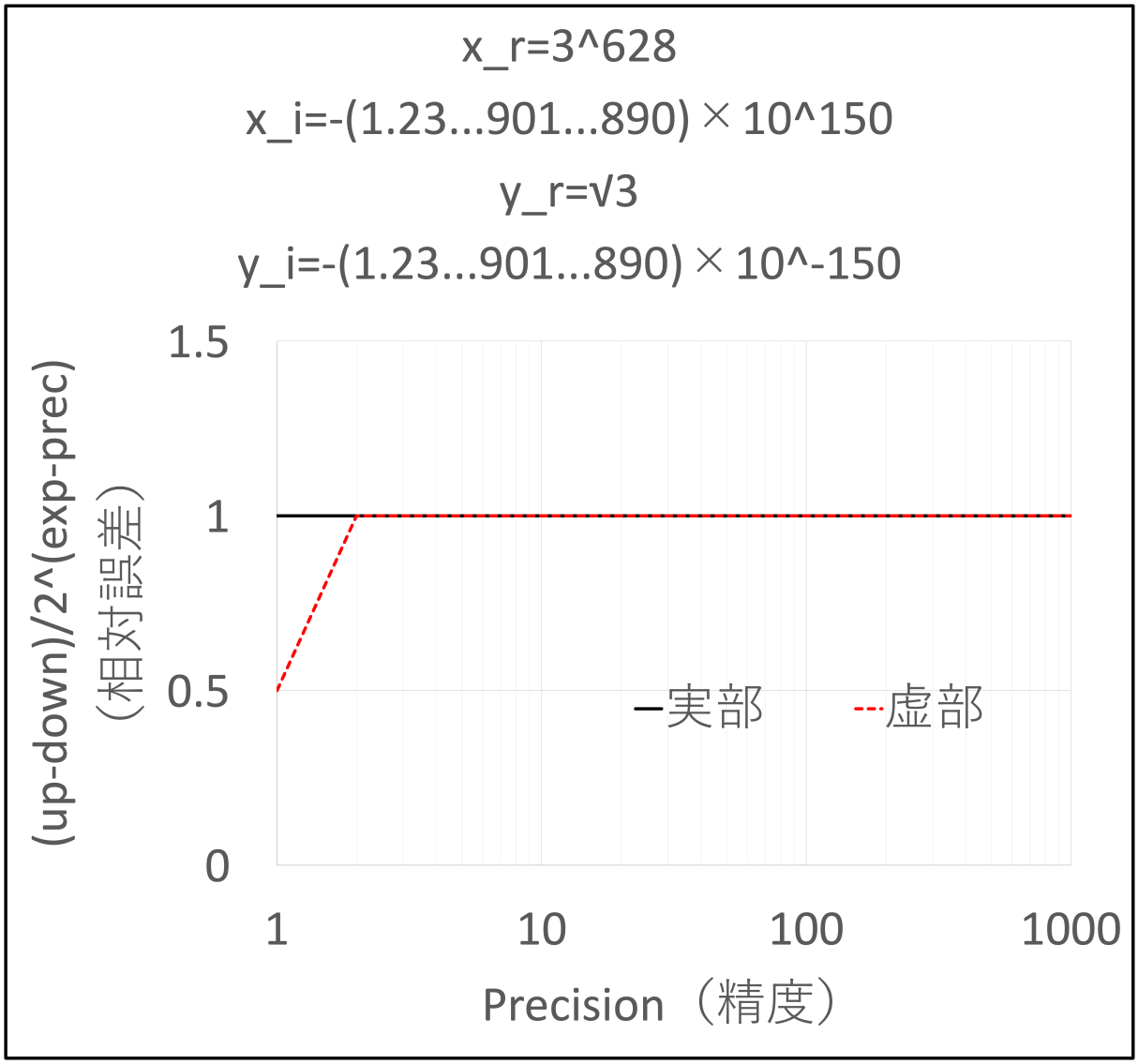

図 33: 精度に対する相対誤差の片対数グラフ（複素数の減算）

x_r,x_i \ y_r,y_i	L^{300}_{+152},L^{300}_{+153}	L^{300}_{+152},S^{302}_{-150}	S^{302}_{-150},L^{300}_{+152}	S^{302}_{-150},S^{303}_{-150}
L^{300}_{+150},L^{300}_{+151}	1.0	1.0	1.0	1.0
L^{300}_{+150},S^{300}_{-150}	1.0	1.0	1.0	1.0
S^{300}_{-150},L^{300}_{+150}	1.0	1.0	1.0	1.0
S^{300}_{-150},S^{301}_{-150}	1.0	1.0	1.0	0.0

