プログラミングテクニック

物理をベースとしたコンピュータシミュレーションは数値計算の技術が必須となる。有名な手法としては方程式の根を求めるための２分法やニュートン法、積分計算を行うための台形則やシンプソンなどがある。

数値計算を行うときに、考慮すべきことは、その計算を行う上での精度と実行速度である。手法によって、精度は良いが実行速度が遅かったり、実行速度は速いが精度が悪かったりするといった特徴がある。そのため、解こうとしている問題において、どちらを重視するかが異なってくる。例えば、コンピュータグラフィックスの分野では評価は見た目にある。作成しようとする画像の品質が達成できる程度の精度が得られるのであれば、実行速度を優先することで、映画などでは製作コストを下げることができたり、ゲームであれば、リアルタイムで計算を行うことが可能になり、表現の幅が広がったりする。

数値計算上の精度と速度について述べたが、一方で、数値計算をプログラミングするにあたっては、計算誤差についての知識は重要である。なぜなら、プログラミングをする上で、バグが入りやすくなる部分の１つがこの計算誤差にあるからだ。

コンピュータ内では扱う値は有効桁数が有限である。そのため、計算で得られた値と真値との間に差ができてしまう。これを**計算誤差**あるいは単純に**誤差**という。計算誤差は大きく分けて次の４つである。

- 丸め誤差
- 情報落ち
- 桁落ち
- 打ち切り誤差

今回は数値計算をプログラミングする上で考慮しなくてはならない計算誤差について学習する。

浮動小数点表現

計算誤差の説明の中で有効桁数について触れた。有効桁数について、浮動小数点表現を例にとり、説明を行っていく。

実数の表現

実数 6.25 を、有効数字を表す「仮数部」と桁数を表す「指数部」を用いた式で表すものとする。ただし、「仮数部」は１以下の値であるとすると、以下のように表せる。

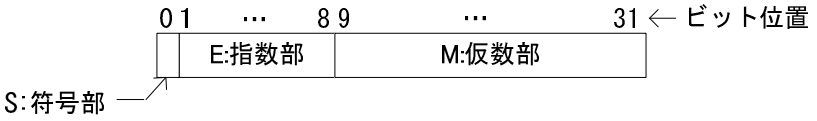
$$6.25 = 10^{1} \times 0.625$$

指数部　仮数部

この時の 0.625 を仮数、10 を底、1 を指数という。任意の実数はこのようにして、指数部と仮数部を用いた式で表現することができ、この表現方法を浮動小数点表現という。小数点を固定した表現方法に比べ、同じビット数ならば表現できる数が多くなるメリットがある。

2 進数浮動小数点表現と正規化

以下の図は 4 バイト（32 ビット）で実数を表すものとした 2 進数浮動小数点表現の例である。



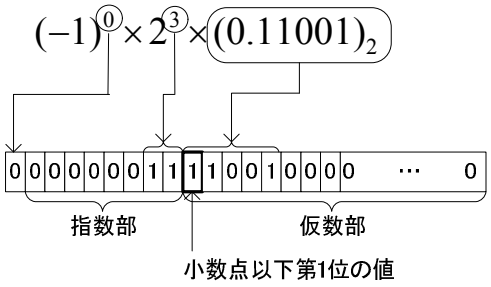
0 ビット目の「S:符号部」は実数の正負を表し、0 ならば正、1 ならば負となる。1～8 ビット目は「E:指数部」であり、実数の桁数を表す。9～31 ビット目は「M:仮数部」であり、実数の有効数字を表す。この仮数部は１以下の値で表す。また、これらの値は全て 2 進数で表現される。式で示すと以下のようになる。

$$(-1)^{S} \times 2^{E} \times M \qquad (1)$$

10 進数 6.25 をこの浮動小数点表現で表すとすると、S は 6.25 が正なので 0 となる。次に E と M であるが、6.25 を固定小数点表現で表すと、$(110.01)_2$ となる。仮数部を表すには、これを 1 以下にしなくてはならない。そのため、整数を表現している部分の最上位桁が、小数を表現している部分の最上位桁になるように、桁合わせを行う。そのためには、3 ビット右シフトした上で 2^3 を乗じればよい。

$$\begin{aligned} 6.25 &= (110.01)_2 \\ &= 2^0 \times (110.01)_2 \\ &= 2^3 \times (0.11001)_2 \end{aligned}$$

