2022/2/22(Tue)

1 Representation of real numbers on a computer

1.1 Floating-point arithmetic (IEEE754: Double precision)

Computers have a limited amount of memory. Therefore, it is not possible to handle an infinite number of digits, so it is necessary to give up at some point. Furthermore, the computer can only handle binary numbers of **0** and **1** inside, so we need to discuss how to fit them into a finite number of digits. Right now, the CPUs of mainstream computers are based on the **IEEE754** standard. The IEEE754 standard includes a technical standard for **floating-point**.

π is an irrational number, so it cannot be held by a computer. However, in the IEEE754 standard, even 1.1 cannot be represented exactly in decimal. The error is already there before the calculation is done!

Let's take a look at a simple example.

Listing 1 roundoff.c

```c
#include <stdio.h>

int main(){
    double i=1.1;
    printf("%.20f\n",i);
}
```

If you don't have any programming environment (if it's too much trouble), use **CES** (`https://www.ces-alpha.org/en/`) or **Ganjin** (`https://ganjin.online/`).

Enter the following command and try to execute it.

```
$ gcc roundoff.c
$ ./a.out
1.10000000000000008882
```

It is certainly not exactly 1.1, and you can see that there is an error. Let's check IEEE754 to see why such an error occurred.

First of all, IEEE754 has two formats: *single* and *double*. *single* is 32 bits long, and *double* is 64 bits long.

$$(-1)^s \times 2^e \times m$$

Sign bit s : 1 bit (0 or 1)

Exponent e : $e_{\min} \leq e \leq e_{\max}$

Significand m : It is expressed in the normalized form, $d_0, d_1, d_2, \cdots, d_{p-1}$. However, d_i can be 0 or 1.

The values of $e_{\max}$, $e_{\min}$, and p vary depending on *single* and *double*, and are defined as shown in Table 1.

Table 1 Exponent and significand

Parameter	single	double
p	24 bit	53 bit
$e_{\max}$	127	1023

Figure 1 shows the representation for the *Single* case. In the state called normalized significand, d_0 is always set to 1, and the numbers are stored starting from d_1. It is called the “hidden” or “implicit” bit.

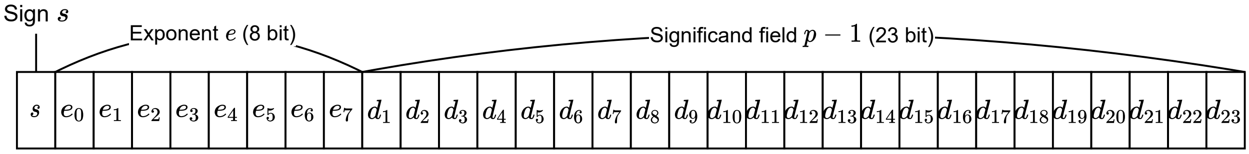

Fig.1 An example of a layout for 32-bit (*Single*) floating-point

If we convert 1.1 to binary, we get a recurring decimal like $(1.000110011001\ldots)_2$. This is the cause of the error.

The precision of the calculation (significant digits) in *double* is $2^{-52} \simeq 2.22 \times 10^{-16} \simeq 10^{-15.65}$, which is approximately 16 digits.

Exercise

Convert the decimal number 7.25 to a floating-point number. However, p (including d_0) must be 6 bits, and e must be 3 bits.

$$\begin{aligned} 7.25 &= (111.01)_2 \\ &= (-1)^0 \times 2^2 \times (1.1101)_2 \end{aligned}$$

Therefore, the answer is

$$0\ 010\ 11010$$

Example of a large error (quadratic equations)

The solution to the quadratic equation $ax^2 + bx + c = 0$ is

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

Transforming this equation, we get

$$\begin{aligned} \frac{-b + \sqrt{b^2 - 4ac}}{2a} &= \frac{b^2 - b^2 + 4ac}{2a(-b - \sqrt{b^2 - 4ac})} \\ &= \frac{4ac}{2a(-b - \sqrt{b^2 - 4ac})} \\ &= \frac{2c}{-b - \sqrt{b^2 - 4ac}} \end{aligned} \tag{1}$$

When calculated with $a = 1,\ b = 10^{15},\ c = 10^{14}$.

