

## Table of Contents

- 1 打ち切り誤差と丸め誤差(Truncation and round off errors)
  - 1.1 整数型と実数型の内部表現
    - 1.1.1 caption:浮動小数点数の内部表現 (IEEE754).
- 2 有効桁数(Significant digits)
- 3 浮動小数点演算による過ち(FloatingPointArithmetic)
  - 3.1 分かっているつもりでも、よくやる間違い.
- 4 機械精度(Machine epsilon)
- 5 衍落ち、情報落ち、積み残し(Cancellation)
  - ◦ 5.0.0.1 衍落ち(Cancellation)
  - 5.0.0.2 情報落ち(Loss of Information)
  - 5.0.0.3 積み残し
- 5.1 pythonにおける精度制御演算
- 6 課題
  - 6.1 有効数字
  - 6.2 2次方程式解の公式の罠
    - 6.2.1 (1)式の導出

# 誤差(Error)

file:/Users/bob/Github/TeamNishitani/jupyter\_num\_calc/error  
[https://github.com/daddygongon/jupyter\\_num\\_calc/tree/master/notebooks\\_p](https://github.com/daddygongon/jupyter_num_calc/tree/master/notebooks_p)  
cc by Shigeto R. Nishitani 2017-20

## 打ち切り誤差と丸め誤差(Truncation and round off errors)

数値計算のねらいは、できるだけ正確・高速に解を得ることである。誤差(精度)と収束性(安定性,速度)が数値計算のキモとなる。前回に説明した収束判定条件による誤差は打ち切り誤差(truncation error)と呼ばれる。ここでは、誤差のもう一つの代表例である、計算機に特有の丸め誤差(roundoff error)について見ておこう。

### 整数型と実数型の内部表現

計算機は一般に無限精度の計算をおこなっているわけではない。CPUで足し算をおこなう以上、一般的な計算においてはCPUが扱う一番効率のいい数の大きさが存在する。これが、32bitのCPUでは1ワード,4byte(4x8bits)である。1ワードで表現できる最大整数は、符号に1bit必要なので、 $2^{(31)-1}$ となる。実数は以下のような仮数部と指数を取る浮動小数点数で表わされる。

caption:浮動小数点数の内部表現 (IEEE754).

記号	$s \times f \times B^{e-E}$
$s$	sign bit(符号ビット:正負の区別を表す)
$f$	fraction portion of the number(仮数部)
$e$	biased exponent(指数部)
$B$	base(基底)で通常は2
$E$	bias(下駄)と呼ばれる
real(単精度)	$s = 1, e = 8, f = 23, E = 127$
double precision(倍精度)	$s = 1, e = 11, f = 52, E = 1023$

$E$ は指数が負となる小数点以下の数を表現するためのもの。演算結果は実際の値から浮動小数点数に変換するための操作「丸め(round-off)」が常に行われる。それに伴って現れる誤差を丸め誤差と呼ぶ。

## 有効桁数(Significant digits)

1ワードの整数の最大値とその2進数表示。

```
2***(4*8-1)-1 # => 2147483647
```

この整数を2進数で表示するように変換するには、bin(n)を用いて、

```
bin(2***(4*8-1)-1) # => 0b11111111111111111111111111111111
```

となり、31個の1が並んでいることが分かる。1ワードの整数の最大桁は、nの長さを戻すコマンドlen(n)を使って、

```
len(str(2***(4*8-1)-1)) # => 10
```

となり、たかだか10桁程度であることが分かる。一方、64bitの場合の整数の最大桁。

```
len(str(2***(8*8-1)-1)) # => 19
```

である。

pythonでは多倍長計算するので、通常のプログラミング言語で起こるintの最大数あたりでの奇妙な振る舞いは示さない。

```
2147483647+100 # => 2147483747
```

単精度の浮動小数点数は、仮数部2進数23bit,2倍長実数で52bitである。この有効桁数は以下の通り。

```
len(str(2***(23))) # => 7  
len(str(2***(52))) # => 16
```