表 39: 精度が 1 から 1000 のときの相対誤差の最大値 (複素数の減算)

C.4.3 乗算（$(x_r + ix_i) \times (y_r + iy_i)$）

第 4.3 節の多倍長精度の乗算（複素数）の実験結果を図 34 と表 40 に示す.

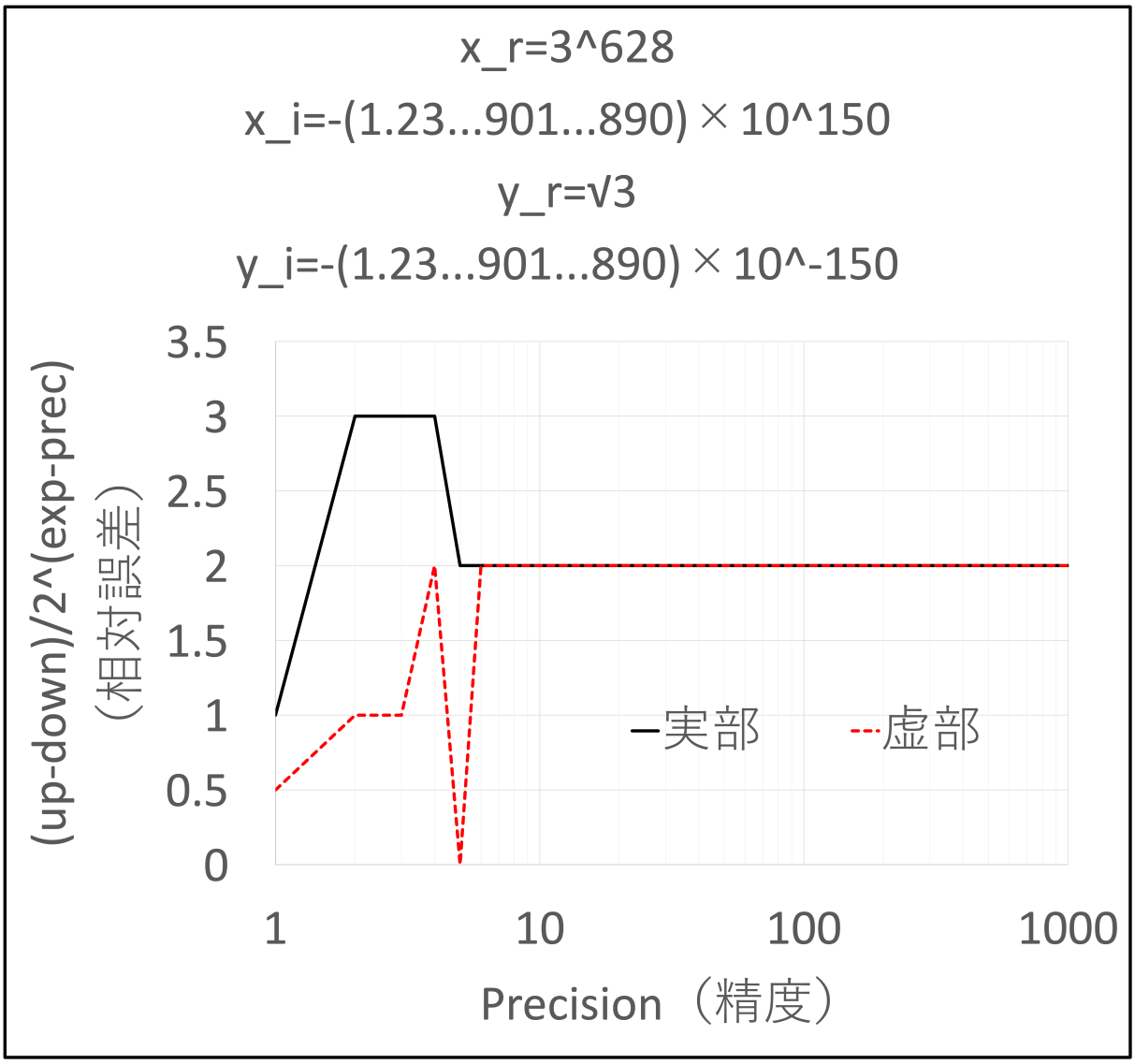

図 34: 精度に対する相対誤差の片対数グラフ（複素数の乗算）

x_r,x_i \ y_r,y_i	L^{300}_{+152},L^{300}_{+153}	L^{300}_{+152},S^{302}_{-150}	S^{302}_{-150},L^{300}_{+152}	S^{302}_{-150},S^{303}_{-150}
L^{300}_{+150},L^{300}_{+151}	2.0	2.0	3.0	2.0
L^{300}_{+150},S^{300}_{-150}	2.0	2.0	3.0	2.0
S^{300}_{-150},L^{300}_{+150}	3.0	3.0	2.0	2.0
S^{300}_{-150},S^{301}_{-150}	2.0	2.0	2.0	2.7×10^{300}

表 40: 精度が 1 から 1000 のときの相対誤差の最大値 (複素数の乗算)

C.4.4 除算（$(x_r + ix_i) \,/\, (y_r + iy_i)$）

第 4.3 節の多倍長精度の除算（複素数）の実験結果を図 35 と表 41 に示す.