よって、E は 3 であり、M は$(0.11001)_2$ となるので、(1)の形式では



となる。仮数部の整数を表現している部分を 0 にし、小数点以下第 1 位を 1 となるよう桁合わせを行うことを正規化という。正規化の目的は有効桁数を最大に保つことにある。例えば、6. 25 は

$$= 2^4 \times (0.011001)_2$$

でも、表現することができるが、小数点以下第 1 位が 0 であるため、正規化されていない状態であり、この場合には表現できる桁数が 2 進数で 1 桁分失っている状態である。

有効桁数

正規化を行った浮動小数点の有効桁数は仮数部のビット数から求めることができる。上で述べた形式であれば、仮数部は 23 ビットであるので、これを 10 進数に変換した時の桁数が有効桁数となる。

10 進数に変換した時の有効桁数を x とすると、

$$2^{23} = 10^{x}$$

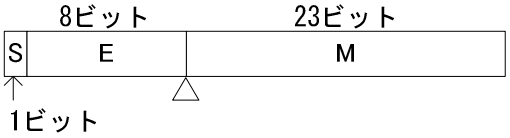
底を 10 として両辺の対数をとると、

$$\begin{aligned} &\log_{10} 2^{23} = \log_{10} 10^{x} \\ &23 \times \log_{10} 2 = x \times \log_{10} 10 \\ &x = 23 \times \log_{10} 2 \\ &x = 23 \times 0.301 \\ &x = 6.923 \end{aligned}$$

となる。実際の有効桁数は小数点以下を切り捨てたものとなるので、この場合の有効桁数 x は 6 となる。

IEEE 単精度の浮動小数点表現

IEEE（米国電気電子技術者協会）標準の浮動小数点表現では、単精度の数値は 32 ビットを用いて以下で表される。符号部、指数部、仮数部の区切りは先ほど示した例と同じであるが、指数部と仮数部の取り扱いが異なる。



S：符号部　　整数および 0 のとき 0、負数のとき 1
E：指数部　　127 増し符号の 2 進数整数
M：仮数部　　2 進数表示で左端の△が小数点位置

※127 増し符号とは実際の累乗数は指数部で表された値から
127 をひいたものであることを示す。

$$(-1)^{S} \times 2^{E-127} \times (1.M)_2 \qquad (2)$$

(2)式では$(1.M)_2$の 1 は実際のビット列には存在しない。IEEE 形式では正規化を行うと、仮数部の 1 ビット目が必ず「1」になることに着目し、有効ビット数を 1 ビットでも多くするため、仮数部を正規化した状態から 1 ビット左シフトし、この「1」が必ず存在するものとして隠している。IEEE 形式の場合、この隠された「1」を含めて読み取る必要があるので注意が必要である。

計算誤差

計算誤差には以下の 4 つがある。

丸め誤差

有効桁数で演算を行うため、下位の桁を削除することで、発生する誤差。場合によって、切り捨て、切り上げ、四捨五入が行われる。

例：10 進数 0.1 を 2 進数で表すと、10 進数では有限桁であるが、2 進数では循環小数 $0.1 = (0.0001100110\cdots)_2$ となり、無限小数となってしまう。仮に小数点以下 5 桁までしか表現できないとすると、$(0.00011)_2 = 0.09375$ となるため、丸め誤差が発生する。ここで、誤差には以下に示す絶対誤差と相対誤差がある。

$$\text{絶対誤差} = |\text{近似値} - \text{真値}| = |0.09375 - 0.1| = 0.00625$$

$$\text{相対誤差} = \frac{|\text{近似値} - \text{真値}|}{\text{真値}} = \frac{|0.09375 - 0.1|}{0.1} = 0.0625$$

情報落ち

絶対値の非常に大きい数と小さい数との間で、加減算を行った時、小さい数が演算結果に反映されなくなるために発生する誤差。

例：指数部の大きく異なる 2 つの数の加減算を行う場合には、指数部の小さい数が指数部の大きい数値に合わされてから演算が行われる。この場合、指数部の小さい数は仮数部の値が右シフトされることで情報落ちが発生する。仮に有効桁数が 4 桁までしか表現できないとすると、

$$\begin{aligned} 1425 + 1.297 &= 0.1425 \times 10^4 + 0.1297 \times 10^1 \\ &= 0.1425 \times 10^4 + 0.0001297 \times 10^4 \\ &= 0.1425 \times 10^4 + 0.0001 \times 10^4 \quad \text{（有効桁数 4 より情報落ちが発生）} \\ &= 0.1426 \times 10^4 \end{aligned}$$

