5.4 誤差

この節では、**誤差**について説明します。誤差を考察することは、前々節で解説した**アルゴリズム**や前節で解説した**計算量**とあわせてプログラムを作成する上で非常に重要な項目となります。特に、数値計算と呼ばれる分野においては、正しい計算結果を得るために必ず考察される項目となっています。

さて、私たちは、円周率 π の値が $3.14159263\cdots$ と無限に続く数であることを知っています。ところが、例えば、直径 10cm の真円をぐるっと取り巻く線の長さを 20πcm と答えることは簡単ですが、実際にこの長さの紐を切り出すこと (具体的な数値で表すこと) は不可能です。そこで私たちは妥協して許容できる範囲の近似値を用いることになります。もちろん、このとき少なからず誤差を生じてしまうことになります。私たちがよく用いる近似法として次の 3 つが良く知られています (数値を上位から m 桁で近似する場合)。

- **切り捨て**： 上位から $m+1$ 桁以後を無視する。
- **切り上げ**： 上位から $m+1$ 桁以後が全て 0 でなければ m 桁目に 1 を加える。
- **四捨五入**： 上位から $m+1$ 桁の数が 5, 6, 7, 8, 9 ならば上位から m 桁目に 1 を加え (切り上げ)、 0, 1, 2, 3, 4 ならば加えない (切り捨て)。

情報科学ではこのような操作を「**丸め**」と呼び、このような操作で生じる誤差を「**丸め誤差**」と呼びます。なお、2.7 節で述べたように、コンピュータが直接扱うことができる数は、**有限**かつ**離散的**な数で、固定小数点表示によって表される整数と浮動小数点表示によって表される実数に限られます。したがって、コンピュータ上で動くプログラムもこの丸め誤差の呪縛から逃れられない宿命を背負っています。特に、最初に紹介した数値計算の分野は、浮動小数点表示による実数を多用するだけでなく、人間が遠く及ばない回数の演算を行うため、丸め誤差が積もり積もってしまうことになります。すなわち、アルゴリズムがいくら正しくても、正しい計算結果が得られるとは限らないということになります。

誤差について詳しく学習する前に誤差の基礎知識について解説しておきます。誤差の一般的な定義は、元の数値 (**真値**) とその**近似値**の差異によって与えられ、式で表せば

$$(\text{誤差}) = (\text{近似値}) - (\text{真値})$$

となります。なお、本テキストでは、誤差に ε, 近似値に a (アルファベット), 真値に α (ギリシャ文字) を用いることにします。すなわち、上式は

$$\varepsilon = a - \alpha$$

となります。また、誤差を考察する際に、誤差の符号を無視してその大きさ (絶対的な大きさ) で論じられることもしばしばあります。この誤差を**絶対誤差**[8] (Absolute error) と呼び、式で表せば

$$(\text{絶対誤差}) = |\,(\text{近似値}) - (\text{真値})\,|$$

[8]誤差は、真値と近似値の差異の大きさを論ずることが主目的なので、誤差の符号をあまり気にしないで論じられることもよくあります。そのため、誤差といえば絶対誤差を指すのが一般的です。どちらかというと、次に紹介する相対誤差の対語として用いられます。

となります。同様に、絶対誤差を記号 ε_A で表せば、上式は

$$\varepsilon_A = |\,\varepsilon\,| = |\,a - \alpha\,|$$

となります。これに対して、誤差を考察する際に、真値に対する誤差の割合 (相対的な大きさ) で論じられることもあります。この誤差を**相対誤差** (Relative error) と呼び、式で表せば

$$(\text{相対誤差}) = \left| \frac{(\text{誤差})}{(\text{真値})} \right|$$

となります。同様に、相対誤差を記号 ε_R で表せば、上式は

$$\varepsilon_R = \left| \frac{\varepsilon}{\alpha} \right| = \left| \frac{a - \alpha}{\alpha} \right|$$

となります (相対誤差においても誤差の符号をあまり気にしないことにします)。ただし、真値が 0 に非常に近い場合は相対誤差が無限大となるため、この様な場合は相対誤差を用いないことにします[9]。