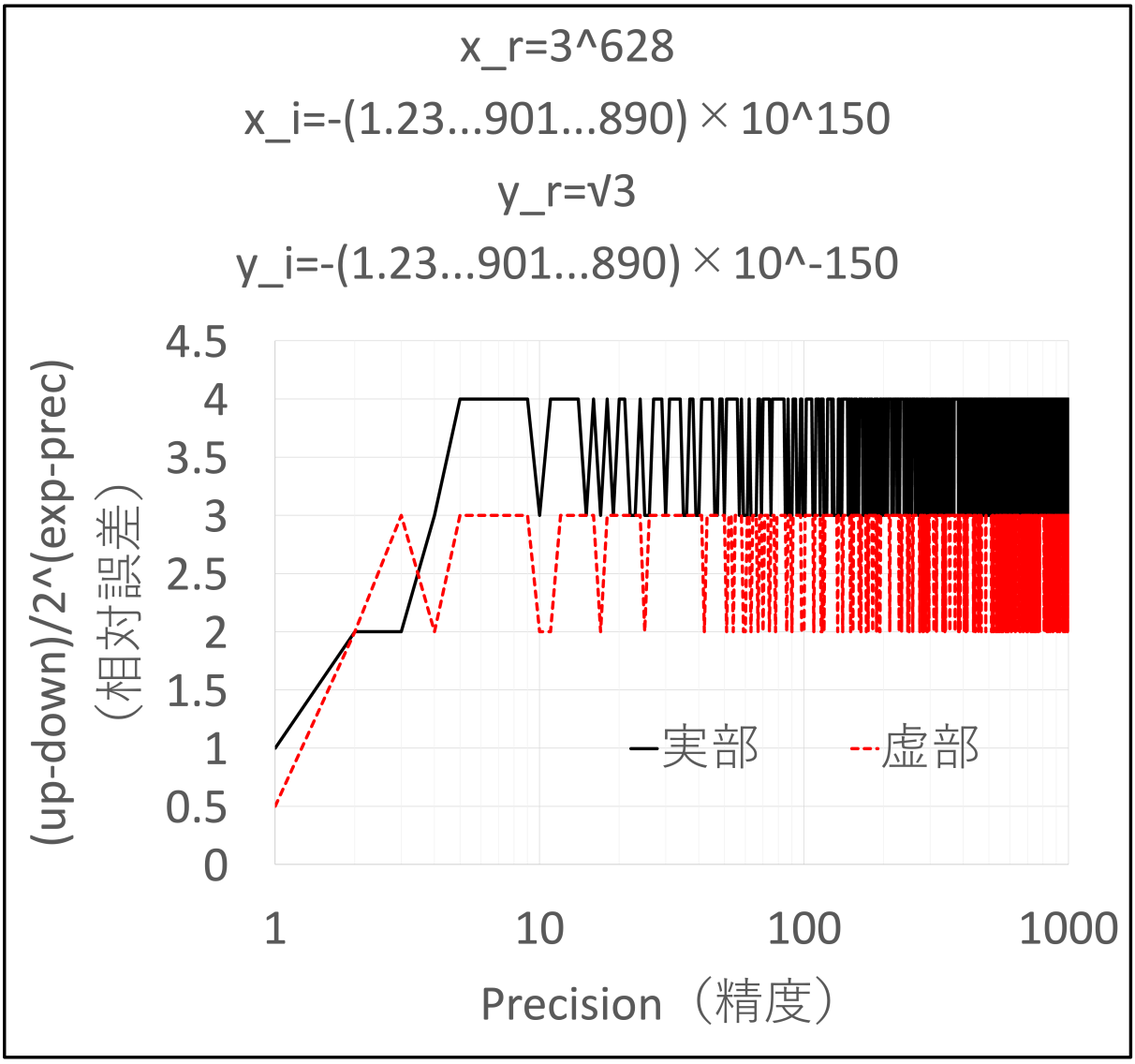

図 35: 精度に対する相対誤差の片対数グラフ（複素数の除算）

x_r,x_i \ y_r,y_i	$L^{300}_{+77},L^{300}_{+78}$	$L^{300}_{+77},S^{302}_{-75}$	$S^{302}_{-75},L^{300}_{+77}$	$S^{302}_{-75},S^{303}_{-75}$
$L^{300}_{+75},L^{300}_{+76}$	1.2×10^{301}	3.0	4.0	5.0
$L^{300}_{+75},S^{300}_{-75}$	4.0	4.0	4.0	3.0
$S^{300}_{-75},L^{300}_{+75}$	4.0	4.0	3.0	5.2×10^{124}
$S^{300}_{-75},S^{301}_{-75}$	4.0	4.0	4.0	4.3×10^{300}

表 41: 精度が 1 から 1000 のときの相対誤差の最大値 (複素数の除算)

C.4.5　改良後の乗算（$(x_r + ix_i) \times (y_r + iy_i)$）

第 4.4 節の多倍長精度の改良後の乗算（複素数）の実験結果を図 36 と表 42〜52 に示す.