対策：いくつかの数値で加減算を行う際には、情報落ちをできるだけ防ぐために、絶対値が小さい順に数値をソートし、指数部が小さい順に演算を行うことで、あまり絶対値の差のない演算にするよう工夫する。

桁落ち

絶対値のほぼ等しい数との間で、減算を行った時、有効桁が減ってしまうために発生する誤差。

例：仮に有効桁数を 6 桁として、$\sqrt{133}=11.5326$ と $\sqrt{131}=11.4455$ との減算を考える。今、これらの数値は小数点以下 4 桁まで正しい数値である。減算を行うと、以下のようになり、

$$\begin{aligned}\sqrt{133}-\sqrt{131} &= 11.5326-11.4455\\ &= 0.0871\end{aligned}$$

計算結果を有効桁数 3 桁となり、桁落ちが発生する。

対策：このような桁落ちを防ぐため、分子の有理化を行ってから計算する。

$$\begin{aligned}\sqrt{133}-\sqrt{131} &= \frac{\left(\sqrt{133}-\sqrt{131}\right)\times\left(\sqrt{133}+\sqrt{131}\right)}{\sqrt{133}+\sqrt{131}}\\ &= \frac{133-131}{\sqrt{133}+\sqrt{131}}\\ &= \frac{2}{22.9781}\approx 0.0870394\end{aligned}$$

計算結果より、有効桁数 6 桁が確保されたことがわかる。

打ち切り誤差

数値計算を行う上で、浮動小数点の計算処理の打ち切りについて、指定した規則で行うために発生する誤差。

例：方程式 $f(x)=0$ の解を求めるアルゴリズムとして、ニュートン・ラフソン法がある。このアルゴリズムは曲線 $f(x)$ において、任意の点 x_0 での接線の傾きを利用することにより、真値に収束させようとする手法である。

いま、整数 n の平方根 $\sqrt{n}$ の近似値を求めるため、$f(x)=x^2-n$ として、$f(x)=0$ を満たす x をニュートン・ラフソン法で求めることを考える。以下の左図はフローチャートである。

