exploringbinary.com

Correct Decimal To Floating-Point Using Big Integers

Aug. 3rd, 2011

Send to Kindle

Producing correctly rounded decimal to floating-point conversions is hard, but only because it is made to be done efficiently. There is a simple algorithm that produces correct conversions, but it's too slow — it's based entirely on arbitrary-precision integer arithmetic. Nonetheless, you should know this algorithm, because it will help you understand the highly-optimized conversion routines used in practice, like David Gay's strtod() function. I will outline the algorithm, which is easily implemented in a language like C, using a "big integer" library like GMP.

$$\frac{664613997892457936451903530140172288}{100000000000000000000}$$

Ratio of Big Integers ($2^{119}/10^{20}$) Producing the 53-Bit Significand of 1e-20

Arbitrary-Precision is Needed

Our task is to write a computer program that uses ***binary arithmetic*** to convert a decimal number — represented as a character string in standard or scientific notation — into an IEEE double-precision binary floating-point number. What makes this task difficult is that a given decimal number may not have a double-precision equivalent; an approximation — the double-precision number *closest* to it — must be used in its place.

An IEEE double-precision binary floating-point number, or *double*, represents a number in binary scientific notation, with a 53-bit significand (subnormal numbers have less than 53 bits, but I won't be discussing them) and an 11-bit power of two exponent. The limited precision of the significand is what makes approximation necessary. For example, a decimal fraction like 0.1 has an infinite binary representation, so it must be rounded to 53 significant bits. A large integer like 9007199254740997 has a binary representation that, although finite, is more than 53 bits long; it must also be rounded to 53 bits.

To round correctly, in general, you need to compute the approximation using more than 53 bits of precision; *arbitrary-precision* is required, an amount which depends on the input. An arbitrary-precision floating point or arbitrary-precision integer based algorithm could be used, but the algorithm I'll discuss uses arbitrary-precision integers — AKA *big integers*. This makes for a more elegant solution.

Big Integer Based Algorithm

The big integer based algorithm I will show you is essentially *AlgorithmM*, as described in William D. Clinger's paper "How to Read Floating Point Numbers Accurately." He presents it in the programming language Scheme, but I will present it to you in words, and by example. (Like Clinger, I will deal only with positive, normal numbers.)

53-Bit Integer Significands

The significand of a normalized double-precision floating-point number is 53 bits, with its most significant bit equal to 1. For the purposes of our conversion algorithm, we'll think of it as a 53-bit integer; that is, an integer between 2^{52} (4503599627370496) and $2^{53} - 1$ (9007199254740991).

Overview

The idea of the algorithm is to represent the decimal number as a fraction, one that produces a 53-bit integer quotient and an arbitrary-precision integer remainder. The significand is the quotient, rounded up or down according to the remainder; the power of two exponent is derived from the binary scale factor used to produce the fraction.

There are two basic cases the algorithm must handle: one where the fraction needs to be *scaled up* to produce a 53-bit quotient, and one where the fraction needs to be *scaled down* to produce a 53-bit quotient. I will demonstrate the algorithm with an example of each case.

Converting 3.14159 to Double-Precision

Here's how to convert the decimal literal 3.14159 to double-precision floating-point:

1. **Express as an integer times a power of ten**

   $3.14159 = 314159 \times 10^{-5}$

   In code: This step is done as the decimal string is parsed. The significant digits are collected (sans the decimal point) and then converted to binary as a big integer. The exponent of the power of ten is determined based on the correct placement of the decimal point. (For strings expressed in decimal scientific notation, the exponent must be converted to binary, and then adjusted for the decimal point accordingly.)

2. **Express as a fraction**

The negative power of ten, 10^{-5}, makes this a fraction, with 314159 as the numerator and 10^5 = 100000 as the denominator:

314159/100000

In code: The positive power of ten is computed as a big integer, raising ten to the absolute value of the exponent.

3. **Scale into the range [2^{52},2^{53})**

314159/100000 is a number r, $2^1 \leq r < 2^2$. We need to scale it *up*, so that we have a scaled value r' in the range $2^{52} \leq r' < 2^{53}$; we do this by multiplying the numerator by 2^{51}:

314159 x 2^{51}/100000 = 707423177667543826432/100000

(We scale the numerator because we need the scaling done *before* the division, not after!)

In code: You could use logarithms to compute the scale factor, but you'd have to use care. You would need arbitrary precision to properly distinguish values near powers of two. A simpler approach, in the spirit of sticking with integer arithmetic, is to use a loop: multiply the numerator by two until the quotient is in range. (For conversions where the quotient starts out *above* range, multiply the *denominator* by two until the quotient is in range.)