ここで、四則演算 (加減乗除) の誤差について考察しておきましょう。なお、四則演算に使用する 2 つの真値を α, β とし、その近似値をそれぞれ a, b とします。また、近似値 a, b の誤差 (絶対誤差) をそれぞれ $\varepsilon_A(a), \varepsilon_A(b)$ とし、相対誤差をそれぞれ $\varepsilon_R(a), \varepsilon_R(b)$ とします。すなわち、記号で式を表せば、それぞれ

$$\begin{cases} \varepsilon_A(a) = a - \alpha \ (= |\,a - \alpha\,|) \\ \varepsilon_R(a) = \left| \dfrac{a - \alpha}{\alpha} \right| = \dfrac{\varepsilon_A(a)}{\alpha} \end{cases} \qquad \begin{cases} \varepsilon_A(b) = b - \beta \ (= |\,b - \beta\,|) \\ \varepsilon_R(b) = \left| \dfrac{b - \beta}{\beta} \right| = \dfrac{\varepsilon_A(b)}{\beta} \end{cases}$$

となります。このとき、近似値の和と差は

$$\begin{array}{lll} \text{近似値} & \text{真値} & \text{誤差} \\ a + b & = (\alpha + \beta) & + \left(\varepsilon_A(a) + \varepsilon_A(b)\right) \\ a - b & = (\alpha - \beta) & + \left(\varepsilon_A(a) - \varepsilon_A(b)\right) \end{array}$$

となり、近似値の和と差の誤差 (絶対誤差) は各近似値の誤差 (絶対誤差) の和と差になることがわかります。一方、少し複雑ですが、近似値の積と商は

$$a \times b = (\alpha \cdot \beta) \cdot \left(1 + \frac{\varepsilon_A(a)}{\alpha} + \frac{\varepsilon_A(b)}{\beta} + \frac{\varepsilon_A(a)}{\alpha} \cdot \frac{\varepsilon_A(b)}{\beta}\right)$$

$$a \div b = \frac{\alpha}{\beta} \cdot \left(1 + \frac{\varepsilon_A(a)}{\alpha}\right) \cdot \left(1 - \frac{\varepsilon_A(b)}{\beta} + \frac{\varepsilon_A(b)^2}{\beta^2} - \cdots\right)$$

となります。このとき、近似値の相対誤差 $\varepsilon_R(a), \varepsilon_R(b)$ が十分小さければ高次の項は非常に小さくなるので無視することができます (逆に、相対誤差が大きければ近似値の計算自体意味を持た

[9] この様な場合は絶対誤差で論じるしかありません。なお、誤差の考察に絶対誤差を用いるのか相対誤差を用いるのかは、その時々の状況に応じて選択する必要があります。もちろん、両方用いる場合もあります。

なくなるため、相対誤差は十分に小さいと仮定します)。したがって、近似値の積と商は

$$a \times b \fallingdotseq (\alpha \cdot \beta) \cdot \left(1 + \frac{\varepsilon_A(a)}{\alpha} + \frac{\varepsilon_A(b)}{\beta}\right)$$

$$a \div b \fallingdotseq \frac{\alpha}{\beta} \cdot \left(1 + \frac{\varepsilon_A(a)}{\alpha} - \frac{\varepsilon_A(b)}{\beta}\right)$$

となり、近似値の積と商の誤差 (相対誤差) は各近似値の誤差 (相対誤差) の和と差になることがわかります。以上、四則演算の誤差についてまとめると

- $\varepsilon_A(a+b) = \varepsilon_A(a) + \varepsilon_A(b)$ (和の絶対誤差)
- $\varepsilon_A(a-b) = \varepsilon_A(a) - \varepsilon_A(b)$ (差の絶対誤差)
- $\varepsilon_R(a \times b) = \varepsilon_R(a) + \varepsilon_R(b)$ (積の相対誤差)
- $\varepsilon_R(a \div b) = \varepsilon_R(a) - \varepsilon_R(b)$ (商の相対誤差)