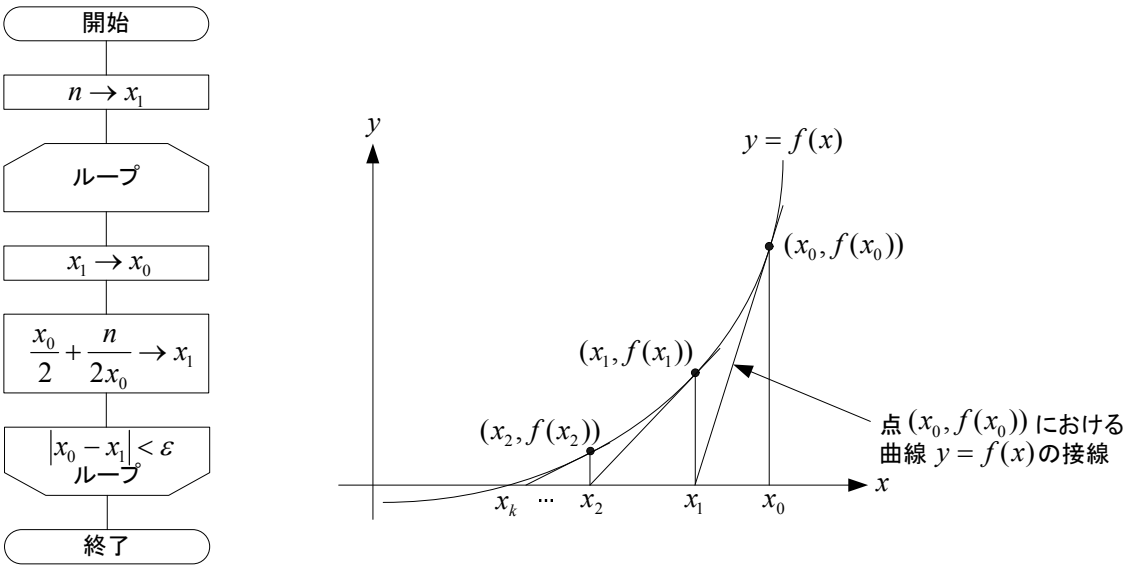


図 1　ニュートン・ラフソン法による近似根の求め方

点 $(x_0, f(x_0))$ における曲線 $y=f(x)$ の接線と x 軸との交点を求め、求めた x_1 を x_0 に設定し直し、

条件$|x_0 - x_1| < \varepsilon$を満たすまで処理を反復する。フローチャートの記述ではループの「終了条件」を記述するので$|x_0 - x_1| < \varepsilon$となるが、C 言語ではループの繰り返しの「継続条件」を記述することになるので、その場合には$|x_0 - x_1| > \varepsilon$となることに注意すること。$\varepsilon$は十分に小さい正の値を意味し、一般には$10^{-3}$、$10^{-6}$が設定される。設定する$\varepsilon$の値がある程度小さければ、$\sqrt{n}$の近似値が求まる。しかし、$\varepsilon$の値が大きい場合には早い段階で計算が打ち切られるため、真値との間の誤差が大きくなってしまう。これを打ち切り誤差という。

備考：曲線$y = f(x_n)$の接線の傾きは$f(x_n) = {x_n}^2 - n$をx_nで微分した$f'(x_n) = 2x_n$である。今、接点を$(x_0, f(x_0))$とすると、その接線の傾きは$f'(x_0) = 2x_0$であるので、接線は以下の式で表すことができる。

$$y = 2x_0(x - x_0) + f(x_0)$$

この接線がx軸と交わる点は$(x_1, 0)$であるので上式に代入して、

$$2x_0x_1 - 2{x_0}^2 + f(x_0) = 0$$

ここで、$f(x_0) = {x_0}^2 - n$を代入すると

$$2x_0x_1 - 2{x_0}^2 + ({x_0}^2 - n) = 0$$

x_1について解くと、

$$x_1 = \frac{x_0}{2} + \frac{n}{2x_0}$$

となる。

参考文献

この文章は以下の文献を参考にしました。

１．数値計算　高橋大輔　著　岩波書店
２．ソフトウェア開発技術者合格教本　大滝みや子　岡島裕史　共著　技術評論社

課題 2-1～2-3 以下のプログラムを作成しなさい。また、理解を深めるため、適宜プログラムを変更し、出力を確かめなさい。

```
/* kadai2-1.c 丸め誤差を確認するプログラム
   float、doubleで0.1はいくつになっているかを確認する */
#include <stdio.h>

int main(void)
{
        float  f = 0.1;
        double d = 0.1;

        printf("0.1 => float  : %.20f¥n", f);
        printf("0.1 => double : %.20f¥n", d);

    return 0;
```

```
}
```

```
/* kadai2-2.c 情報落ちを確認するプログラム
   大きい値と小さい値での加減算                    */
#include <stdio.h>

int main(void)
{
        double a, b;

        a = 1.0e+10;
        b = 1.0e-10;

        printf("a   = %22.10f¥n", a);
        printf("b   = %22.10f¥n", b);
        printf("a+b = %22.10f¥n",  a+b);
        printf("a-b = %22.10f¥n",  a-b);

    return 0;
}
```

```
/* kadai2-3.c 桁落ちを確認するプログラム */
#include <stdio.h>

int main(void)
{
        double a, b;

        a = 1.234567;
        b = 1.234566;

        printf("a   = %.6e¥n", a);
        printf("b   = %.6e¥n", b);
        printf("a-b = %.6e¥n", a-b);

        return 0;
}
```

※以上のプログラムは VisualStudio2008 上で動作確認を行った。

課題 2-4　ニュートン・ラフソン法で 3 の平方根を求めるプログラムを作成しなさい。ただし、打ち切り誤差について確認するため、終了条件で用いる値 ε を 10^{-1}、10^{-2}、10^{-3}、10^{-4}、10^{-5}、10^{-6} として、出力を確かめるものとする。3 の平方根は 1.732050807568877…である。小数点以下の表示を 15 桁として確認しなさい。

※各結果については考察を含めるものとする。