4. **Divide and round**

707423177667543826432/100000 = 7074231776675438 remainder 26432

The quotient is an integer q in the range $2^{52} \leq q \leq 2^{53} - 1$. The remainder is an integer r in the range $0 \leq r < 100000$, and it tells us which way to round q —

up or down to the next integer, which we do according to IEEE 754 round-half-to-even rounding: if *r* is less than half the denominator, we round *q* down; if *r* is more than half the denominator, we round *q* up; if *r* equals half the denominator, we round *q* down if *q* is even, and round *q* up if *q* is odd. In our example, *r* is less than half the denominator, so *q* rounds down to **7074231776675438**; this is our significand.

(In the event that *q* rounds up to 2^{53}, set it to 2^{52} instead, and decrement the exponent of the scale factor.)

In code: The remainder is easily calculated as *r* = numerator – *q* x denominator. For the “half the denominator” comparison, note that the denominator will either be 1 (for integers that start out in range and thus don’t need scaling) or divisible by 2: if it’s 1, there’s no remainder and no rounding; if it’s divisible by 2, then half the denominator is an integer, and the remainder can be compared to it with integer operations. For the test for evenness, just divide the quotient by two and check for a remainder of zero.

5. **Express in normalized binary scientific notation**

For this step, it’s easiest to think in terms of binary. The significand, 7074231776675438, converts to

11001001000011111100111110000000110111000011001101110

This value is scaled, by 2^{51}; we **unscale** it by multiplying it by 2^{-51}. Actually, we don’t multiply; we just express the number in unnormalized binary scientific notation:

$11001001000011111100111110000000110111000011001101110 \times 2^{-51}$

To be represented in the double-precision format, the significand must be **normalized**; this means we must move the radix point 52 bits to the left, and multiply the resulting binary fraction by 2^{52}. The unscaled and normalized value becomes:

1.100100100001111110011111000000011011100001100110111 x 2^{-51} x 2^{52}

By the laws of exponents, 2^{-51} x 2^{52} = 2^1, so our resulting double-precision value, in normalized binary scientific notation, is

1.100100100001111110011111000000011011100001100110111 x 2^1

In code: The code for this step is trivial: all we do is compute a power of two exponent, by adding 52 to the negative of the exponent of the scale factor.

6. **Encode as a double**

Normalized binary scientific notation maps straightforwardly to the IEEE double-precision format; here are the values of the fields for our example, in decimal:

- The sign field is 0, since the number is positive.
- The exponent field is 1024, which equals the power of two exponent (1) plus the exponent bias (1023).
- The fraction field is 2570632149304942, which is 7074231776675438 – 2^{52}, the trailing 52 bits of the significand (the leading 1 bit is implicit).

Here is the encoded double, in binary:

0100000000001001001000011111100111110000000110111000011001101110

The sign field is 0, the exponent field is 10000000000, and the fraction field is 10010010000111111001111100000000110111000011001101110.

This double has the exact decimal value of

3.14158999999999988261834005243144929409027099609375

In code: In C, you can set these values directly by casting the double as a 64-bit integer and treating the fields as integer subfields within it.

Converting 1.2345678901234567e22 to Double-Precision

Here's how to convert the decimal literal 1.2345678901234567e22 to double-precision floating-point:

1. **Express as an integer times a power of ten**

   1.2345678901234567e22 = 12345678901234567 x 10^6

2. **Express as a fraction**

   The positive power of ten makes this an integer, but for the purposes of our algorithm, we still want to express it as a fraction; the numerator is 12345678901234567 x 10^6 = 12345678901234567 x 1000000 = 12345678901234567000000, and the denominator is 1:

   12345678901234567000000/1

3. **Scale into the range [2^{52},2^{53})**

   12345678901234567000000/1 is a number *r*, $2^{73} \le r < 2^{74}$. We need to scale it *down*, so that we have a scaled value *r'* in the range $2^{52} \le r' < 2^{53}$; we do this by multiplying the denominator by 2^{21}:

12345678901234567000000/2^{21} = 12345678901234567000000/2097152

4. **Divide and round**

   12345678901234567000000/2097152 = 5886878443352969 remainder 1355712

   The remainder is *more* than half the denominator, so the quotient — our significand — rounds up to **5886878443352970**.

5. **Express in normalized binary scientific notation**

   In binary, 5886878443352970 is

   10100111010100001010110110010011100111011001110001010

   Unscaled, it is

   10100111010100001010110110010011100111011001110001010 x 2^{21}

   Normalized, it is

   1.010011101010000101011011001001110011101100111000101 x 2^{21} x 2^{52}

   Simplified, it is the resulting double-precision value

   1.010011101010000101011011001001110011101100111000101 x 2^{73}

6. **Encode as a double**

   Here are the values of the fields for our example, in decimal:

   - The sign field is 0, since the number is positive.
   - The exponent field is 1096, which equals the power of two exponent (73) plus the exponent bias (1023).