となります。

誤差の考察において、**精度**と呼ばれるもう 1 つ重要な要素があります。精度は、真値と近似値がどれくらい似ているかを調べるための指標で、真値と近似値を比べたとき上位から各桁の値が一致する桁の数 (桁数) で表します。例えば、真値が `1.23123123` でその近似値が `1.23123108` であったとき、上位から 7 桁目まで一致しているので、近似値の精度は 7 桁ということになります。言い換えれば、近似値は 7 桁の精度を持つことになります。ただし、真値が `1.0000000` でその近似値が `0.9999999` であるような場合 (実際には誤差が非常に小さい場合)、精度を 0 桁としてしまうのは非常に不合理なので、正確には精度の定義である

$$(\text{精度}) = -\log_n (\text{相対誤差})$$

を計算します (底 n は相対誤差の基数を用い、精度は小数点以下を切り捨てる)。なお、この計算式で得られる精度は n 進数に換算された精度となります。

ここで、コンピュータが直接扱うことのできる浮動小数点表示によって表される実数の精度を求めておきましょう。精度は真値と近似値を比べたとき上位から各桁の値が一致する桁の数によって与えられましたから、浮動小数点表示によって表される実数の精度は仮数部のビット数によって決まることになります。さて、2.7 節でも紹介したように、現在私たちの身の回りに存在するコンピュータの浮動小数点表示は IEEE754 形式という規格を満たしています。IEEE754 形式には単精度浮動小数点表示と倍精度浮動小数点表示があり、仮数部のビット数はそれぞれ 24 ビットと 53 ビットとなっています。以後、単精度浮動小数点表示に絞って話を進めます。まず、次の 2 つの浮動小数点数 a と b を見比べてみましょう。

$a =$

-	--------	00000000000000000000000

∧
1.

$b =$

-	--------	00000000000000000000001

∧
1.

この 2 つの浮動小数点数を比較すると、a の仮数部と b の仮数部では最下位ビットが異なっています。すなわち、仮数部における誤差が 2 進数 0.00000000000000000000001 $(= 2^{-23})$ であるということがわかります。したがって、この差異を精度に関するものだと考えれば、a の仮数部 (真値) と b の仮数部 (近似値) の相対誤差は

$$\varepsilon_R = \left| \frac{(b \text{ の仮数部}) - (a \text{ の仮数部})}{(a \text{ の仮数部})} \right| = \left| \frac{(1 + 2^{-23}) - 1}{1} \right|$$

となり、浮動小数点表示によって表される実数の精度は

$$(\text{精度}) = -\log_{10} \varepsilon_R = -\log_{10} 2^{-23} = 6.923689\cdots$$

となります。すなわち、浮動小数点表示によって表される実数は 10 進数で 6 桁に相当する精度を持つことになります。なお、$6.923689\cdots$ は 7 に非常に近いため、概算で精度 7 桁として扱うことも多々あります[10]。最後に、単精度浮動小数点表示において 2^{-23} という数は、数と数の違いを特徴付ける相対的に最小の数 (相対誤差) となります。そのため、この相対誤差は**機械イプシロン**という呼び名で紹介され、コンピュータの性能を表す指標として用いられています[11]。

5.4.1　丸め誤差

丸め誤差は、2.7 節で紹介したように、ただ 10 進数から 2 進数に変換しただけで発生します。また、前節で紹介したように、1 つ 1 つの演算による丸め誤差は小さくても、コンピュータで何万回も四則演算を繰り返せば累積される誤差は膨大になり、正しい結果が得られるという保証は全く無くなってしまいます。そこで、以下のような場合には、特に注意してプログラムを組む必要があります。以後、IEEE754 形式の単精度浮動小数点表示によって表される (10 進数 7 桁の精度を持つ) 実数を対象に話を進めます。

丸め誤差の累積　例えば、真値 α_i $(i = 1, 2, 3, \cdots, n)$ に対して近似値 a_i が常にそれぞれ誤差 $\varepsilon = 0.000001$ を持っていると仮定し、近似値 a_i の総和を考えてみましょう。近似値 a_i の総和は

$$\sum_{i=1}^{n} a_i = (\alpha_1 + \varepsilon) + (\alpha_2 + \varepsilon) + \cdots + (\alpha_n + \varepsilon) = (\alpha_1 + \alpha_2 + \cdots + \alpha_n) + n \cdot \varepsilon$$

となり、n が大きくなれば丸め誤差の累積が大きくなることがわかります。ところで、n が 100 程度であれば真値 α_i の総和に対して近似値 a_i の総和は累積された丸め誤差 $n \cdot \varepsilon = 100 \times 0.000001 = 0.0001$ を持つことになります。したがって、精度の下位 2 桁が丸め誤差によって汚染され、精度の有効桁数が 5 桁に減ってしまいます。これはまだ我慢できるとして、n が 1000000 にもなれば累積された丸め誤差が $n \cdot \varepsilon = 1000000 \times 0.000001 = 1$ となり、精度の全ての桁が丸め誤差に汚染されてしまい、計算結果が全く意味のない値になってしまいます。このような丸め誤差の累積は、単精度浮動小数点表示よりさらに大きな精度を持つ倍精度浮動小数点表示を用いることで回

[10] 浮動小数点表示によって表される実数の精度は、単に仮数部のビット数を用いて、$(\text{精度}) = \log_{10} 2^{24} = 7.22471\cdots$ と計算する方法もあります。

[11] 最近は、ほとんどのコンピュータが実用計算に必要な精度を持つ IEEE754 形式の規格を満たしているため、「機械イプシロン」という言葉をあまり耳にしなくなりました。しかしながら、さらに精度が必要な数値計算や数学的に厳密な数を求めたい場合は必ず「機械イプシロン」を考慮しなければならないことに注意してください。

避することができます。また、丸め誤差の累積とは関係ありませんが、一見したところ高精度の計算が必要でない計算でも、予期しない誤差が発生する場合があるので、安全のため高精度の倍精度浮動小数点表示を用いるようにしましょう[12]。したがって、今後行う C 言語によるプログラミング演習でも、単精度浮動小数点表示 (`float`) ではなく、倍精度浮動小数点表示 (`double`) を常に使用することにします。

積み残し 例えば、精度 7 桁の 2 つの数 $a = 123.4567$ と $b = 0.000001111111$ の和を考えて見ましょう。このとき 2 つの数の和は

$$a + b = 123.4567 + 0.000001111111 = \underline{123.4567}01111111$$

となります。ところが、単精度浮動小数点表示で表される実数は精度が 7 桁であることから、計算結果は下線の 123.4567 となってしまい、全く加算されていないことになります。このような誤差を**積み残し**と呼び、絶対値が極端に違う数を加算する場合に発生します。そこで、例えば、n 個の数 a_i $(i = 1, 2, 3, \cdots, n)$ の総和を計算する場合などは、数 a_i を絶対値の小さい順 $a'_1, a'_2, a'_3, \cdots, a'_n$ $(|a'_1| \leq |a'_1| \leq |a'_3| \leq \cdots \leq |a'_n|)$ に並び替えて、絶対値の小さな数の順に加算すれば積み残しを軽減することができます。

桁落ち 例えば、精度 7 桁の 2 つの数 $a = 1.234567$ と $b = 1.234561$ の差を考えて見ましょう。このとき 2 つの数の差は

$$a - b = 1.234567 - 1.2345561 = 0.000006\underline{******}$$

となります。ところが、単精度浮動小数点表示で表される実数は精度が 7 桁であることから、計算結果は下線の 6****** となってしまい、精度が極端に落ちてしまいます。このような誤差を**桁落ち**と呼び、絶対値が同じような数を減算する場合に発生します。そこで、例えば、a の真値 α が 1.2345671234567 で、b の真値 β が 1.2345612345612 であったとするなら、あらかじめ近似値を $a' = \alpha - 1.23456 = 0.000007123456$ および $b' = \beta - 1.23456 = 0.000007123456$ と変形しておき、

$$a' - b' = 0.000007123456 - 0.000001234561 = 0.000005\underline{888895}$$

を計算すれば、精度の高い計算結果を得ることができ、桁落ちを軽減することができます。また、問題によっては、計算のやり方が複数あって、その中に桁落ちの起こらない方法が見つかる場合もあります (例えば 2 次方程式の解)。

[12] その昔、浮動小数点数の演算時間は普通の演算時間の数倍かかったため、できるだけ使わないようにという風潮がありました。しかしながら、現在では、CPU の進歩によって浮動小数点数の演算時間も普通の演算時間も同一の時間で実行できるようになったため、浮動小数点数を気兼ねなく使えるようになった。