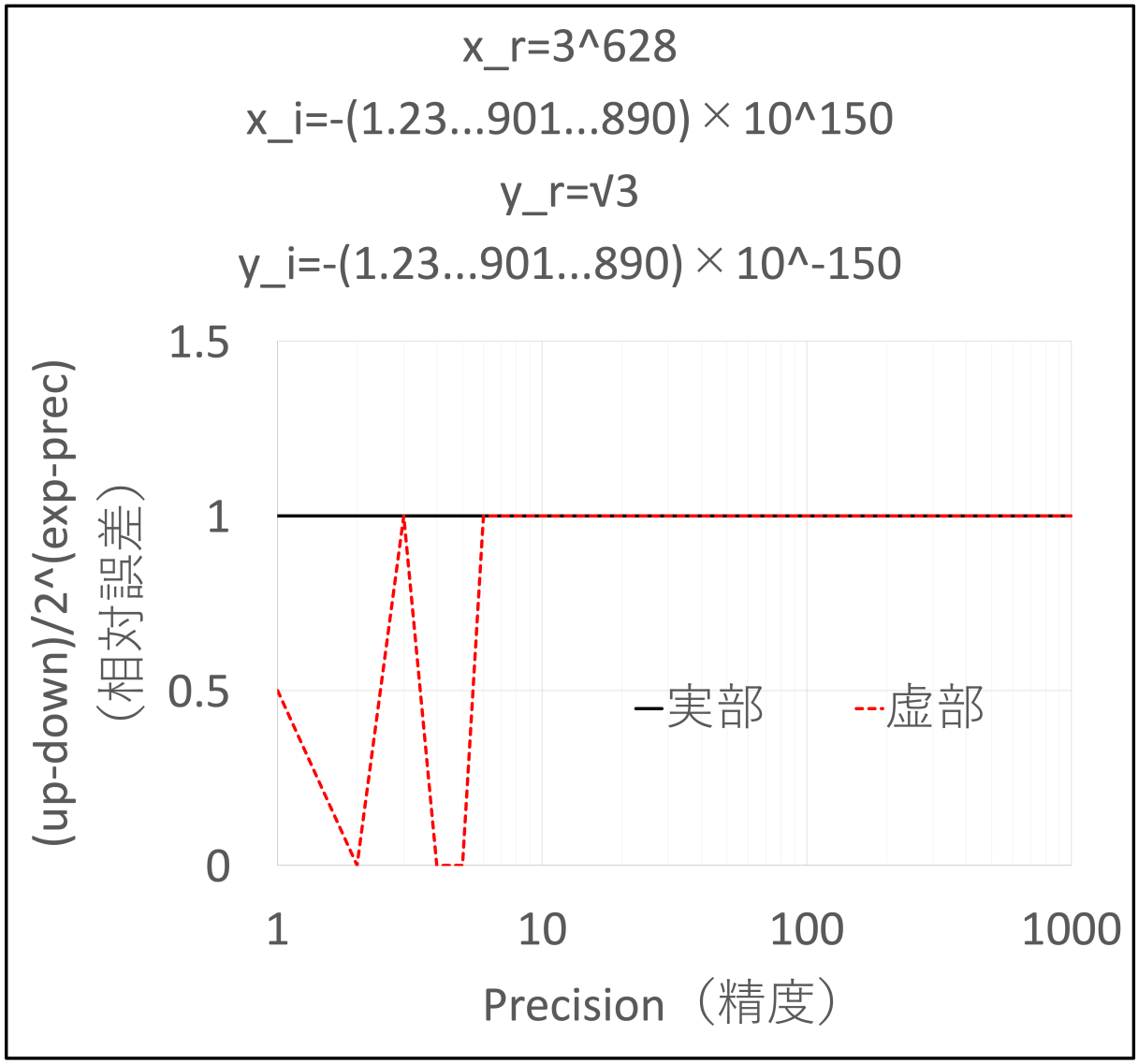

図 36: 精度に対する相対誤差の片対数グラフ（mic_micmic_mul_1ulp を使用したときの複素数の乗算）

x_r, x_i \ y_r, y_i	$S^{302}_{-150}, L^{300}_{+152}$	$S^{302}_{-150}, S^{303}_{-150}$
$L^{300}_{+150}, L^{300}_{+151}$	2.0	2.0
$S^{300}_{-150}, S^{301}_{-150}$	2.0	1.3×10^{300}

表 42: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 1 増やしたときの複素数の乗算)

x_r, x_i \ y_r, y_i	$S^{302}_{-150}, L^{300}_{+152}$	$S^{302}_{-150}, S^{303}_{-150}$
$L^{300}_{+150}, L^{300}_{+151}$	2.0	2.0
$S^{300}_{-150}, S^{301}_{-150}$	2.0	1.3×10^{300}

表 43: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 2 増やしたときの複素数の乗算)

x_r, x_i \ y_r, y_i	$S^{302}_{-150}, L^{300}_{+152}$	$S^{302}_{-150}, S^{303}_{-150}$
$L^{300}_{+150}, L^{300}_{+151}$	2.0	2.0
$S^{300}_{-150}, S^{301}_{-150}$	2.0	1.3×10^{300}

表 44: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 4 増やしたときの複素数の乗算)

x_r,x_i \ y_r,y_i	S^{302}_{-150},L^{300}_{+152}	S^{302}_{-150},S^{303}_{-150}
L^{300}_{+150},L^{300}_{+151}	2.0	2.0
S^{300}_{-150},S^{301}_{-150}	2.0	1.3×10^{300}

