What Every Programmer Should Know About Floating-Point Arithmetic

Last updated: October 15, 2015

Contents

Abstract

The article provides simple answers to the common recurring questions of novice programmers about floating-point numbers not 'adding up' correctly, and more in-depth information about how IEEE 754 floats work.

1 Why don't my numbers add up?

So you've written some absurdly simple code, say for example:

```
0.1 + 0.2
```

and got a really unexpected result:

```
0.30000000000000004
```

Well, this document is here to:

- Explain concisely why you get that unexpected result
- Tell you how to deal with this problem
- If you're interested, provide in-depth explanations of why floating-point numbers have to work like that and what other problems can arise

You should look at the Sec. 2 first - but don't stop there!

2 Basic Answers

2.1 Why don't my numbers, like 0.1 + 0.2 add up to 0.3?

. . . and instead I get a weird result like 0.30000000000000004?

Because internally, computers use a binary floating point format that cannot accurately represent a number like 0.1, 0.2 or 0.3 *at all*.

When the code is compiled or interpreted, your "0.1" is already rounded to the nearest number in that format, which results in a small rounding error even before the calculation happens.

2.2 Why do computers use such a stupid system?

It's not stupid, just different. Decimal numbers cannot accurately represent a number like 1/3, so you have to round to something like 0.33 - and you don't expect 0.33 + 0.33 + 0.33 to add up to 1, either - do you?

Computers use binary numbers because they're faster at dealing with those, and because a tiny error in the 17th decimal place sometimes doesn't matter at all since the numbers you work with aren't round (or that precise) anyway.

2.3 What can I do to avoid this problem?

That depends on what kind of calculations you're doing.

- If you really need your results to add up exactly, especially when you work with money: use a special decimal datatype. An alternative is to work with integers, e.g. do money calculations entirely in cents.
- If you just don't want to see all those extra decimal places: simply format your result rounded to a fixed number of decimal places when displaying it.

2.4 Why do other calculations like 0.1 + 0.4 work correctly?

In that case, the result (0.5) *can* be represented exactly as a floating-point number, and it's possible for rounding errors in the input numbers to cancel each other out - But that can't necessarily be relied upon (e.g. when those two numbers were stored in differently sized floating point representations first, the rounding errors might not offset each other).

In other cases like 0.1 + 0.3, the result actually isn't *really* 0.4, but close enough that 0.4 is the shortest number that is closer to the result than to any other floating-point number. Many languages then display that number instead of converting the actual result back to the closest decimal fraction.

3 Number formats

3.1 Binary Fractions

As a programmer, you should be familiar with the concept of binary integers, i.e. the representation of integer numbers as a series of bits:

$$123_{10} = 1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0,$$

$$456_8 = 4 \cdot 8^2 + 5 \cdot 8^1 + 6 \cdot 8^0 = 302_{10}$$

$$\begin{aligned} 1001001_2 &= 1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\ &= 64_{10} + 8_{10} + 1 = 73_{10} \end{aligned}$$

This is how computers store integer numbers internally. And for fractional numbers, they do the same thing:

$$0.123_{10} = 1 \cdot 10^{-1} + 2 \cdot 10^{-2} + 3 \cdot 10^{-3}$$

$$\begin{aligned} 0.100101 &= 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 0 \cdot 2^{-3} + 1 \cdot 2^{-4} + 0 \cdot 2^{-5} + 1 \cdot 2^{-6} \\ &= \frac{1}{2} + \frac{1}{16} + \frac{1}{64} = \frac{37}{64} = .578125_{10} \end{aligned}$$

While they work the same in principle, binary fractions are different from decimal fractions in what numbers they can accurately represent with a given number of digits, and thus also in what numbers result in rounding errors:

Specifically, binary can only represent those numbers as a finite fraction where the denominator is a power of 2. Unfortunately, this does not include most of the numbers that can be represented as finite fraction in base 10, like 0.1.

Fraction	Base	Positional Notation	Rounded to 4 digits
1/10	10	0.1	0.1
1/3	10	0.3333...	0.3333
1/2	2	0.1	0.1
1/10	2	0.00011...	0.0001

And this is how you already get a rounding error when you just *write down* a number like 0.1 and run it through your interpreter or compiler. It's not as big as 3/80 and may be invisible because computers cut off after 23 or 52 binary digits rather than 4. But the error is there and *will* cause problems eventually if you just ignore it.