pythonでは普通に整数を扱う場合には、1ワードによる制約はなく、大きな数を扱える。

しかし、numpyなどでarrayとして作る場合には最大数に対する制約が現れる。下の例では、

- tmpは普通の変数なので、最大数に対する制約はない。しかし、
- xやyはarrayとして宣言した時点で、int64やfloat64に型が決まってしまう。

- 足し算するとOverflowErrorを吐きます。
  - pythonは中ではCに翻訳しているので、"C long"型よりも大きすぎてダメと根をあげているのがわかるでしょう。

```
In [1]: import numpy

tmp = numpy.int(2**8*8-1)
print(tmp) #=> 9223372036854775807
print(tmp+100) #=> 9223372036854775907

x = numpy.array([0, 0])
print(x.dtype) #=> int64
y = numpy.append(x, tmp+1)
print(y[2]) #=> 9.22337203685e+18
print(y[2]+10)
print(y.dtype)

x[0] = tmp # OK
print(x[0])
x[1] = tmp+1

9223372036854775807
9223372036854775907
int64
9.22337203685e+18
9.22337203685e+18
float64
9223372036854775807
-----
OverflowError Traceback (most recent call last)
<ipython-input-1-153177f82482> in <module>
    14 x[0] = tmp # OK
    15 print(x[0])
--> 16 x[1] = tmp+1

OverflowError: Python int too large to convert to C long
```

## 浮動小数点演算による過ち (FloatingPointArithmetic)

「丸め」にともなって誤差が生じる。CやFortran等の通常のプログラミング言語では「丸める」仕様なのでプログラマーが気をつけなければならない。以下のようなC programでは、予期したのとは違う結果となる。

```
// プログラムリスト : 実数のケタ落ち
#include <stdio.h>

int main(void){
    float a,b,c;
    double x,y,z;

    a=1.23456789;
    printf(" a= %17.10f\n",a);
    b=100.0;
```

```
c=a+b;
printf("%20.10f %20.10f %20.10f\n",a,b,c);

x=(float)1.23456789;
y=(double)100;
z=x+y;
printf("%20.12e %20.12e %20.12e\n",x,y,z);

x=(double)1.23456789;
y=(double)100;
z=x+y;
printf("%20.12e %20.12e %20.12e\n",x,y,z);

return 0;
}
```

分かっているつもりでも、よくやる間違い。

```
// プログラムリスト : 丸め誤差
#include <stdio.h>

int main(void){
    float x=77777,y=7,y1,z,z1;
    y1=1/y;
    z=x/y;
    z1=x*y1;
    printf("%10.2f %10.2f\n",z,z1);
    if (z!=z1){
        printf("z is not equal to z1.\n");
    }
    printf("Surprising?? \n\n\n\n%10.5f %10.5f\n",z,z1);
    return 0;
}
```

これを避けるには、EPSILONという小さな数字を定義しておいて、値の差の絶対値を求めるfabsを使って

\hspace{100mm}

とすべき。このときは数学関数であるfabsを使っているので、

```
maple
> gcc -lm test.c
```

とmath libraryを明示的に呼ぶのを忘れないように。

```
In [2]: x=77777 # change this digits
y=7
y1=1.0/y
z=x/y
z1=x*y1
printf("%63.20f %30.20f % (z, z1))
if (z != z1):
    print("no, different")
else:
    print("yes, identical")
```

```
In [3]: e = 10**(-4)
x=77777 # change this digits
y=7
y1=1.0/y
z=x/y
z1=x*y1
print("%30.20f %30.20f" % (z, z1))
if ( abs(z-z1) < e ):
    print("yes, identical")
else:
    print("no, different")
```

機械精度(Machine epsilon)

上の例では、浮動小数点数で計算した場合に小さい数の差を区別することができなくなるということを示している。つまり、小さい数を足したときにその計算機がその差を認識できなくなる限界ということ。これは、昔はCPU固有の精度で、今でも機械精度(Machine epsilon)と呼ばれる。今は、言語の実装によって違ってくるが、以下のようにして求めることができる。

```
In [4]: e=1.0  
w=1.0+e  
while(w>1.0):  
    print('%.15-1.10e %.15-1.10e %.15-1.10e' % (e,w,w-1.0))  
    e = e/2.0  
    w = 1.0+e
```

