2. 数値計算の基本(誤差と計算量)

現在のコンピュータは突き詰めればトランジスタのオンとオフの 2 進数なので、無限に小さな数や大きな数を制限なく使えるわけではなく、扱える数にいろいろな制約があり、そのための誤差が発生する。数値計算では、解析的に解けない(答えが:厳密には分からない)問題を扱うので、解が妥当なのか、間違っているのかを評価する過程が非常に重要になる。ここでは、評価の指針となる誤差の問題について、少しまとめて考える。ここではプログラム言語として Octave を使った例を示しているが、Octave に特有というわけではなく、どの言語でも同じように誤差は発生する。一般に、誤差はプログラム言語にはあまり関係がなく、ハードウェアの制約、アルゴリズム(解法)から生じることが多い。

2.1. 表現の制約

コンピュータはトランジスタのオンとオフが基本になっているので、実際コンピュータが扱う数値は、すべて 0 と 1 の 2 進数になっている。現在、ほとんどのコンピュータでは、IEEE 754 というアメリカ電気学会(IEEE)の浮動小数点演算規格に沿って数値を表現している。浮動小数点形式には二種類あり、単精度(single)は 32 ビット(2^{32} 通り)、倍精度(double)は 64 ビット(2^{64} 通り)が割り当てられている。単精度は数値の精度が倍精度に比べて荒いが、メモリの使用量が少なくて済むので、計算が高速になる。逆に、倍精度は、数値の精度は高いが、メモリの消費が多くなり計算量が多くなって、通常は計算速度が遅くなる。したがって、精度の要らない変数は、なるべく単精度にし、精度の必要な変数は倍精度にすることが一般的。

IEEE 754 では、ある数値 a を$a = \mathrm{s} \times \mathrm{M} \times 2^{\mathrm{e}}$という二進数で表記し、s を正負を表す符号ビット、M を仮数部、e を指数部として、それぞれビットを次のように割り当てている。

	符号 s	仮数部 M	指数部 e
単精度(32 ビット)	1 ビット	23 ビット	8 ビット
倍精度(64 ビット)	1 ビット	52 ビット	11 ビット

倍精度の場合は、仮数部 M は、2 進数で、52 桁つまり 10 進数では $\log_{10}(2^{52})$=15.6 桁にも及ぶ数字を表記することができるので、十分大きな数が扱えるような気もするが、数値計算を精密に計算しようとすると、桁の制限があることで思わぬ障害に出くわすことがある。

(例) n!を考えるとき、コンピュータで扱うことのできる n の大きさはどれくらいか?

ベクトルの要素をすべてかけ算する(総積Π)関数 **prod** があるので、それを使って n!を求めてみる。

1. 特に何も工夫せずに表した場合

```
> n=170;
> N=1:n;     %N は要素が 1 2 3... n のベクトル
> prod(N)    %ベクトルの要素の総積
ans = 7.2574e+306
```

のように、n=170=7.2574×10^{306} は扱えるが、n=171 にすると、

```
> n=171;  prod(1:n)
ans = Inf
```

となり、**Inf**(無限)になってしまう。

2. 対数で扱う場合

大きな数を扱うには、対数を使うのが基本。$n!=n(n-1)...1$ なので、両辺の log(常用対数)をとり、

$$\log(n!) = \log(n) + \log(n-1) + \ ... + \log(2) + \log(1)$$

$$= \sum \log(n)$$

として扱う。常用対数を表す関数は **log10** なので、これを使う。

```
> n=171;
> d=sum(log10(1:n))    %sum で合計をとる
d = 309.09
```

となり、$\sum \log(n) = 309.09$となる。ここで、

```
> 10^d      % 10^(309.09)のこと
ans = Inf
```

として、そのまま n!を表示しようとすると、**Inf** になって意味がない。そこで $10^{309.09}$ の指数部(10^{309})と仮数部($10^{0.09}$)を分離して計算する。$10^{309.09}=10^{0.09}\times10^{309}$

```
> 10^(d-309)  % 仮数部
ans = 1.2410
```

より正確に求めたければ、

```
> format long
> 10^(d-309)
ans = 1.24101807021760
```

したがって、171! ~1.2410 … × 10^{309}と扱うことができる。ただし、両辺の間には誤差が生じてしまっている。

2.2 誤差

誤差とは、近似値や計算値が真の値からどれくらい差があるかを表す量だ。一般的に誤差には絶対誤差と相対誤差(誤差率)という表現の仕方(定義)がある。真の値を a、近似値を x とすると、絶対誤差は、

$$\Delta = x - a$$

相対誤差は、

$$\Delta_R = \Delta / a$$

で定義される。数値計算では真の値 a が分からない場合も多いので、理論値や平均値あるいは近似値で代用することもある。

2.3 丸め誤差

(例) αが 1 に比べて小さい場合、1 + αが 1 でない結果が得られる最小の値を考える。

たとえば、$\alpha = 10^{-15}$ の場合は、

```
> 1 + 1e-15
ans = 1.0000
```

となり、小数点以下に **0000** が表示されるが、$\alpha = 10^{-16}$ の場合は、

```
> 1 + 1e-16
ans = 1
```

