

数値解析 (塩田)

— 誤差の発生メカニズム —

(1) 丸め誤差

全ての数値は「コンピュータで扱える数値」に近似される。

例 1 $e^{\pi\sqrt{163}}$ は、本当は無理数

262537412640768743.99999999999250072597198185688879...

なのだが、十分な精度がないと

262537412640768744

という整数のように見えてしまう。C 言語の double 型で計算すると

```
/* expsqrt163pi.c */
#include <stdio.h>
#include <math.h>
int main(){
    printf("exp(sqrt(163) * M_PI) = %lf", exp(sqrt(163) * M_PI));
    return 0;
}
```

exp(sqrt(163) * M_PI) = 262537412640768256.000000

のようにそもそも有効数字が足りない。python で精度を変えて計算してみれば

```
# expsqrt163pi.py
from decimal import *
for p in range(10, 61, 5):
    getcontext().prec = p
    pi = Decimal('3.141592653589793238462643383279502884197169399(以下略)')
    x = Decimal('163')
    y = (pi * x.sqrt()).exp()
    print(u' 精度 =', p, ": ", y)
```

```
精度 = 10 : 2.625374097E+17
精度 = 15 : 2.62537412640764E+17
精度 = 20 : 262537412640768744.17
精度 = 25 : 262537412640768743.99999990
精度 = 30 : 262537412640768744.000000000024
精度 = 35 : 262537412640768743.9999999999924980
精度 = 40 : 262537412640768743.99999999992500725944
精度 = 45 : 262537412640768743.9999999999250072597198209
精度 = 50 : 262537412640768743.999999999925007259719818568865
精度 = 55 : 262537412640768743.99999999992500725971981856888793537
精度 = 60 : 262537412640768743.9999999999250072597198185688879353856337
```

となって整数ではないことがわかる。

(2) オーバーフロー

大き過ぎる数は仕様の限界を超えて overflow のエラーを起こす。

例 2 double 型で 10 のべき乗を順番に求めてゆくと、やがて無限大を表す inf が表示される。

```

/* error_overflow10.c */
#include<stdio.h>
int main(){
    double x = 1.0;
    for(int i = 1; i <= 311; i++){
        x *= 10.0;
        printf("10^%d = %e\n", i, x);
    }
    return 0;
}

```

```

10^306 = 1.000000e+306
10^307 = 1.000000e+307
10^308 = 1.000000e+308
10^309 = inf
10^310 = inf
10^311 = inf

```

(inf は無限大 infinity のこと)

例 3 int 型で 2 のべき乗を求めてゆくと ...

```

/* error_overflow2.c */
#include<stdio.h>
int main(){
    int x = 1;
    for(int i = 1; i <= 35; i++){
        x *= 2;
        printf("2^%2d = %d\n", i, x);
    }
    return 0;
}

```

```

(中略)
2^29 = 536870912
2^30 = 1073741824
2^31 = -2147483648
2^32 = 0
2^33 = 0

```

例 4 int 型で 3 のべき乗を求めてゆくと ...

```

/* error_overflow3.c */
#include<stdio.h>
int main(){
    int x = 1;
    for(int i = 1; i <= 24; i++){
        x *= 3;
        printf("3^%2d = %d\n", i, x);
    }
    return 0;
}

```

```

(中略)
3^18 = 387420489
3^19 = 1162261467
3^20 = -808182895
3^21 = 1870418611
3^22 = 1316288537
3^23 = -346101685
3^24 = -1038305055

```

(3) アンダーフロー

小さ過ぎる数は正しく表現できなかつたり 0 になってしまつたりする。

例 5 $(1/10^n) \times 10^n = 1$ のはずだが ...

```
/* error_underflow.c */
#include<stdio.h>
#include<math.h>
int main(){
    for(int n = 300; n <= 326; n++){
        double x = 1.0;
        for(int i = 1; i <= n; i++) x /= 10;
        for(int i = 1; i <= n; i++) x *= 10;
        printf("n = %d のとき %.12lf\n", n, x);
    }
    return 0;
}
```

```
n = 314 のとき 0.9999999999964
n = 315 のとき 0.9999999998482
n = 316 のとき 0.999999983660
n = 317 のとき 0.999999736627
n = 318 のとき 0.999998748496
n = 319 のとき 0.999988867183
n = 320 のとき 0.999988867183
n = 321 のとき 0.998012604599
n = 322 のとき 0.988131291682
n = 323 のとき 0.988131291682
n = 324 のとき 0.000000000000
```

(4) 累積誤差

塵も積もれば山となる。

例 6 $\frac{1}{5}$ を 5×10^n 個足すと整数のはずだが ...

```
/* error_accumulation.c */
#include<stdio.h>
int main(){
    long N = 5;
    double x = 0.2, sum;
    for(int n = 0; n < 10; n++){
        sum = 0.0;
        for(int i = 0; i < N; i++) sum += x;
        printf("n = 5 × 10^%2d のとき 和 = %20.7lf\n", n, sum);
        N *= 10;
    }
    return 0;
}
```