表 45: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 8 増やしたときの複素数の乗算)

x_r,x_i \ y_r,y_i	S^{302}_{-150},L^{300}_{+152}	S^{302}_{-150},S^{303}_{-150}
L^{300}_{+150},L^{300}_{+151}	1.0	1.0
S^{300}_{-150},S^{301}_{-150}	1.0	1.3×10^{300}

表 46: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 16 増やしたときの複素数の乗算)

x_r,x_i \ y_r,y_i	S^{302}_{-150},L^{300}_{+152}	S^{302}_{-150},S^{303}_{-150}
L^{300}_{+150},L^{300}_{+151}	1.0	1.0
S^{300}_{-150},S^{301}_{-150}	1.0	1.3×10^{300}

表 47: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 32 増やしたときの複素数の乗算)

x_r,x_i \ y_r,y_i	S^{302}_{-150},L^{300}_{+152}	S^{302}_{-150},S^{303}_{-150}
L^{300}_{+150},L^{300}_{+151}	1.0	1.0
S^{300}_{-150},S^{301}_{-150}	1.0	1.3×10^{300}

表 48: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 64 増やしたときの複素数の乗算)

x_r,x_i \ y_r,y_i	S^{302}_{-150},L^{300}_{+152}	S^{302}_{-150},S^{303}_{-150}
L^{300}_{+150},L^{300}_{+151}	1.0	1.0
S^{300}_{-150},S^{301}_{-150}	1.0	1.3×10^{300}

表 49: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 128 増やしたときの複素数の乗算)

x_r,x_i \ y_r,y_i	$S^{302}_{-150},L^{300}_{+152}$	$S^{302}_{-150},S^{303}_{-150}$
$L^{300}_{+150},L^{300}_{+151}$	1.0	1.0
$S^{300}_{-150},S^{301}_{-150}$	1.0	1.0

表 50: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 2 倍にしたときの複素数の乗算)

x_r,x_i \ y_r,y_i	$S^{302}_{-150},L^{300}_{+152}$	$S^{302}_{-150},S^{303}_{-150}$
$L^{300}_{+150},L^{300}_{+151}$	1.0	1.0
$S^{300}_{-150},S^{301}_{-150}$	1.0	1.0

表 51: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 3 倍にしたときの複素数の乗算)

x_r,x_i \ y_r,y_i	$L^{300}_{+152},L^{300}_{+153}$	$L^{300}_{+152},S^{302}_{-150}$	$S^{302}_{-150},L^{300}_{+152}$	$S^{302}_{-150},S^{303}_{-150}$
$L^{300}_{+150},L^{300}_{+151}$	1.0	1.0	1.0	1.0
$L^{300}_{+150},S^{300}_{-150}$	1.0	1.0	1.0	1.0
$S^{300}_{-150},L^{300}_{+150}$	1.0	1.0	1.0	1.0
$S^{300}_{-150},S^{301}_{-150}$	1.0	1.0	1.0	1.0

表 52: 精度が 1 から 1000 のときの相対誤差の最大値 (mic_micmic_mul_1ulp を使用したときの複素数の乗算)

C.4.6　改良後の除算（$(x_r + ix_i) \,/\, (y_r + iy_i)$）

第 4.4 節の多倍長精度の改良後の除算（複素数）の実験結果を図 37 と表 53〜63 に示す.