3.2 Why use Binary?

At the lowest level, computers are based on billions of electrical elements that have only two states, (usually low and high voltage). By interpreting these as 0 and 1, it's very easy to build circuits for storing binary numbers and doing calculations with them.

While it's possible to simulate the behavior of decimal numbers with binary circuits as well, it's less efficient. If computers used decimal numbers internally, they'd have less memory and be slower at the same level of technology.

Since the difference in behavior between binary and decimal numbers is not important for most applications, the logical choice is to build computers based on binary numbers and live with the fact that some extra care and effort are necessary for applications that require decimal-like behavior.

3.3 Floating Point Numbers

3.4 Why floating-point numbers are needed

Since computer memory is limited, you cannot store numbers with infinite precision, no matter whether you use binary fractions or decimal ones: at some point you have to cut off. But how much accuracy is needed? And *where* is it needed? How many integer digits and how many fraction digits?

- To an engineer building a highway, it does not matter whether it's 10 meters or 10.0001 meters wide - his measurements are probably not that accurate in the first place.
- To someone designing a microchip, 0.0001 meters (a tenth of a millimeter) is a *huge* difference - But he'll never have to deal with a distance larger than 0.1 meters.
- A physicist needs to use the speed of light (about 300000000 in SI units) and Newton's gravitational constant (about 0.0000000000667 in SI units) together in the same calculation.

To satisfy the engineer and the chip designer, a number format has to provide accuracy for numbers at very different magnitudes. However, only *relative* accuracy is needed. To satisfy the physicist, it must be possible to do calculations that involve numbers with different magnitudes.

Basically, having a fixed number of integer and fractional digits is not useful - and the solution is a format with a *floating point*.

3.5 How floating-point numbers work

The idea is to compose a number of two main parts:

- A **significand** that contains the number's digits. Negative significands represent negative numbers.
- An **exponent** that says where the decimal (or binary) point is placed relative to the beginning of the significand. Negative exponents represent numbers that are very small (i.e. close to zero).

Such a format satisfies all the requirements:

- It can represent numbers at wildly different magnitudes (limited by the length of the exponent)
- It provides the same relative accuracy at all magnitudes (limited by the length of the significand)
- allows calculations across magnitudes: multiplying a very large and a very small number preserves the accuracy of both in the result.

Decimal floating-point numbers usually take the form of scientific notation with an explicit point always between the 1st and 2nd digits. The exponent is either written explicitly including the base, or an **e** is used to separate it from the significand.

Significand	Exponent	Scientific notation	Fixed-point value
1.5	4	$1.5\ 10^4$	15000
-2.001	2	$-2.001\ 10^2$	-200.1
5	-3	$5\ 10^{-3}$	0.005
6.667	-11	6.667e-11	0.00000000006667

3.6 The standard

Nearly all hardware and programming languages use floating-point numbers in the same binary formats, which are defined in the IEEE 754 standard. The usual formats are 32 or 64 bits in total length:

	Single precision	Double precision
Total bits	32	64
Sign bits	1	1
Significand bits	23	52
Exponent bits	8	11
Smallest number	$2^{-126} \approx 1.2 \times 10^{-38}$	$2^{-1022} \approx 2.2 \times 10^{-308}$
Largest number	ca. $2 \times 2^{127} \approx 3.4 \times 10^{38}$	ca. $2 \times 2^{1023} \approx 1.8 \times 10^{308}$

Note that there are some peculiarities:

- The **actual bit sequence** is the sign bit first, followed by the exponent and finally the significand bits.
- The exponent does not have a sign; instead an **exponent bias** is subtracted from it (127 for single and 1023 for double precision). This, and the bit sequence, allows floating-point numbers to be compared and sorted correctly even when interpreting them as integers.
- The significand's most significant bit is assumed to be 1 and omitted, except for special cases.
- There are separate **positive and a negative zero** values, differing in the sign bit, where all other bits are 0. These must be considered equal even though their bit patterns are different.
- There are special **positive and negative infinity** values, where the exponent is all 1-bits and the significand is all 0-bits. These are the results of calculations where the positive range of the exponent is exceeded, or division of a regular number by zero.
- There are special **not a number** (or NaN) values where the exponent is all 1-bits and the significand is *not* all 0-bits. These represent the result of various undefined calculations (like multiplying 0 and infinity, any calculation involving a NaN value, or application-specific cases). Even bit-identical NaN values must *not* be considered equal.

4 Errors

4.1 Rounding Errors

Because floating-point numbers have a limited number of digits, they cannot represent all real numbers accurately: when there are more digits than the format allows, the leftover ones are omitted - the number is *rounded*. There are three reasons why this can be necessary:

Large Denominators In any base, the larger the denominator of an (irreducible) fraction, the more digits it needs in positional notation. A sufficiently large denominator will require rounding, no matter what the base or number of available digits is. For example, 1/1000 cannot be accurately represented in less than 3 decimal digits, nor can any multiple of it (that does not allow simplifying the fraction).

Periodical digits Any (irreducible) fraction where the denominator has a prime factor that does not occur in the base requires an infinite number of digits that repeat periodically after a certain point. For example, in decimal 1/4, 3/5 and 8/20 are finite, because 2 and 5 are the prime factors of 10. But 1/3 is not finite, nor is 2/3 or 1/7 or 5/6, because 3 and 7 are not factors of 10. Fractions with a prime factor of 5 in the denominator can be finite in base 10, but not in base 2 - the biggest source of confusion for most novice users of floating-point numbers.

Non-rational numbers Non-rational numbers cannot be represented as a regular fraction at all, and in positional notation (no matter what base) they require an infinite number of non-recurring digits.

4.2 Rounding modes

There are different methods to do rounding, and this can be very important in programming, because rounding can cause different problems in various contexts that can be addressed by using a better rounding mode. The most common rounding modes are:

Rounding towards zero - simply truncate the extra digits. The simplest method, but it introduces larger errors than necessary as well as a bias towards zero when dealing with mainly positive or mainly negative numbers.

Rounding half away from zero - if the truncated fraction is greater than or equal to half the base, increase the last remaining digit. This is the method generally taught in school and used by most people. It minimizes errors, but also introduces a bias (away from zero).

Rounding half to even also known as **banker's rounding** - if the truncated fraction is greater than half the base, increase the last remaining digit. If it is equal to half the base, increase the digit only if that produces an even result. This minimizes errors and bias, and is therefore preferred for bookkeeping.

Examples in base 10:

	Towards zero	Half away from zero	Half to even
1.4	1	1	1
1.5	1	2	2
-1.6	-1	-2	-2
2.6	2	3	3
2.5	2	3	2
-2.4	-2	-2	-2

4.3 Comparing floating-point numbers

Due to rounding errors, most floating-point numbers end up being slightly imprecise. As long as this imprecision stays small, it can usually be ignored. However, it also means that numbers expected to be equal (e.g. when calculating the same result through different correct methods) often differ slightly, and a simple equality test fails. For example:

```
1  float a = 0.15 + 0.15
2  float b = 0.1 + 0.2
3  if(a == b) // can be false!
4  if(a >= b) // can also be false!
```

4.4 Don't use absolute error margins

The solution is to check not whether the numbers are exactly the same, but whether their difference is very small. The error margin that the difference is compared to is often called *epsilon*. The most simple form:

```
1  if( fabs(a-b) < 0.00001) // wrong - don't do this
```

This is a bad way to do it because a fixed epsilon chosen because it "looks small" could actually be way too large when the numbers being compared are very small as well. The comparison would return "true" for numbers that are quite different. And when the numbers are very large, the epsilon could end up being smaller than the smallest rounding error, so that the comparison always returns "false". Therefore, it is necessary to see whether the *relative error* is smaller than epsilon:

```
1  if( fabs((a-b)/b) < 0.00001 ) // still not right!
```

4.5 Look out for edge cases

There are some important special cases where this will fail:

- When both `a` and `b` are zero. `0.0/0.0` is "not a number", which causes an exception on some platforms, or returns false for all comparisons.
- When only `b` is zero, the division yields "infinity", which may also cause an exception, or is greater than epsilon even when `a` is smaller.
- It returns `false` when both `a` and `b` are very small but on opposite sides of zero, even when they're the smallest possible non-zero numbers.

Also, the result is not commutative (`nearlyEquals(a,b)` is not always the same as `nearlyEquals(b,a)`). To fix these problems, the code has to get a lot more complex, so we really need to put it into a function of its own:

```
1  #include <float.h>  // compiler-dependent,
2                      // may be in a non-standard location
3
4  int nearlyEqual (float a, float b, float eps)
5  {
6    float absA = fabs(a);
7    float absB = fabs(b);
8    float diff = fabs(a - b);
9
10   if (a == b)
11   {
12     // shortcut, handles infinities
13     return true;
14   }
15   else if (a == 0 || b == 0 || diff < FLT_MIN)
```

```
16    {
17      // a or b is zero or both are extremely close
18      // to it; relative error is less meaningful here
19      return (diff < (eps * FLT_MIN));
20    }
21    else
22    {
23      // use relative error
24      return (diff / (absA + absB) < eps);
25    }
26  }
```

This method passes tests for many important special cases, but as you can see, it uses some quite non-obvious logic. In particular, it has to use a completely different definition of error margin when `a` or `b` is zero, because the classical definition of relative error becomes meaningless in those cases.

There are some cases where the method above still produces unexpected results (in particular, it's much stricter when one value is nearly zero than when it is exactly zero), and some of the tests it was developed to pass probably specify behaviour that is not appropriate for some applications. Before using it, make sure it's appropriate for your application!

4.6 Error Propagation

While the errors in single floating-point numbers are very small, even simple calculations on them can contain pitfalls that increase the error in the result way beyond just having the individual errors "add up".

In general:

- Multiplication and division are "safe" operations
- Addition and subtraction are dangerous, because when numbers of different magnitudes are involved, digits of the smaller-magnitude number are lost.
- This loss of digits can be inevitable and benign (when the lost digits also insignificant for the final result) or catastrophic (when the loss is magnified and distorts the result strongly).
- The more calculations are done (especially when they form an iterative algorithm) the more important it is to consider this kind of problem.

- A method of calculation can be *stable* (meaning that it tends to reduce rounding errors) or *unstable* (meaning that rounding errors are magnified). Very often, there are both stable and unstable solutions for a problem.

There is an entire sub-field of numerical analysis devoted to studying the numerical stability of algorithms. For doing complex calculations involving floating-point numbers, it is absolutely necessary to have some understanding of this discipline.

5 Apendix

```
  /*
   * Reverse-engineering computer representation
   * of floating point numbers
   */

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

void printbits_float (float v);

int main ()
{
    int i;
    float z;
    float o;

    /*
     * two zeroes
     */
    z = (float) 0.;

    printbits_float (z);
    printf ("% 25.16f\n", z);
    printbits_float (-z);
    printf ("% 25.16f\n", -z);
    printf ("\n");

    /*
```

```
29      * two infinities
30      */
31     z = (float) 1. / 0.;
32
33     printbits_float (z);
34     printf ("% 25.16f\n", z);
35     printbits_float (-z);
36     printf ("% 25.16f\n", -z);
37     printf ("\n");
38
39     /*
40      * 1/infinity -> zero
41      */
42     z = (float) 1. / z;
43
44     printbits_float (z);
45     printf ("% 25.16f\n", z);
46     printbits_float (-z);
47     printf ("% 25.16f\n", -z);
48     printf ("\n");
49
50     /*
51      * NaN
52      */
53     z = (float) 0. / 0.;
54
55     printbits_float (z);
56     printf ("% 25.16f\n", z);
57     printbits_float (-z);
58     printf ("% 25.16f\n", -z);
59     printf ("\n");
60
61     /*
62      * 'Regular' binary numbers
63      */
64     o = 1. / 8;
65     for (i = 0; i < 10; i++)
66     {
67         printbits_float (o);
```

```
68          printf ("% 25.16f\n", o);
69          printbits_float (-o);
70          printf ("% 25.16f\n", -o);
71          printf ("\n");
72          o *= 2;
73      }
74
75      o = 1. / 8;
76      for (i = 0; i < 10; i++)
77      {
78          printbits_float (1.f + o);
79          printf ("% 25.16f\n", 1 + o);
80          printbits_float (1.f - o);
81          printf ("% 25.16f\n", 1 - o);
82          printf ("\n");
83          o *= 2;
84      }
85
86      return 0;
87  }
```

```
1   /*
2    * print float in a binary form
3    */
4   #include <stdio.h>
5
6   void printbits_float (float v);
7
8   void printbits_float (float v)
9   {
10      int i;
11      int *j = (int *) &v;
12      int n = 8 * sizeof (v);
13
14      for (i = n - 1; i >= 0; i--)
15      {
16          if ((i == 22) || (i == 30))
17              putchar (' ');
18          putchar ('0' + (((*j) >> i) & 1));
```

```
19          }
20  }
```