1.0000000000e+00 2.0000000000e+00 1.0000000000e+00  
5.0000000000e-01 1.5000000000e+00 5.0000000000e-01  
2.5000000000e-01 1.2500000000e+00 2.5000000000e-01  
1.2500000000e-01 1.1250000000e+00 1.2500000000e-01  
6.2500000000e-02 1.0625000000e+00 6.2500000000e-02  
3.1250000000e-02 1.0312500000e+00 3.1250000000e-02  
1.5625000000e-02 1.0156250000e+00 1.5625000000e-02  
7.8125000000e-03 1.0078125000e+00 7.8125000000e-03  
3.9062500000e-03 1.0039062500e+00 3.9062500000e-03  
1.9531250000e-03 1.0019531250e+00 1.9531250000e-03  
9.7656250000e-04 1.0009765625e+00 9.7656250000e-04  
4.8828125000e-04 1.0004882812e+00 4.8828125000e-04  
2.4414062500e-04 1.0002441406e+00 2.4414062500e-04  
1.2207031250e-04 1.0001220703e+00 1.2207031250e-04  
6.1035156250e-05 1.0000610352e+00 6.1035156250e-05  
3.0517578125e-05 1.0000305176e+00 3.0517578125e-05  
1.5258789062e-05 1.0000152588e+00 1.5258789062e-05  
7.6293945312e-06 1.0000076294e+00 7.6293945312e-06  
3.8146972656e-06 1.0000038147e+00 3.8146972656e-06  
1.9073486328e-06 1.0000019073e+00 1.9073486328e-06  
9.5367431641e-07 1.0000009537e+00 9.5367431641e-07  
4.7683715820e-07 1.0000004768e+00 4.7683715820e-07  
2.3841857910e-07 1.0000002384e+00 2.3841857910e-07  
1.1920928955e-07 1.0000001192e+00 1.1920928955e-07  
5.9604644775e-08 1.0000000596e+00 5.9604644775e-08  
2.9802322388e-08 1.0000000298e+00 2.9802322388e-08

1.4901161194e-08 1.0000000149e+00 1.4901161194e-08  
 7.4505805969e-09 1.0000000075e+00 7.4505805969e-09  
 3.7252902985e-09 1.0000000037e+00 3.7252902985e-09  
 1.8626451492e-09 1.0000000019e+00 1.8626451492e-09  
 9.3132257462e-10 1.0000000009e+00 9.3132257462e-10  
 4.6566128731e-10 1.0000000005e+00 4.6566128731e-10  
 2.3283064365e-10 1.0000000002e+00 2.3283064365e-10  
 1.1641532183e-10 1.0000000001e+00 1.1641532183e-10  
 5.8207660913e-11 1.0000000001e+00 5.8207660913e-11  
 2.9103830457e-11 1.0000000000e+00 2.9103830457e-11  
 1.4551915228e-11 1.0000000000e+00 1.4551915228e-11  
 7.2759576142e-12 1.0000000000e+00 7.2759576142e-12  
 3.6379788071e-12 1.0000000000e+00 3.6379788071e-12  
 1.8189894035e-12 1.0000000000e+00 1.8189894035e-12  
 9.0949470177e-13 1.0000000000e+00 9.0949470177e-13  
 4.5474735089e-13 1.0000000000e+00 4.5474735089e-13  
 2.2737367544e-13 1.0000000000e+00 2.2737367544e-13  
 1.1368683772e-13 1.0000000000e+00 1.1368683772e-13  
 5.6843418861e-14 1.0000000000e+00 5.6843418861e-14  
 2.8421709430e-14 1.0000000000e+00 2.8421709430e-14  
 1.4210854715e-14 1.0000000000e+00 1.4210854715e-14  
 7.1054273576e-15 1.0000000000e+00 7.1054273576e-15  
 3.5527136788e-15 1.0000000000e+00 3.5527136788e-15  
 1.7763568394e-15 1.0000000000e+00 1.7763568394e-15  
 8.8817841970e-16 1.0000000000e+00 8.8817841970e-16  
 4.4408920985e-16 1.0000000000e+00 4.4408920985e-16  
 2.2204460493e-16 1.0000000000e+00 2.2204460493e-16

```
In [5]: from decimal import *
getcontext(),prec=80
e=Decimal(1.0)
w=Decimal(1.0)+e
while(w>1.0):
    print('-%15.10e %-15.10e' %(e,w))
    e = e/Decimal(2.0)
    w = Decimal(1.0)+e
```

```

1.0000000000e+00 2.0000000000e+00
5.0000000000e-01 1.5000000000e+00
2.5000000000e-01 1.2500000000e+00
1.2500000000e-01 1.1250000000e+00
6.2500000000e-02 1.0625000000e+00
3.1250000000e-02 1.0312500000e+00
1.5625000000e-02 1.0156250000e+00
7.8125000000e-03 1.0078125000e+00
3.9062500000e-03 1.0039062500e+00
1.9531250000e-03 1.0019531250e+00
9.7656250000e-04 1.0009765625e+00
4.8828125000e-04 1.0004882812e+00
2.4414062500e-04 1.0002441406e+00
1.2207031250e-04 1.0001220703e+00
6.1035156250e-05 1.0000610352e+00
3.0517578125e-05 1.0000305176e+00
1.5258789062e-05 1.0000152588e+00
7.6293945312e-06 1.0000076294e+00
3.8146972656e-06 1.0000038147e+00
1.9073486328e-06 1.0000019073e+00
9.5367431641e-07 1.0000009537e+00
4.7683715820e-07 1.0000004768e+00

