Error in Floating Point Arithmetic

Sevika Singh

Professor Kocak

CSC 210

Writing Credit Project

Floating point representation of numbers is a compact method of storing real numbers on a computer, similar to scientific notation. Representing real numbers on computers can be done is several different ways. Floating point representations had been written differently between various machines, and so was hard to translate between systems. Today there are standards by which these are written, one of the most common today being the IEEE-754 standard.

The advantage to floating point representation however, is that it can be used to a much wider range of numbers than other types of representation. This is done by representing a wide range of values with a fixed number of significant digits and scaled using an exponent. The term floating point itself means that the decimal point (or binary point) can be placed anywhere, as long as it is relative to the significant digits of the number.

A disadvantage to using scientific notation is that most numbers can be expressed in different ways, for example all of the following are equal:

-1234.5678

-1.2345678 X 10^3

-1234567.8 X 10^-3

This problem can be solved by using floating point notation though! Because computers are more efficient, each number can only have one unique representation, and so floating point numbers have to be normalized. This means that each numbers significand has to be a fraction with no leading zeroes. This works well for every number expect zero itself, and this is because zero has only zeroes in its significand. Another value that would require a different, or special representation would be NaN (not a number), as this represents the result of illegal operations, the taking the square root of a negative number.

A computer stores floating point numbers in a specific, pre-defined format. Numbers must be stored with a one bit sign, a significand (of which the length must be predetermined), and an exponent (also, of a given length). However, there is no sign bit for the exponent, and so representing negative exponents becomes a question. This can be done by using two’s complement values, but the most common way is through biasing.

Floating point numbers have several essential characteristics. Precision describes how precise a floating point value can be, and is defined as the number of bits in the significand. For example, a computer that uses eight bits for its significand has 8-bit precision. So as a result, the larger the number of bits in the significand, the greater the CPU’s precision, and so the more precise the value being stored can be. The gap is the difference between two adjacent values, and its value depends on the value of the exponent. This gap is usually expressed (let’s say for the floating point value of X) as 2^(XE^-precision). The range is set by the smallest and largest possible values. For example, a floating point number with an 8-bit significand and a 4-bit register has a range of -.11111111 X 2^7 to .11111111 X 2^7 (which equals -127.5 to 127.5). We can see how all of these characteristics come together through an example. Lets say we have a floating point representation of a number with a 1-bit sign, and 8-bit exponent with a bias of 128, and a 23-bit significand (and so have 23 bit precision). The range would be -.111 1111 1111 1111 1111 1111 X 2^127 to .111 1111 1111 1111 1111 1111 X 2^127 (or about -1.7 X10^38 to 1.7 X 10^38), and the gap would depend on the actual value itself. For example if you have .1 X 2^127, the gap is 2^(127-23) or 2^104.

Overflow is something that happens when an operation produces a result that cannot be stored in the computers floating point registers. If the exponent is larger than the maximum exponent that can be allowed, then there is an overflow. Overflow can be positive or negative, depending on the sign of the value.

Underflow is something similar, but occurs when an operation produces a result that is between zero and either the positive or negative smallest possible value. Underflow can also be positive or negative.

These overflows can begin to hint at problems with this system. It is seen that no matter how many bits the significand has, there is no way it can represent every possible number. Many operations will produce results that have significands that are too large for the CPU registers. Because of this, values must be transformed to fit, through a process called rounding.

The goal of rounding is to find a representation that is as close to the actual value as possible. For example, rounding the value .1011 1010 to 4 bits would give you the value .1100. This is known as rounding to nearest, or unbiased rounding. This has a maximum error of +-1/2 LSB, or one half the value of the least significant bit of the rounded result. This is the most common method of rounding, but there are several others. One method is known as “rounding toward 0”, in which the extra bits are just truncated. It is simple, but can produce results that are up to 1 LSB off! Another method is “rounding towards +infinity”, which is essentially a ceiling function for fractions; all values are rounded up to the next possible value. This results in negative values becoming truncated and positive values being rounded up to the next valid value. A similar method is “rounding towards –infinity”, where negative values are rounded down and positive values are truncated.

These rounding methods (except round toward zero) require additional bits beyond those in the final representation, for extra precision. The most important of these bits is the round bit (which is followed by the guard bit). A third bit that may be used is the sticky bit. These bits are all extension of the result.

Despite there being so many tools and methods for storing numbers, binary floating point representation still has something known as representation error. This is the difference between the approximated representation of a number and its exact mathematical value.

Lets look at an example of an error in floating point arithmetic, in which the computer does not abide by the mathematical laws because, it doesn’t have the full capability to be perfect. The following calculation does not give the expected value:

> 0.1 + 0.2

ans = 0.30000000000000004

It would appear that 1/10 and 2/10 either cannot be exactly represented as binary, or perhaps that the addition of the values shows that the representation of 1/10 and the representation of 2/10 add up to more than 3/10, guaranteeing that their representation is larger than the actual values.

Such errors in representation can cause many problems, whether or not the end-users are aware of them. For example, say you take the calculation seen above, and do it over and over again. With each subsequent operation, the error is going to continue to multiply. Now, however small the errors maybe be, they will still exist nonetheless. This may or may not cause problems, but the chances that they will are great. If someone expects a computer to calculate 1+1=2, but in reality if the computer calculates something like 1+1 to be 2.00000000000000001, then depending on what the data is being used for, the errors can cause serious consequences.

The point of researching floating point representation errors was to examine the faults of technology, and how much we depend on it, but perhaps should not. It is always safer to count on human calculations and operations (despite our imperfections and faults), but it is more common for us to count on and fully trust technology, even though it is proven and guaranteed that computers will make errors, and thus, cause errors and miscalculations for us.