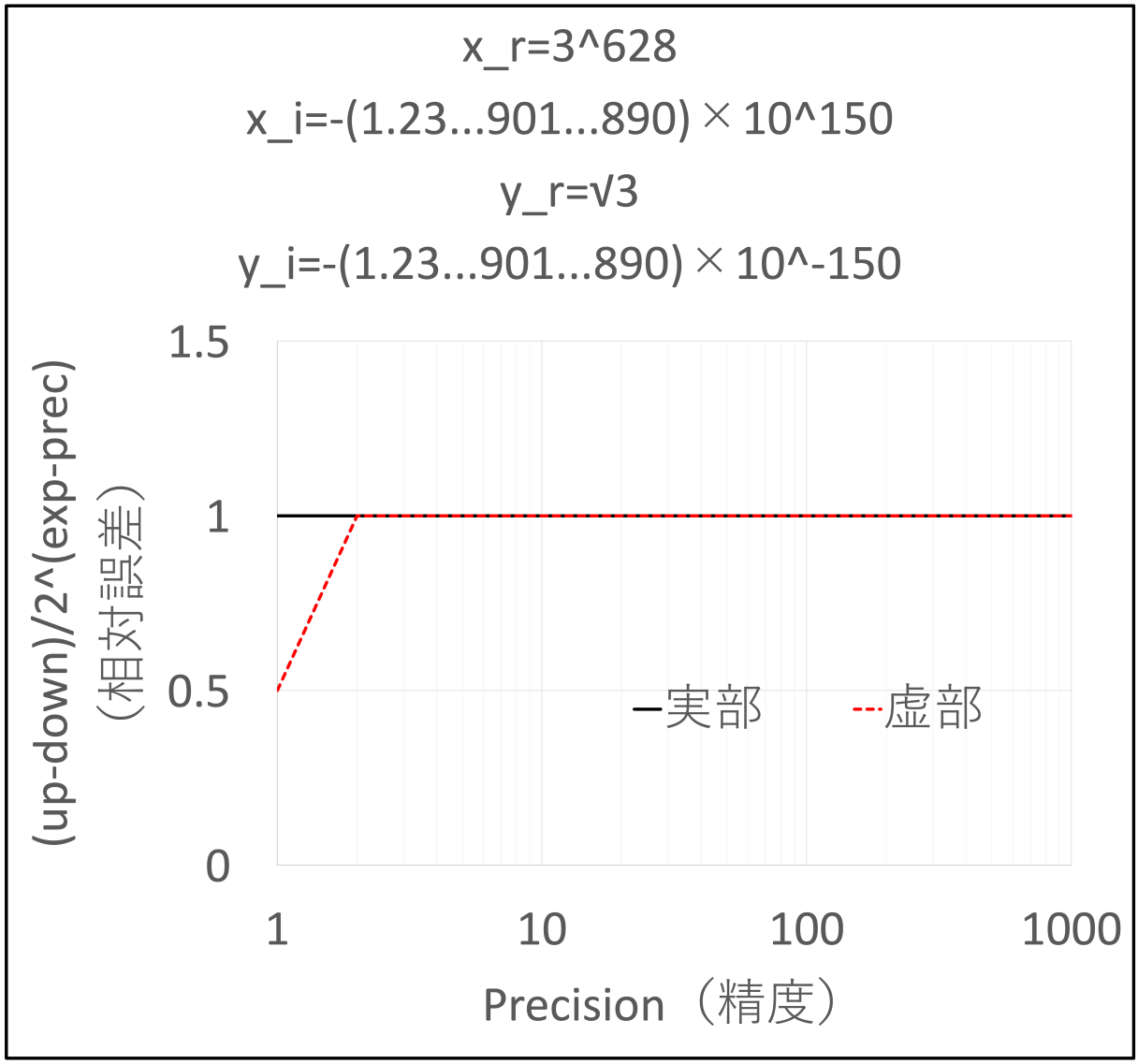

図 37: 精度に対する相対誤差の片対数グラフ（mic_micmic_div_1ulp を使用したときの複素数の除算）

x_r,x_i \ y_r,y_i	L_{+77}^{300},L_{+78}^{300}	S_{-75}^{302},S_{-75}^{303}
L_{+75}^{300},L_{+76}^{300}	3.0×10^{300}	3.0
S_{-75}^{300},S_{-75}^{301}	3.0	4.3×10^{300}

表 53: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 1 増やしたときの複素数の除算)

x_r,x_i \ y_r,y_i	L_{+77}^{300},L_{+78}^{300}	S_{-75}^{302},S_{-75}^{303}
L_{+75}^{300},L_{+76}^{300}	1.2×10^{301}	2.0
S_{-75}^{300},S_{-75}^{301}	2.0	1.1×10^{300}