```

```
3.4544674220e-77 1.0000000000e+00  
1.7272337110e-77 1.0000000000e+00  
8.6361685551e-78 1.0000000000e+00  
4.3180842775e-78 1.0000000000e+00  
2.1590421388e-78 1.0000000000e+00  
1.0795210694e-78 1.0000000000e+00  
5.3976053469e-79 1.0000000000e+00  
2.6988026735e-79 1.0000000000e+00  
1.3494013367e-79 1.0000000000e+00  
6.7470066837e-80 1.0000000000e+00
```

## 桁落ち, 情報落ち, 積み残し(Cancellation)

有効数字がそれぞれ5桁で計算した結果を示せ。

### 桁落ち(Cancellation)

```
maple  
0.723657  
- 0.723649  
-----
```

### 情報落ち(Loss of Information)

```
maple  
72365.7  
- 1.23659  
-----
```

積み残し

```
maple  
72365.7  
- 0.001  
-----
```

## pythonにおける精度制御演算

pythonにはroundという標準的な丸める関数がありますが、使用には注意が必要です。

<https://note.nkmk.me/python-round-decimal-quantize/>

```
In [6]: print('0.4 =>', round(0.4))  
print('0.5 =>', round(0.5))  
print('0.6 =>', round(0.6))
```

```
0.4 => 0  
0.5 => 0  
0.6 => 1
```

```
In [7]: print('0.5 =>', round(0.5))  
print('1.5 =>', round(1.5))  
print('2.5 =>', round(2.5))  
print('3.5 =>', round(3.5))
```

```
0.5 => 0
```

```
1.5 => 2  
2.5 => 2  
3.5 => 4
```

```
In [8]: print('0.05 =>', round(0.05, 1))  
print('0.15 =>', round(0.15, 1))  
print('0.25 =>', round(0.25, 1))  
print('0.35 =>', round(0.35, 1))  
print('0.45 =>', round(0.45, 1))
```

```
0.05 => 0.1  
0.15 => 0.1  
0.25 => 0.2  
0.35 => 0.3  
0.45 => 0.5
```

注釈 浮動小数点数に対する round() の振る舞いは意外なものかもしれません： 例えば、 round(2.675, 2) は予想通りの 2.68 ではなく 2.67 を与えます。これはバグではありません： これはほとんどの小数が浮動小数点数で正確に表せないことの結果です。

### 2. 組み込み関数 round() – Python 3.6.3 ドキュメント

さらに、 pythonにはdecimalという、10進固定及び浮動小数点数演算用のライブラリがあります。この中のquantize関数で直感的な四捨五入が実現できます。こちらの方が良さそう。

```
In [1]: from decimal import *  
  
def pretty_p(result,a,b,operator):  
    print('context.prec:{}\n'.format(getcontext().prec))  
    print(' %20.14f % (a)\n')  
    print(' %1s%20.14f % (operator, b)\n')  
    print('-----')  
    print(' %20.14f % (result)\n')  
  
n = 10  
getcontext().prec = n  
  
a=Decimal('0.723657').quantize(Decimal(10)**-n)  
b=Decimal('0.723649').quantize(Decimal(10)**-n)  
pretty_p(a-b,a,b,'-')  
  
a=Decimal('0.723657').quantize(Decimal(10)**-n)*100000  
b=Decimal('0.123659').quantize(Decimal(10)**-n)*10  
pretty_p(a-b,a,b,'-')  
  
a=Decimal('0.723657').quantize(Decimal(10)**-n)*100000  
b=Decimal('0.1').quantize(Decimal(10)**-n)/100  
pretty_p(a+b,a,b,'+')
```

```
context.prec:10  
0.723657000000000  
- 0.723649000000000  
-----  
0.000008000000000  
context.prec:10  
72365.69999999999709  
- 1.236590000000000  
-----  
72364.4634099999666  
context.prec:10  
72365.69999999999709
```

```

+ 0.00100000000000
-----
72365.70100000000093

In [2]:
n = 5
getcontext().prec = n

a=Decimal('0.723657').quantize(Decimal(10)**-n)
b=Decimal('0.723649').quantize(Decimal(10)**-n)
pretty_p(a-b,a,b,'-')

a=Decimal('0.723657').quantize(Decimal(10)**-n)*100000
b=Decimal('0.123659').quantize(Decimal(10)**-n)*10
pretty_p(a-b,a,b,'-')

a=Decimal('0.723657').quantize(Decimal(10)**-n)*100000
b=Decimal('0.1').quantize(Decimal(10)**-n)/100
pretty_p(a+b,a,b,'+')