- The fraction field is 1383278815982474, which is 5886878443352970 – 2^{52}.

Here is the encoded double, in binary:

0100010010000100111010100001010110110010011100111011001110001010

The sign field is 0, the exponent field is 10001001000, and the fraction field is 0100111010100001010110110010011100111011001110001010.

This double has the exact decimal value of

12345678901234567741440

The Integers Get Huge

For very large and very small decimal numbers, the scaled fractions have very large numerators and denominators. For example, for DBL_MAX, which is defined as the decimal literal 1.7976931348623158e308, the algorithm produces this scaled fraction:

17976931348623158000000000000000000000000000000000000000000000000000

(The numerator is 1.7976931348623158 x 10^{308}; the denominator is 2^{971}. The quotient rounds to 9007199254740991, which in binary is 53 bits of 1s.)

Another example is DBL_MIN, which is defined as 2.2250738585072014e-308; it results in a fraction with even larger integers.

Handling Remaining Cases

The algorithm as presented only deals with positive, normal numbers. If you were to implement this according to the IEEE 754 specification, you'd have to handle zero, negative numbers, subnormal numbers, underflow, and overflow:

- To handle zero, check for it as a special case when parsing the input string.
- To handle negative numbers, treat the number as positive and then just negate the converted result.
- To handle subnormal numbers, reduce the precision of the significand — to between 1 and 52 bits — when scaling to 53 bits would take the power of two exponent into the subnormal range.
- To handle underflow, return zero when the result would be below the subnormal range.
- To handle overflow, return "infinity" when the result would be bigger than DBL_MAX.

Discussion

Big Integers vs. Big Floats

As I mentioned, correct conversion can be done using arbitrary-precision floating point arithmetic — AKA *big floats*. However, I think the big integer based algorithm is cleaner. Its main advantage comes from the "baselessness" of integers. You can work with them conceptually as decimal numbers, blissfully ignorant that they are implemented in binary — conversion of integers to and from binary is exact.

A big advantage of "baselessness" is in rounding. To round correctly, there must be enough precision to ensure the accuracy of the rounding bits — significant bit 54 and an arbitrary number of bits beyond. In the integer algorithm, the required precision comes automatically, through scaling. Each conversion gets the rounding precision it needs, as reflected in the integer remainder. And the calculation is simple: the remainder is compared to one-half of the denominator.

In a floating-point algorithm, precision needs to be set explicitly (either on a case-by-case basis, or once, covering the worst case). Also, the calculation is not as straightforward, requiring knowledge of the absolute location of the rounding bits. For example, consider 3.08984926168550152811e-32, a decimal number very nearly halfway between two floating-point numbers. The calculation a floating-point algorithm must make is to compare 2^{-158} (one-half ULP) to $2^{-158} + 2^{-234}$ (the value of the 77 required rounding bits).

Sometimes Arbitrary-Precision Is Overkill

The algorithm I presented is simple and works for every case, but sometimes it is overkill. For many conversions, a simple IEEE double-precision floating-point calculation does the trick. For example, 3.14159 can be converted by this line of standard C code: *double d = 314159.0/100000;*. This works because both 314159 and 100000 are exactly representable in double-precision, and a standard IEEE floating-point multiplication is guaranteed to produce a correctly rounded result. (This is one optimization used in David Gay's strtod().)

Arbitrary-precision *is* required for my second example though, since 12345678901234567 is not exactly representable in double-precision; *double d = 12345678901234567.0*1000000;* produces a result that is 1 ULP off.

(After publishing this I realized that the C statements above are processed by the compiler — constant folding. There's no guarantee that the compiler will use floating-point division and multiplication instructions; it could use its own algorithms. My point is still valid though. In a real program, the two integer arguments will not be constants, so floating-point instructions will be used — at runtime.)

Implemented In a C++ Program

I implemented the big integer algorithm in a C++ program using GMP (MPIR). I validated it by comparing its results to David Gay's strtod(). (I have used strtod()

as a benchmark against which to compare other conversion routines; by writing this program, I’ve gotten a way to confirm that strtod() in fact produces correct results.)

Summary

I showed you a simple algorithm that uses binary integer arithmetic to convert a decimal string to an IEEE double-precision binary floating-point number. The algorithm starts by representing a decimal number as fraction, with a numerator and denominator that are arbitrary-precision integers. The fraction is scaled by a power of two — positive or negative — so that the integer part of the resulting quotient will contain exactly 53 bits. The 53-bit quotient is rounded to the nearest integer, unambiguously according to the value of the integer remainder. The rounded quotient — the 53 bits of the floating-point number — is unscaled and normalized by setting the power of two exponent of the floating-point number accordingly.

---

https://www.exploringbinary.com/correct-decimal-to-floating-point-using-big-integers/