となり、小数点以下の **0000** が消えていて、微妙に答えが違う。これは、コンピュータが **1+1e-15** は「1 ではないが、1 に非常に近い数値(=1.000)」として処理しているのに対し、**1+1e-16** は「厳密に 1」として処理していることに起因している。倍精度の場合、仮数部に 52 ビット割り当てられているので、1 から次に大きな数までの距離は、$2^{-52} \approx 2.22 \times 10^{-16}$ になる。これを機械誤差(ε)という。1×10^{-16} は機械誤差より小さいので、**1+1e-16** は厳密に 1 と見なされてしまう。これを丸め誤差という。Octave ではこの機械誤差に、**eps** という組み込み関数が割り当てられている。

```
> format long ;  2^(-52)
ans = 2.22044604925031e-016
> eps
ans = 2.22044604925031e-016
```

2.4 桁落ち

(例) 極端に桁の違う数値の計算

たとえば、$10^{16}+1-10^{16}$ は、言うまでもなく厳密に 1 だが、これをコンピュータに計算させてみると、

```
> 10^16 +1  -10^16
ans = 0
```

0 という明らかに間違った答えを出す。だが順番を変えて、

```
> 10^16 -10^16 +1
ans = 1
```

とすると、正しい答え(=1)になる。これは、初めの例では、$10^{16}+1$ を行った時点で、$10^{16}+1=10^{16}$ と解釈されて(桁落ちして)しまうため、そこから 10^{16} を引けば 0 になってしまうからだ。

(例) $\frac{1}{\sqrt{a+b}-\sqrt{a}}, \frac{\sqrt{a+b}+\sqrt{a}}{\mathrm{b}}$

という二式は $a,\ b>0$ のとき同値である(分子分母に$\sqrt{a+b}+\sqrt{a}$をかけると分かる) しかし、a と b の差が極端に大きいとき、コンピュータに計算させるとどうだろうか? $a=1,\ b=10^{-14}$ のとき、両者を評価してみよう。

```
> a=1; b=1e-14;
> L=1./(sqrt(a+b)-sqrt(a))
L = 2.047e+014
> R=(sqrt(a+b)+sqrt(a))./b
R = 2.000e+014
```

このように、**L** と **R** で、答えが 2%程度違ってくる。どちらがより真の値に近いのだろうか、考えてみてほしい。

このように非常に近い値の引き算を取ると、計算の精度が極端に落ちる場合がある。その場合、計算の順番を変えることで精度を保つことが出来る場合がある。

(例) 関数 $f(x)$の微分係数 $f'(x)$の数値解を求める。微分係数の定義 $f'(x)$は、

$$f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$$

なので、h を限りなく小さくすれば、数値解の精度も限りなく上がりそうな気がする。本当にそうだろうか??

例として、$f(x)=x^3$、$x=1$ のとき、絶対誤差

$$\Delta(h) = \left| f'(x) - \frac{f(x+h) - f(x)}{h} \right|$$

を評価してみよう。(error1.m)　h を等間隔のベクトルにすると、評価しづらいので、$\log_{10}(h)$を取ったときに等間隔になるように、$h=10^a$ として、a を等間隔のベクトルにしよう。

```
% error1.m
a=0:-0.2:-20;
h=10.^a;   %h をベクトルとして定義
x=1;       %定義
delta=abs(3*x.^2 - ((x+h).^3-x.^3)./h);
loglog(h,delta,'ro-');  %両対数プロット
xlabel('h');
ylabel('¥Delta(h)');
```

h をベクトルで指定して、5 行目で **delta** をベクトルのまま求めているところがポイント。**abs** は絶対値を返す関数。求めた **delta** を **h** の関数として、両対数のグラフ(**loglog** プロット)で表示する。これを実行すると、下の図が得られる。(y 軸のラベルにあるΔは**¥Delta** で表せる)