Listing 2 quadratic.c

```c
#include <stdio.h>
#include <math.h>

int main(){
    double a=1;
    double b=1000000000000000;
    double c=100000000000000;
```

```
    double x1=(-b+sqrt(b*b-4*a*c))/(2*a);
    printf("%.16f\n",x1);

    double x2=(2*c)/(-b-sqrt(b*b-4*a*c)); // Equation (1)

    printf("%.16f\n",x2);
}
```

```
$ gcc quadratic.c -lm
$ ./a.out
-0.1250000000000000
-0.1000000000000000
```

This result shows that

$$\frac{-b+\sqrt{b^2-4ac}}{2a} = -0.1250000000000000$$
$$\frac{2c}{-b-\sqrt{b^2-4ac}} = -0.1000000000000000$$

but it is strange that the answers are different.

2 Interval arithmetic

2.1 Operation between floating-point numbers

Let $\mathbb{F}$ be the set of floating-point numbers defined by IEEE754. Rounding, such as truncation and rounding up, in IEEE754 is defined only for the results of the five operations $+, -, *, /$ and $\sqrt{\cdot}$. Note that rounding of input, output, sin, cos, etc. is not defined.

Let $\circ \in \{+, -, \times, /\}$. In this case, the following rounding modes exist.

Round to nearest : $\tilde{\circ} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$, and round to the nearest floating-point number.
Round upward : $\bar{\circ} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$, $\inf\{x \in \mathbb{F} \mid x \geq a \circ b\}$
Round downward : $\underline{\circ} : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$, $\sup\{x \in \mathbb{F} \mid x \leq a \circ b\}$

Consider floating-point operations. That is, if $a, b \in \mathbb{F}$, then the four arithmetic operations can be defined as follows.

$$\begin{aligned} a + b &\in [a \underline{+} b, a \overline{+} b] = \{c \in \mathbb{R} \mid a \underline{+} b \leq c \leq a \overline{+} b\} \\ a - b &\in [a \underline{-} b, a \overline{-} b] \\ a \cdot b &\in [a \underline{\cdot} b, a \bar{\cdot} b] \\ a / b &\in [a \underline{/} b, a \overline{/} b] \end{aligned}$$

In C compilers (such as gcc), `fesetround` function can be used to change the rounding mode by using `fenv.h`.

```
//You can change the rounding mode like this!

#include <fenv.h>

fesetround(FE_UPWARD); //Change to upward rounding mode
```

```
fesetround(FE_DOWNWARD); //Change to downward rounding mode
fesetround(FE_TONEAREST); //Change to nearest rounding mode
```

I mentioned earlier that it is difficult to accurately represent 1.1 on a computer, but it is now possible to calculate the correct interval.

What is the correct interval? For example, $\pi \simeq 3.14$ is just an approximation, but if we say $\pi \in [3.14, 3.15]$, this is the correct interval (correct information). Let's check this in the actual code.

```
#include <stdio.h>
#include <math.h>
#include <fenv.h>

int main(){
    fesetround(FE_UPWARD);
    double a_u=11.0/10.0;

    fesetround(FE_DOWNWARD);
    double a_l=11.0/10.0;

    /*
    Do not use 1.1
    Recall that the rounding mode can be applied only to certain operations.
    */

}
```

In this case, we should have $11/10 \in$ [`a_l`, `a_u`]. (However, we cannot control the rounding mode inside the `printf` function.)

So, if we want to use interval arithmetic to calculate 0.1+1.1, how do we do that? First of all, as in the case of 11/10, we can take the interval $1/10 \in$ [`b_l`, `b_u`]. Therefore, by calculating as in Figure 2, we can get the correct interval, $a + b \in$ [`c_l`, `c_u`], can't we?

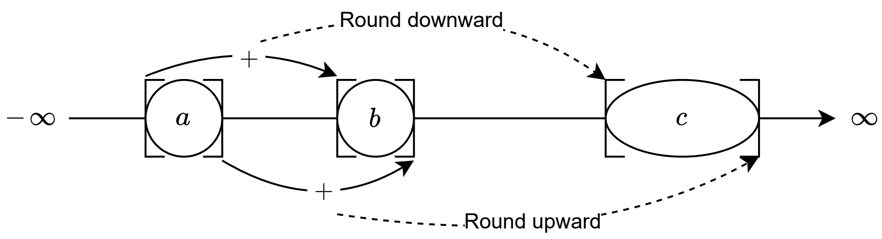

Fig.2 [`a_l`, `a_u`] + [`b_l`, `b_u`]

Thus, the four arithmetic operations on the interval can be summarized as follows. $(a_l, a_u, b_l, b_u \in \mathbb{F})$

Additon : $\{a + b \mid a \in [a_l, a_u],\ b \in [b_l, b_u]\} = [a_l,\ a_u] + [b_l,\ b_u] = [a_l \underline{+} b_l,\ a_u \overline{+} b_u]$

Subtraction : $[a_l,\ a_u] - [b_l,\ b_u] = [a_l \underline{-} b_u,\ a_u \overline{-} b_l]$

Multiplication : $[a_l,\ a_u] \times [b_l,\ b_u] = [\min\{a_l \underline{\cdot} b_l,\ a_u \underline{\cdot} b_l,\ a_l \underline{\cdot} b_u,\ a_u \underline{\cdot} b_u\}, \max\{a_l \overline{\cdot} b_l,\ a_u \overline{\cdot} b_l,\ a_l \overline{\cdot} b_u,\ a_u \overline{\cdot} b_u\}]$

Division : $[a_l,\ a_u]/[b_l,\ b_u] = [\min\{a_l \underline{/} b_l,\ a_u \underline{/} b_l,\ a_l \underline{/} b_u,\ a_u \underline{/} b_u\}, \max\{a_l \overline{/} b_l,\ a_u \overline{/} b_l,\ a_l \overline{/} b_u,\ a_u \overline{/} b_u\}]$ (but $0 \notin [b_l, b_u]$)

Exercise

Modify the code in `addition.c` (Listing 3) to calculate $0.1 + 1.1$ using interval arithmetic.

Listing 3 addition.c

```c
#include <stdio.h>
#include <math.h>
#include <fenv.h>

int main(){
    double a=1.0/10.0; //0.1
    double b=11.0/10.0; //1.1

    double c=a+b;

    printf("%lf\n",c);
}
```

```
(How to compile)
$ gcc addition.c -lm

(execution)
$ ./a.out
```

Interval of solutions to a quadratic equation

This is just an example. When $a = 1,\ b = 10^{15},\ c = 15^{14}$, the result is as follows.

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a} \in [-0.125000, -0.062500]$$

The actual answer is $x = -0.1$, so we can say that we have calculated the correct interval.

Listing 4 int_quadratic.c

```c
#include <stdio.h>
#include <math.h>
#include <fenv.h>

int main(){
    double a=1;
    double b=1000000000000000;
    double c=100000000000000;

    fesetround(FE_TONEAREST);
    double int_sqrt=b*b-4*a*c;

    fesetround(FE_UPWARD);
    double sqrt_u=sqrt(int_sqrt);
    fesetround(FE_DOWNWARD);
    double sqrt_l=sqrt(int_sqrt);

    fesetround(FE_UPWARD);
    double denom_u= -b + sqrt_u;
    fesetround(FE_DOWNWARD);
```

```c
    double denom_l= -b + sqrt_l;

    fesetround(FE_TONEAREST);
    double numerator=2*a;
    fesetround(FE_UPWARD);
    double ans_u[4];
    ans_u[0]=denom_l/numerator;
    ans_u[1]=denom_u/numerator;
    fesetround(FE_DOWNWARD);
    double ans_l[4];
    ans_l[0]=denom_l/numerator;
    ans_l[1]=denom_u/numerator;

    double min,max;

    max = ans_u[0];
    for(int i=0; i<2; i++){
        if(max < ans_u[i]){
            max=ans_u[i];
        }
    }

    min = ans_l[0];
    for(int i=0; i<2; i++){
        if(min > ans_l[i]){
            min=ans_l[i];
        }
    }

    printf("ans in [%lf, %lf]\n",min,max);

}
```

Similarly, we can find the interval of the solution for $\dfrac{2c}{-b-\sqrt{b^2-4ac}}$.

2.2 About matrix operations

Next, we will discuss matrix calculations.

The n-dimensional vectors $a \in \mathbb{F}^n$ and $b \in \mathbb{F}^n$ of floating-point numbers can be computed as follows

$$a \cdot b \subset [a \underline{\cdot} b,\ a \bar{\cdot} b]$$

Note that $a \underline{\cdot} b$ means that all inner product operations are computed with downward rounding, and $a \bar{\cdot} b$ means that all inner product operations are computed with upward rounding.