(中略)

```
n = 5 × 10^ 3 のとき 和 =          1000.0000000
n = 5 × 10^ 4 のとき 和 =         10000.0000000
n = 5 × 10^ 5 のとき 和 =        99999.9999991
n = 5 × 10^ 6 のとき 和 =       999999.9999108
n = 5 × 10^ 7 のとき 和 =      9999999.9995118
n = 5 × 10^ 8 のとき 和 =     100000000.9018808
n = 5 × 10^ 9 のとき 和 =    1000000087.8303370
```

(5) 積み残し

$|a| \gg |b|$ のときに $a + b$ を計算すると b の下の方の桁が無視される。

例 7 $r = 0.999$ のとき、等比級数の和 $1 + r + r^2 + \dots$ の理論値は $\frac{1}{1-r} = 1000$ だが ...

```
/* error_geometric_series.c */
#include<stdio.h>
#include<math.h>
int main(){
    double r = 0.999, x;
    printf("昇順に加えると\n");
    for(int j = 4; j <= 7; j++){
        long n = (long)pow(10, j);
        x = 0.0;
        for(long i = 0; i <= n; i++) x += pow(r, i);
        printf("10^%d 項目までの和 = %17.12lf\n", j, x);
    }
    printf("\n");
    printf("降順に加えるとちょっとましで\n");
    for(int j = 4; j <= 7; j++){
        long n = (long)pow(10, j);
        x = 1.0;
        for(long i = n; i >= 0; i--) x += pow(r, i);
        printf("10^%d 項目までの和 = %17.12lf\n", j, x);
    }
    return 0;
}
```

昇順に加えると

```
10^4 項目までの和 = 999.954871827366
10^5 項目までの和 = 999.999999999951
10^6 項目までの和 = 999.999999999951
10^7 項目までの和 = 999.999999999951
```

降順に加えるとちょっとましで

```
10^4 項目までの和 = 999.954871827368
10^5 項目までの和 = 999.999999999999
10^6 項目までの和 = 999.999999999999
10^7 項目までの和 = 999.999999999999
```

(6) 桁落ち

$a \div b$ のときに $a - b$ を計算すると著しく精度が落ちる。

例 8 $\left(\frac{1}{n} - \frac{1}{n+1}\right) \times n \times (n+1) = 1$ のはずが ...

```

/* error_cancellation.c */
#include<stdio.h>
#include<math.h>
int main(){
    double n, x, y, z, w;
    for(int i = 5; i < 16; i++){
        n = pow(10, i);
        x = 1 / n;
        y = 1 / (n + 1);
        z = x - y;
        w = z * (n * (n + 1));
        printf("n = 10^%2d のとき %15.12f\n", i, w);
        n *= 10;
    }
    return 0;
}

```

```

n = 10^ 5 のとき 1.0000000000001
n = 10^ 6 のとき 0.9999999999892
n = 10^ 7 のとき 0.9999999999072
n = 10^ 8 のとき 1.000000010319
n = 10^ 9 のとき 1.000000150211
n = 10^10 のとき 1.000000614600
n = 10^11 のとき 0.999999968276
n = 10^12 のとき 1.000048435881
n = 10^13 のとき 1.000906716545
n = 10^14 のとき 0.993964740578
n = 10^15 のとき 0.986076131526

```

(7) おまけその1：どこがバグかな?

$\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \dots$ の計算をしているはずなのに ...

```
/* zeta1.c */
#include<stdio.h>
int main(){
    int n;
    double x = 0.0;
    for(n = 1; n < 11; n++){
        x += 1 / n;
        if(n == 1) printf("1/1                = %15.12f\n", x);
        if(n == 2) printf("1/1 + 1/2          = %15.12f\n", x);
        if(n >= 3) printf("1/1 + 1/2 + ... + 1/%2d = %15.12f\n", n, x);
    }
    return 0;
}
```

```
1/1                = 1.000000
1/1 + 1/2          = 1.000000
1/1 + 1/2 + 1/3    = 1.000000
1/1 + 1/2 + ... + 1/ 4 = 1.000000
1/1 + 1/2 + ... + 1/ 5 = 1.000000
1/1 + 1/2 + ... + 1/ 6 = 1.000000
1/1 + 1/2 + ... + 1/ 7 = 1.000000
1/1 + 1/2 + ... + 1/ 8 = 1.000000
1/1 + 1/2 + ... + 1/ 9 = 1.000000
1/1 + 1/2 + ... + 1/10 = 1.000000
```

(8) おまけその2：おもしろい配列の仕様

```
/* array.c
おもしろい配列の仕様
理由は a[i] = *(a+i) = *(i+a) = i[a] だからなんだけど、わかるかな?
*/
#include<stdio.h>
main(){
    int  a[] = {1,4,2,8,5,7};
    char b[] = "Kochi";

    for(int i = 0; i < 6; i++) printf("%d[a] = %d, ", i, i[a]);
    printf("\n");
    for(int i = 0; i < 5; i++) printf("%d[b] = %c, ", i, i[b]);
    return 0;
}
```

```
0[a] = 1,  1[a] = 4,  2[a] = 2,  3[a] = 8,  4[a] = 5,  5[a] = 7,
0[b] = K,  1[b] = o,  2[b] = c,  3[b] = h,  4[b] = i,
```