表 54: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 2 増やしたときの複素数の除算)

x_r,x_i \ y_r,y_i	L_{+77}^{300},L_{+78}^{300}	S_{-75}^{302},S_{-75}^{303}
L_{+75}^{300},L_{+76}^{300}	1.5×10^{300}	2.0
S_{-75}^{300},S_{-75}^{301}	2.0	1.1×10^{300}

表 55: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 4 増やしたときの複素数の除算)

x_r,x_i \ y_r,y_i	$L^{300}_{+77},L^{300}_{+78}$	$S^{302}_{-75},S^{303}_{-75}$
$L^{300}_{+75},L^{300}_{+76}$	9.3×10^{298}	2.0
$S^{300}_{-75},S^{301}_{-75}$	2.0	1.1×10^{300}

表 56: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 8 増やしたときの複素数の除算)

x_r,x_i \ y_r,y_i	$L^{300}_{+77},L^{300}_{+78}$	$S^{302}_{-75},S^{303}_{-75}$
$L^{300}_{+75},L^{300}_{+76}$	3.6×10^{296}	1.0
$S^{300}_{-75},S^{301}_{-75}$	1.0	1.1×10^{300}

表 57: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 16 増やしたときの複素数の除算)

x_r,x_i \ y_r,y_i	$L^{300}_{+77},L^{300}_{+78}$	$S^{302}_{-75},S^{303}_{-75}$
$L^{300}_{+75},L^{300}_{+76}$	5.6×10^{291}	1.0
$S^{300}_{-75},S^{301}_{-75}$	1.0	1.1×10^{300}

表 58: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 32 増やしたときの複素数の除算)

x_r,x_i \ y_r,y_i	$L^{300}_{+77},L^{300}_{+78}$	$S^{302}_{-75},S^{303}_{-75}$
$L^{300}_{+75},L^{300}_{+76}$	3.5×10^{290}	1.0
$S^{300}_{-75},S^{301}_{-75}$	1.0	1.1×10^{300}

表 59: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 64 増やしたときの複素数の除算)

x_r,x_i \ y_r,y_i	$L^{300}_{+77},L^{300}_{+78}$	$S^{302}_{-75},S^{303}_{-75}$
$L^{300}_{+75},L^{300}_{+76}$	3.5×10^{290}	1.0
$S^{300}_{-75},S^{301}_{-75}$	1.0	1.1×10^{300}

表 60: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 128 増やしたときの複素数の除算)

x_r,x_i \ y_r,y_i	$L^{300}_{+77},L^{300}_{+78}$	$S^{302}_{-75},S^{303}_{-75}$
$L^{300}_{+75},L^{300}_{+76}$	1.0	2.0
$S^{300}_{-75},S^{301}_{-75}$	1.0	1.0

表 61: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 2 倍にしたときの複素数の除算)

x_r,x_i \ y_r,y_i	$L^{300}_{+77},L^{300}_{+78}$	$S^{302}_{-75},S^{303}_{-75}$
$L^{300}_{+75},L^{300}_{+76}$	1.0	1.0
$S^{300}_{-75},S^{301}_{-75}$	1.0	1.0

表 62: 精度が 1 から 1000 のときの相対誤差の最大値 (ビット数を 3 倍にしたときの複素数の除算)

x_r,x_i \ y_r,y_i	$L^{300}_{+77},L^{300}_{+78}$	$L^{300}_{+77},S^{302}_{-75}$	$S^{302}_{-75},L^{300}_{+77}$	$S^{302}_{-75},S^{303}_{-75}$
$L^{300}_{+75},L^{300}_{+76}$	1.0	1.0	1.0	1.0
$L^{300}_{+75},S^{300}_{-75}$	1.0	1.0	1.0	1.0
$S^{300}_{-75},L^{300}_{+75}$	1.0	1.0	1.0	1.0
$S^{300}_{-75},S^{301}_{-75}$	1.0	1.0	1.0	1.0

表 63: 精度が 1 から 1000 のときの相対誤差の最大値 (mic_micmic_div_1ulp を使用したときの複素数の除算)