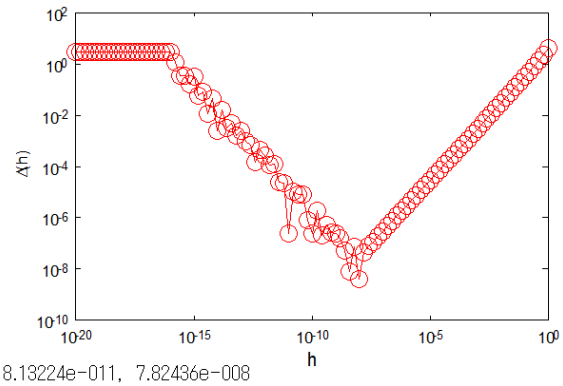

図 2.4

図 2.4 を見ると、h を 1 から小さくしていくと、絶対誤差Δが直線的に減少する。つまり計算の精度は上がっている。しかし、h～10^{-8} より小さくすると、絶対誤差Δが増加し始めていく。さらに、10^{-16} 以下ではΔが一定値(=3)になっていて、せっかく計算した答えには何の意味もなくなってしまうことが分かる。数学的に $h\to 0$ で定義されているからといって、何も考えずに h を小さくしてしまうと、思わぬ計算ミスに繋がるので注意が必要だ。

2.5 打ち切り誤差

コンピュータの表現の制約による誤差の他に、アルゴリズム上の制約によって誤差が生じる場合がある。常微分方程式で学ぶ予定のオイラー法は、テイラー展開の二次以上の項を無視するという近似を使っている。このようとき生じる誤差を打ち切り誤差という。高次の項を含めれば計算はより正確になるが、ある程度のところで打ち切ってしまわないと、計算量が膨大になってしまうので、打ち切り誤差をある程度容認したのがオイラー法というわけだ。

2.6 計算量

数値計算プログラムの善し悪しは、誤差の少なさだけで決まるように思うかもしれないが、そうとも限らない。たとえば、あまりにも誤差を少なくすることを狙いすぎたために、精度はそれほど上がらないにも関わらず、計算量だけが増えて答えを得るのに非常に時間がかかるという結果に陥ることがある。なるべく誤差の少ない計算を目指すのは当然としても、なるべく計算時間が少なくてすむ(少ない時間的リソース)、あるいは必要なメモリが少なくてすむ(少ない物理的リソース)ような計算を目指す必要もある。

(例)　n 次多項式

$$f(x)= a_1+ a_2x+ a_3x^2+ \cdots\ a_nx^{n-1}$$

の計算を行いたい。どういう手順(アルゴリズム)で計算したらいいだろうか?

方法 1. 単純に考えて、まず浮かぶのが、a_2x を求め、 a_3x^2 を求め、.... a_nx^{n-1} を求める。つまり先に $n-1$ 個の項それぞれについて計算し、最後に和を取るという方法だ。この場合、x^2 から x^n まで求めるのに、$n-1$ 回かけ算をし、それぞれの項に係数 a_2、a_3 ･･･ a_n をかけるのに、$n-1$ 回かけ算をし、さらに n 回足し算を行う。つまり、計算量は、$2n-2$ 回のかけ算と、n 回の足し算になる。

方法 2. まず $h_1=a_nx+ a_{n-1}$ を求め、次に $h_2= h_1x+ a_{n-2}$ を求め...というのを $n-1$ 回繰り返すという方法。この方法では一回の足し

算と一回のかけ算を $n-1$ 回行うので、計算量は $n-1$ 回のかけ算と、n 回の足し算になる。

方法 1 と 2 を比べると、方法 2 の方が計算量が少ないことが分かる。本当に方法 2 の方が速く計算が出来るのか? 実際にパソコンに計算させて比較してみよう。係数 a_n はランダムな数にすることにして、x=0.3 のとき、10^6 次という巨大な多項式を計算する。Octave には、**cputime** という関数が用意されているので、これを使って、計算する直前と直後の **cputime** の差を取ると、実際に計算に要した時間が求められる。

```
% polynomial.m
clear;  n=10^6; x=0.3;
a=rand(1,n); %rand を使って n 個の係数を作る。
%  方法 1
before=cputime; %開始前の時間を計測
for k=1:n
    f(k)=a(k)*x^(k-1);
end
answer=sum(f)
time=cputime-before    %かかった時間を求める

% 方法 2
h=a(n)*x+a(d-1);
before=cputime;
for k=n:-1:3
    h=x*h + a(k-2);
end
answer=h
time=cputime-before
```

方法 2 の方は、下から 4 行目の **h=x*h + a(k-2)** とすることで、for ループの一回前の古い **h** (右辺)で、新しい **h** (左辺)をメモリ上で毎回上書きしているので、使用しているメモリも少くなる。

これを実行すると、

```
> polynomial
answer = 1.1779
time = 12.745
answer = 1.1779
time = 5.5692
```

というような結果になった。(乱数を使っているので、**answer** と **time** の値は毎回変わる) 答えは方法 1,2 で同じだが、方法 1 は、12.7 秒程度かかり、方法 2 では、5.5 秒しかかかっていないことが分かる。

プログラムを書くことに慣れてきたら、計算量が少なくて済むか、メモリを消費しないようにしているか、考えながらプログラムを書くようにしてほしい。