```

```

context.prec:5
0.72366000000000
- 0.72365000000000
-----
0.00001000000000
context.prec:5
72366.0000000000000000
- 1.23660000000000
-----
72365.0000000000000000
context.prec:5
72366.0000000000000000
+ 0.00100000000000
-----
72366.0000000000000000

```

## 課題

### 有効数字

- 大きな数とおしのわずかな差は、丸め誤差にとくに影響を受ける。  $23.173 - 23.094$  を有効数字がそれぞれ5桁、4桁、3桁、2桁で計算した結果を示せ。同様に、 $0.81321/(23.173 - 23.094)$  を有効数字がそれぞれ5桁、4桁、3桁、2桁で計算した結果を示せ。
- 10進数10桁および3桁の有効桁数をもった計算機になったつもりで、以下の条件で預金を求める計算をおこなえ。
  - 元本を10000万円とする
  - 利息0.3%とする
  - 複利計算で10年でいくらになるか。

## 2次方程式解の公式の罠

- 係数を  $a = 1, b = 10000000, c = 1$ としたときに、通常の解の公式を使った解と、解と係数の関係（下記の記述を参照）を使った解とを出力するプログラムをpythonで作成すると以下の通りとなる。解の有効数字が2種類の計算方法の違いでどう違うか、いくつかの精度で実行させた結果を使って解説せよ。

2次方程式  $ax^2 + bx + c = 0$  の係数  $a, b, c$  が特殊な値をもつ場合、通常の解の公式

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

にしたがって計算するとケタ落ちによる間違った答えを出す。その特殊な値とは

$$\sqrt{b^2 - 4ac} \approx |b|$$

となる場合である。

ケタ落ちを防ぐには、 $b > 0$  の場合は、

$$x_1 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

として、ケタ落ちを起こさずに求め、この解を使って、解と係数の関係より

$$x_2 = \frac{c}{a x_1} \dots (1)$$

で求める。 $b < 0$  の場合は、解の公式のたし算の方

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

を使って同様に求める。

In [5]:

```

import numpy as np
def solve_by_formula(a,b,c):
    x0 = (-b + np.sqrt(b**2-4*a*c))/(2*a)
    x1 = (-b - np.sqrt(b**2-4*a*c))/(2*a)
    return (x0, x1)

def solve_precise(a,b,c):
    x0 = (-b - np.sqrt(b**2-4*a*c))/(2*a)
    x1 = c/(a*x0)
    return (x0, x1)

```

```

print(solve_by_formula(1,10000000,1))
print(solve_precise(1,10000000,1))

```

```

(-9.96515154838562e-08, -9999999.9999999)
(-9999999.9999999, -1.00000000000001e-07)

```

In [6]:

```

from decimal import *
print("prec=40")
getcontext().prec = 40
print(solve_normal_formula(Decimal('1'),
                           Decimal('10000000'),
                           Decimal('1')))
print(solve_precise_formula(Decimal('1'),
                           Decimal('10000000'),
                           Decimal('1')))
print("\nprec=20")
getcontext().prec = 20
print(solve_normal_formula(Decimal('1'),
                           Decimal('10000000'),
                           Decimal('1')))

```

```

print(solve_precise_formula(Decimal('1'),
    Decimal('10000000'),
    Decimal('1')))

prec=40
(Decimal('-9999999.999998999999999999990000000000000'), Decimal('-1.0000000000000001000000000000000E-7'))
(Decimal('-9999999.999998999999999999990000000000000'), Decimal('-1.0000000000000001000000000000000E-7'))

prec=20
(Decimal('-9999999.999999000000'), Decimal('-1.000000E-7'))
(Decimal('-9999999.999999000000'), Decimal('-1.000000000000100000E-7'))

```

## (1)式の導出

(1)式の導出を示しておく。解と係数の関係を導びくと

$$\begin{aligned}
 (x - x_1)(x - x_2) &= x^2 - (x_1 + x_2)x + x_1 x_2 \\
 &= \frac{a}{a}x^2 + \frac{b}{a}x + \frac{c}{a}
 \end{aligned}$$

となる。(1)式は最後の定数項より、

$$\frac{c}{a} = x_1 x_2$$

から導かれる。

In [ ]: