Exploring Binary

Why 0.1 Does Not Exist In Floating-Point

By Rick Regan September 6th, 2012

Many new programmers become aware of binary floating-point after seeing their programs give odd results: "Why does my program print 0.10000000000000001 when I enter 0.1?"; "Why does 0.3 + 0.6 = 0.89999999999999991?"; "Why does 6 * 0.1 not equal 0.6?" Questions like these are asked every day, on online forums like stackoverflow.com.

The answer is that most decimals have infinite representations in binary. Take 0.1 for example. It's one of the simplest decimals you can think of, and yet it looks so complicated in binary:

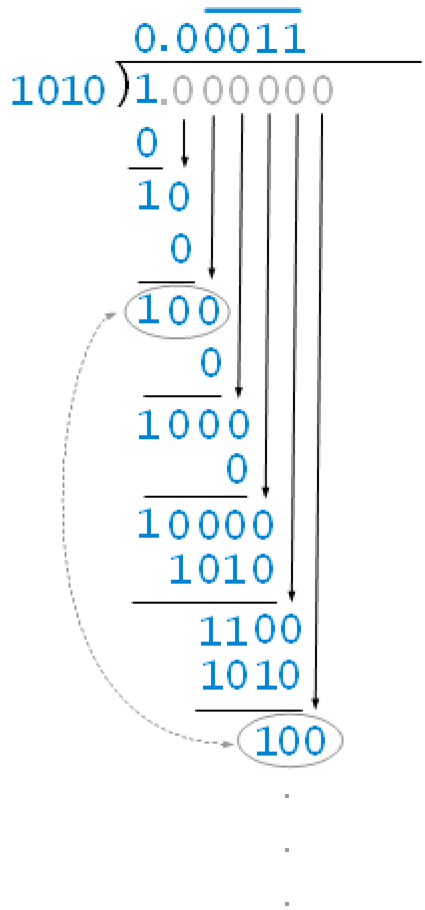
```
0.0001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011001100110011001100110011001100110011001100110011001100110011001100110011
0011...
```

Decimal 0.1 In Binary (To 1369 Places)

The bits go on forever; no matter how many of those bits you store in a computer, you will never end up with the binary equivalent of decimal 0.1.

0.1 In Binary

0.1 is one-tenth, or 1/10. To show it in binary — that is, as a bicimal — divide binary 1 by binary 1010, using binary long division:



Computing One-Tenth In Binary

The division process would repeat forever — and so too the digits in the quotient — because 100 (“one-zero-zero”) reappears as the working portion of the dividend. Recognizing this, we can abort the division and write the answer in [repeating bicimal](#) notation, as $0.0\overline{0011}$.

0.1 In Floating-Point

$0.0\overline{0011}$ is a finite representation of an infinite number of digits. That doesn't help us with floating-point. Floating-point does not represent numbers using repeat bars; it represents them with a fixed number of bits. In double-precision floating-point, for example, 53 bits are used, so the otherwise infinite representation is rounded to 53 significant bits.

Let's see what 0.1 looks like in double-precision. First, let's write it in binary, truncated to 57 significant bits:

0.000110011001100110011001100110011001100110011001100110011001…

Bits 54 and beyond total to greater than half the value of bit position 53, so this rounds up to

0.0001100110011001100110011001100110011001100110011001101

In decimal, this is

0.1000000000000000055511151231257827021181583404541015625

which is slightly greater than 0.1.

If you were to print that to 17 significant decimal digits you'd get 0.10000000000000001 (printing rounds the result as well). Note that if you were to print to less than 17 digits, the answer would be 0.1. That's just an illusion though — the computer has not stored 0.1.

It Can Be Slightly Greater or Slightly Less Than 0.1

Depending on how many bits of precision are used, the floating-point approximation of 0.1 could be *less* than 0.1. For example, in half-precision, which uses 11 significant bits, 0.1 rounds to 0.0001100110011 in binary, which is 0.0999755859375 in decimal.

0.1 Is Just One of Many Examples

0.1 is the most commonly used example in discussions about floating-point "inaccuracies" — that is why I chose it. But there are many, many more examples. How can you tell if an arbitrary decimal has an equivalent bicimal that terminates or repeats?

Of course, you could do what I did above: convert the decimal to an integer over a power of ten and then do binary division. If you get a remainder of zero, the bicimal is terminating; if you encounter a working dividend you've seen before, the bicimal is repeating. This method is great because you see the binary representation unfold before your eyes. However, it's tedious. Binary division is challenging, even if you know how to do it.

There is a simpler test: a decimal has an equivalent terminating bicimal if and only if the decimal, written as a proper fraction in lowest terms, has a denominator that is a power of two. (It takes a bit of number theory to understand why this works, but the explanation is similar to why decimals terminate only for fractions with powers of two and/or powers of five in their denominators.) By this rule, you can see that 0.1 has an infinite bicimal: 0.1 = 1/10, and 10 is not a power of two. 0.5, on the other hand, terminates: 0.5 = 5/10 = 1/2. If asked whether a decimal has a corresponding bicimal that terminates or repeats, this is the test to use.

Some Terminating Bicimals Don't Exist in Floating-Point Either

It's important to note that some decimals with terminating bicimals don't exist in floating-point either. This happens when there are more bits than the precision allows for. For example,

0.500000000000000166533453693773481063544750213623046875 converts to

0.1000000000000000000000000000000000000000000000000001 1

but that’s 54 bits. Rounded to 53 bits it becomes

0.100000000000000000000000000000000000000000000000001

which in decimal is

0.5000000000000002220446049250313080847263336181640625

Such precisely specified numbers are not likely to be used in real programs, so this is not an issue that’s likely to come up.

Summary

In pure math, every decimal has an equivalent bicimal. In floating-point math, this is just not true.

EB

Related

- Pi and e In Binary
- When Doubles Don’t Behave Like Doubles
- When Floats Don’t Behave Like Floats

Get articles by RSS (What Is RSS?)

Get articles by e-mail

Numbers in computers / Binary arithmetic, Convert to binary, Convert to decimal, Decimals, Floating-point, Fractions

15 comments

1. James

September 6, 2012 at 3:55 pm

Good article. I like to demonstrate this with the fact that 1/3 can't be represented finitely in either system. Not being able to represent 1/(2*5) doesn't seem all that stupid then.

2. **Rick Regan**

September 6, 2012 at 4:46 pm

@James,

Right. The reason I don't use examples like 1/3 in this context is that I like to show that terminating decimals, to the surprise of many, may not terminate in another base. That fractions like 1/3 don't terminate may not be as surprising.

Thanks.

3. **Georg**

September 6, 2012 at 11:15 pm

Rick, you really like to play around with ties, don't you??? But this means the rounding you mention does only happen if the rounding mode "to nearest, ties to even" is used. If, for example, "round towards zero" was used, the rounding process would yield
0.50000000000000011102230246251565404236316680908203125 ==
3F E0 00 00 00 00 00 01 (IEEE DP hex) ->
0.1000000000000000000000000000000000000000000000000001 (bicimal)

All modern programming languages harness the extented-precision internal registers (e.g. 80bits in case of Intel/AMD) of modern floating-point hardware. Programmers concerned about accuracy normally try to use these extended precision registers as much as possible. Thus, rounding issues come up in almost all programs…
Numerical-analysis evangelists like Kahan suggest to try out code using all available rounding modes (usually "to nearest, ties to even" as standard mode, but there are also the modes "towards zero" and "away from zero") in order to check whether an algorithm is subject to rounding problems or not.

From a didactical point of view, imho you are quite right to pick your examples from the large set of "unknown unknowns" of most people …

4. **Rick Regan**

September 7, 2012 at 8:57 am

Georg,

I reused that example from one of my articles on halfway case conversions, where I assumed "to nearest, ties to even". Rounding mode doesn't matter here though; the point is, you're going to lose that 54th bit no matter what.

I'm not sure of the point you're making about extended-precision.

---

5. **Jeremie Pelletier**

   September 8, 2012 at 3:01 pm

   I'm fairly sure he means the 'long double' type from C++ or the 'real' type from D. Both types mean 'at least 64bit-wide'. For the x87 coprocessor, it uses the full 80bits offered by the hardware.

   The D language site has a nice essay on floating points: http://dlang.org/d-floating-point.html

---

6. **Rick Regan**

   September 8, 2012 at 6:04 pm

   Jeremie,

   I was just trying to understand how the precision is relevant; 0.1 does not exist for any (finite) precision.

---

7. **Cloudsdale**

   January 21, 2016 at 5:59 am

   "It takes a bit of number theory to understand why this works, but the explanation is similar to why decimals terminate only for fractions with powers of two and/or powers of five in their denominators."

Can you tell us some more about that?
I was always intrigued why are 2 and 5 so special in decimal system. They reverse each other (1/2 = 0.5, 1/5 = 0.2), their inverse powers are related (1/2^5 = 1/32 = 0.03125, 1/5^5 = 1/3125 = 0.00032), and the number of non-repeating decimal digits depend on the number of factors 2 and 5 in the denominator (every 2, 5, or a pair of 2 and 5, gives one decimal digit in the expansion). But WHY is this so?
I know that 2 and 5 are factors of 10 (the base of the decimal system), but how exactly does it translate to these observations?

---

8. **Cloudsdale**

January 21, 2016 at 6:03 am

"By this rule, you can see that 0.1 has an infinite bicimal: 0.1 = 1/10, and 10 is not a power of two. 0.5, on the other hand, terminates: 0.5 = 5/10 = 1/2. If asked whether a decimal has a corresponding bicimal that terminates or repeats, this is the test to use."

So basically, the only decimal fractions that have finite binary representations are those in which the denominator ends with 0 or 5?

---

9. **Rick Regan**

January 21, 2016 at 12:48 pm

@Cloudsdale,

Yes, it is all about the factors of the base. In decimal, a fraction terminates if it can be written in the form $n/10^a$. If you are given an arbitrary fraction, it has to reduce to the form $m/(2^b \cdot 5^c)$, b and c ≥ 0, for it to terminate. From there it is trivial to put it into the form $n/10^a$: multiply by either 2^{c-b} or 5^{b-c}, depending on whether c or b is bigger, respectively. So the number of digits is n = max(b,c).

In binary, a similar rule holds: a fraction terminates if it can be written in the form $n/2^a$, so you have to be able to reduce it as such.

Regarding the patterns of powers of two and five, you might want to check out my article https://www.exploringbinary.com/seeing-powers-of-five-in-powers-of-two-and-vice-versa/ .

10. **Rick Regan**

January 21, 2016 at 12:55 pm

@Cloudsdale,

No. Decimal fractions that have finite binary representations have a denominator, after reducing to lowest terms, that is a power of two; those end in 2, 4, 6, or 8 (see my article https://www.exploringbinary.com/patterns-in-the-last-digits-of-the-positive-powers-of-two/).

11. **Vardhan**

April 4, 2016 at 2:42 am

In 0.1 if 0.000110011.. (recurring bicimal) represents the real 0.1 decimal, then the truncated bicimal (to 53 digits) should be a value lesser than 0.1 (as you are removing a positive recurring part after the truncation) and not > 0.1 as given above. Can you clarify about this?

12. **Rick Regan**

April 4, 2016 at 7:53 am

@Vardhan,

I discuss this in the article. It is due to rounding. (In double-precision, it rounds *up*.)

13. **Jama**

October 18, 2016 at 1:26 pm

Props to you sir! After reading a bunch of articles I still wasn't be able to get my head around why this was the case (folks were mentioning prime numbers) and your article made it super clear. Thanks!

14. **Yuri**

February 12, 2017 at 2:18 am

Best article i read simple and explanatory thank you

15. **Ferdinando de Magdelania**

September 3, 2017 at 10:55 pm

Is this the answer?
http://www.dec64.com/

Comments are closed.