Chapter 0. Finite Accuracy Calculation

将数学公式搬运到电脑的过程中，经常会发现这些程序有非预期的表现。简单如在Python执行以下命令：

```
>>> 1+1.01-1.01
0.9999999999999998
>>> 1-1.01+1.01
1.0
```

或是使用求根公式$\frac{-b\pm\sqrt{b^2-4ac}}{2a}$对$x^2-24691356x-1=0$进行求根：

```
>>> x
-4.097819328308106e-08（错误的）
>>> x_real
-4.050000332100021e-08（正确的）
```

如此简单操作竟有意想不到的情况。为什么会出现这些错误？要如何避免它们？这些都是数值计算解决的问题。这一问题不仅发生在个人开发者中，即使最知名的企业也常被困扰：早期奔腾系列FDIV（浮点除）指令集实现具有数值问题，直接导致当年Intel损失4.75亿美元[1]。也许数值计算不能解决一切出现在计算机数值实践中的问题，但毫无疑问：不经过严格数值训练人员编写的计算程序并不可靠。

实数的有限精度表示

计算机系统本质上只能表示整数。对于实数及其代数运算，计算机提供的是一种称为浮点数的替代集合[2]。但这种实数集与浮点数集之间并非一一映射，导致很多实数性质在浮点数集合中并不成立。这种使用浮点数集合以及相关运算近似实数集和其代数的方法称为计算机的有限精度表示。

1 浮点模型

浮点（英语：floating point）是一种对于实数的近似值数值表现法，由一个有效数字（即尾数）加上幂指数来表示，通常是乘以某个基数的整数次指数得到。以这种表示法表示的数值，称为**浮点数**（floating-point number）。利用浮点进行运算，称为**浮点计算**，但这种有限精度运算常伴随着近似或舍入误差。

计算机使用浮点数运算的原因在于硬件本质上只处理整数。例如：$4\div2=2$，$4=100_{(2)}$、$2=010_{(2)}$，在二进制相当于退一位数。则$1.0\div2=0.5=0.1_{(2)}$也就是$\frac{1}{2}$。依此类推，二进制的$0.01_{(2)}$就是十进制$\frac{1}{2^2}=\frac{1}{4}=0.25$。由于十进位制无法准确换算成二进位制的部分小数，如0.1，因此只能使用有限和的近似表达。

浮点表示方法类似于基数为10的科学记数法，但在计算机上，使用2为基数的幂级数来表示。一个浮点数a由两个数m和e来表示：$a=m\times be$。在任意一个这样的系统中，我们选择一个基数b（记数系统的基）和精度p（即使用多少位来存储）。尾数m是形如$\pm d**.***ddd\ldots ddd$的p位数（每一位是一个介于0到b-1之间的整数，包括0和b-1）。如果m的第一位是非0整数，m称作**正规化**的浮点数。有一些规范使用一个单独的符号位（s 代表+或者-）来表示正负，这样m必须是正的。e是指数。

这些表示法的设计，来自于对于值的表现范围，与精度之间的取舍：可以在某个固定长度的存储空间内表示出某个实数的近似值。例如，一个指数范围为±4的4位十进制浮点数可以用来表示43210，4.321或0.0004321，但是没有足够的精度来表示432.123和43212.3（必须近似为432.1和43210）。

2 IEEE 754标准

如前所述，浮点模型本质就是一种使用幂级数有限和近似实数的方法。但模型中对于尾数、指数、符号位的分配并无要求。每一种取舍方式都生成了新的浮点模型。为了保证实数运算在不同的硬件平台中有一致行为，电气电子工程师协会（IEEE）规定了标准化的浮点数模型IEEE 754[3]。这一模型已被当前绝大多数硬件使用，因此后文都基于该模型进行讨论，并称为标准浮点数模型。

在标准浮点数模型中，首位是符号位S，然后是指数位E，最后是尾数位M，它简单的表示为：

$$S \times M \times 2^{E-e} \tag{1}$$

其中的e是固定偏置。对于单精度浮点数（float）和双精度浮点数（double），其取值范围是：

	S	E	F	Value
float	any	$[1, 254]$	any	$(-1)^S \times 2^{E-127} \times 1.F$
	any	0	nonzero	$(-1)^S \times 2^{-126} \times 0.F^*$
	0	0	0	$+0.0$
	1	0	0	-0.0
	0	255	0	$+\infty$
	1	255	0	$-\infty$
	any	255	nonzero	NaN
double	any	$[1, 2046]$	any	$(-1)^S \times 2^{E-1023} \times 1.F$
	0	0	0	$(-1)^S \times 2^{-1022} \times 0.F^*$
	1	0	0	$+0.0$
	0	2047	0	-0.0
	1	2047	0	$+\infty$
	any	2047	nonzero	NaN

这种规定的结果使得浮点数在二进制与十进制下的有效数字数如下[4]：

Type	Significant digits	Number of bytes
float	6 - 7	4
double	15 - 16	8

Type	Exponent length	Mantissa length
float	8 bits	23 bits
double	11 bits	52 bits

这也是常说的单精度浮点表示范围$[1.175494351 \times 10^{-38}, 3.402823466 \times 10^{38}]$，有效数字7位；双精度浮点表示范围$[2.2250738585072014 \times 10^{-308}, 1.7976931348623158 \times 10^{308}]$，有效数字15位由来。

举一些简单例子帮助理解标准浮点的记法：

$$1.0 = 0011111111110000(+48\ more\ zeros) = +1 \times 2^{1023-1023} \times (1.0)_2 \tag{2}$$

这种记录方法可以通过以下代码验证：

```
union Udoub
{
    double d;
    unsigned char c[8];
};
void main()
{
    Udoub u;
    u.d = 6.5;
    for (int i = 7; i >= 0; i--)
        printf("%02x", u.c[i]);
    printf("\n");
}
```

它会打出401a000000000000，如果写成二进制形式则是0100 0000 0001 1010，进一步改写：

$$0100000000011010(+48\ more\ zeros) = +1 \times 2^{1025-1023} \times (1.1010)_2 = 6.5$$

得到最初的实数6.5。

理解浮点数系统运作原理后，不难想到：利用有限项级数和近似实数，会把一个区间内所有实数映射到浮点数集的一个点上。这种区间称为浮点数分辨力。由于浮点数集在数轴上并非均匀分布，故取其最小值做为近似误差界估计ϵ，该值在C++中可通过numeric _limits::epsilon()可以获得，小于该数字的两个实数在计算机中不可分辨。利用ϵ，可以得出对两浮点数做N次运算带来的误差界限：$\sqrt{N}\epsilon$。

3 一些例子

3.1 一元二次方程求根

考虑求解一元二次方程：

$$x^2 - 2 * 12345678x - 1 = 0$$

若使用一般求根公式：

$$x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

较小解为：

$$x_{min} = -4.097819328308106 \times 10^{-8}$$

但实际上，真正解为：

$$x_{min} = -4.050000332100021 \times 10^{-8}$$

这是因为分子中$4ac$和b作运算舍入误差引起的。故需要使用以下的求根公式：

$$\begin{aligned} q &= -\frac{1}{2}(b + sign(b)\sqrt{b^2 - 4ac}) \\ x_1 &= \frac{q}{a} \\ x_2 &= \frac{c}{q} \end{aligned} \tag{3}$$

该公式给出解为：

$$x_{min} = -4.050000332100025 \times 10^{-8}$$

3.2 浮点数减法

在本章最开头已经给出了浮点数减法的例子：

```
>>> 1+1.01-1.01
0.9999999999999998
>>> 1-1.01+1.01
1.0
```

这这浮点数相减带来的精度问题有专门的名称：**灾难性抵消**（英语：catastrophic cancellation）[5]。发生灾难性抵消是因为减法运算对邻近数值的输入是病态的：即使近似值$\tilde{x} = x(1+\delta_x)$和$\tilde{y} = y(1+\delta_y)$与真实值$x$和$y$相比，相对误差$|\delta_x| = |x-\tilde{x}|/|x|$和$|\delta_y| = |y-\tilde{y}|/|y|$不大，近似值差的$\tilde{x}-\tilde{y}$与真实值差相对误差也会与真实值差$x-y$成反比：

$$\begin{aligned}\tilde{x}-\tilde{y} &= x(1+\delta_x) - y(1+\delta_y) \\ &= x - y + (x-y)\frac{x\delta_x - y\delta_y}{x-y} \\ &= (x-y)(1+\frac{x\delta_x - y\delta_y}{x-y})\end{aligned}$$

因此，浮点数近似运算$\tilde{x}-\tilde{y}$与实数运算$x-y$的相对误差为：

$$\frac{x\delta_x - y\delta_y}{x-y}$$

当输入接近时，该误差会极大增加。

3.3 有限差分近似

考虑如下导数的近似有限差分近似：

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

在数学上，该式随h缩小而收敛。但程序中，$x+h$和x本身都是浮点数会有舍入误差，并且差分近似微分也有截断误差。假设$x = 10.3, h = 0.0001$，则$f(x+h)$和$f(x)$天生带上了ϵ的舍入误差，又假设这一切都是在单精度下进行的，有$\epsilon \approx 10^-7$，整个系统的舍入误差约在$\epsilon x/h \approx 0.01$，是一个相当糟糕的精度。如果$f(x)$的值能准确获取，那么误差界限会限制到$\epsilon|f(x)/h|$水平。

4 参考文献

1. 1994 - Annual Report. Intel. June 20, 2020 [June 20, 2020]. ↩

2. Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations*. JHU press. ↩

3. https://zh.wikipedia.org/wiki/IEEE_754 ↩

4. Type float | Microsoft Learn ↩

5. 灾难性抵消 - 维基百科，自由的百科全书 (wikipedia.org) ↩