Precision Issues in Floating-Point Arithmetic

Prof. Naga Kandasamy
ECE Department, Drexel University

March 9, 2021

The following notes are derived from these sources:

- Steven Chapra, *Applied Numerical Methods with MATLAB for Engineers and Scientists*, McGraw Hill, 2012.
- Randall Hyde, *Write Great Code. Volume I: Understanding the Machine*, No Starch Press, 2004.

Floating point (FP) numbers provide only an approximation of real numbers. This is because there is an infinite number of possible real values, whereas the FP representation uses a finite number of digits or bits and therefore, can represent only a finite number of values. When the underlying FP representation cannot exactly represent the real value, the FP number must instead use the closest value that it can exactly represent. These notes briefly discuss the implications of such approximations on the accuracy of numerical calculations.

To represent real numbers, most FP formats use some number of bits to represent a *mantissa* and a smaller number of bits to represent an *exponent*.[1] The mantissa is a base value, usually falling within a limited range, say between zero and one in the binary system. The exponent is a multiplier that when applied to the mantissa produces values outside this range. To facilitate our discussion, let us adopt an FP format on a hypothetical decimal computer with a mantissa of three significant digits and an exponent with two digits, where both the mantissa and the exponent are signed values. Our format looks like

$$\pm m.mm\ \mathbf{e} \pm ee, \tag{1}$$

where m represents the digits for the mantissa and e represents the digits allocated for the exponent. The above representation can approximate all positive values between 0.00 and 9.99×10^{99} as well as values on the negative side of the range. However this format cannot exactly represent all values in this range; for example, to represent a value like $9,876,546,234$, the above FP representation would have to approximate this value using 9.88×10^9 (or 9.88e+9 in the scientific or programming language notation). As another example, a simple rational number with a finite number of digits

[1] IEEE Standard 754 floating point is the most common representation today for real numbers on computers. The layout for the single precision FP value is as follows: 1 sign bit, followed by 8 bits for the exponent, and 23 bits for the mantissa. The layout for double precision comprises of 1 sign bit, followed by 11 bits for the exponent, and 52 bits for the mantissa. The mantissa uses the sign-magnitude notation to represent both positive and negative numbers whereas the exponent is biased—using excess 127 coding for single precision and excess 1023 for double precision.

such as $2^{-5} = 0.03125$ will have to be stored as 3.13×10^{-2}, thereby introducing a *roundoff error*. This represents a relative error of

$$\left|\frac{0.03125 - 0.0313}{0.03125}\right| = 0.0016.$$

While we could store a number like 0.03125 exactly by expanding the digits of the significand, quantities with infinite digits must always be approximated. For example, the constant $\pi (= 3.14159\ldots)$ would have to be represented as 3.14×10^0 in our format, leading to a relative error of

$$\left|\frac{3.14159 - 3.14}{3.14159}\right| = 0.0005.$$

Although adding more significand digits can improve the approximation, such quantities will always have some roundoff error when stored in a computer.

There is a another subtle disadvantage with our FP format in that it can provide multiple representations for the same value; for example, 5.00e+1 and 0.50e+2 are different representations of the same value. As there are a finite number of different representations possible (since we only have a finite number of bits or digits), whenever a single value has two possible representations, that's one less different value the format can represent. This issue is solved by *normalizing* the number when it is stored in the computer as a binary number.[2]

Yet another issue with the FP format is that it complicates the process of addition and subtraction. When adding or subtracting two FP numbers, we must adjust the two values so that their exponents are the same. For example, when adding 1.23e1 and 4.56e0, we could convert 4.56e0 to 0.456e1 and then add them, producing 1.686e1. Unfortunately, this result does not fully fit into the three significant digits of our format, and so, we must either *round* or *truncate* the result to three significant digits. Since rounding generally produces the most accurate result, let's round the result to obtain 1.69e1. Here we were able to round the result because we maintained four significant digits during the calculation. If on the other hand, our FP calculation was limited to only three digits at all times, we would have had to truncate 0.456e1 to 0.45e1 prior to the addition operation, obtaining 1.68e1 as the result, which is even less precise. So, to improve the accuracy of FP calculations, it is necessary to use extra digits, called *guard digits* (or *guard bits* in the case of binary arithmetic) during the calculation.

The accuracy lost during a single computation usually isn't too bad. Of greater concern is the loss of precision during a sequence of FP operations where the error can accumulate and greatly affect the computation itself. Suppose we add 1.23e3 and 1.00e0. Adjusting the numbers so that the exponents are the same before the addition produces 1.23e3 + 0.001e3. The sum of these values is

[2]The process is basically the same as when normalizing a floating-point decimal number. For example, decimal 1234.567 is normalized as 1.234567×10^3 by moving the decimal point so that only one digit appears before the decimal. The exponent expresses the number of positions the decimal point was moved left (positive exponent) or moved right (negative exponent). Similarly, the floating-point binary value 1101.101 is normalized as 1.101101×2^3 by moving the decimal point 3 positions to the left, and multiplying by 2^3. The mantissa is always represented in binary as composed of an implicit leading bit left of the radix point and the fraction bits to the right of the radix point as $1.fff\ldots f$, where $fff\ldots f$ denote the fractional part. Since in a normalized mantissa, the digit 1 always appears to the left of the decimal point, the leading 1 is omitted from the mantissa in the IEEE storage format because it is redundant.

1.23e3 even after rounding. This is quite reasonable: after all, if we only maintain three significant digits, adding a small value should not affect the result. However suppose we were to add 1.00e0 to 1.23e3 ten times. The first time we add 1.00e0 to 1.23e3, we get 1.23e3. We obtain the same result the second time, and the third time, and so on until the tenth time. On the other hand, if we had added 1.00e0 to itself ten times, and then added the result 1.00e1 to 1.23e3, we would obtain a different result: 1.24e3. So, this is an important fact to know about FP arithmetic: *the order of evaluation can affect the accuracy of the result*. Our results would be better when adding or subtracting numbers if their relative magnitudes (that is, the size of the exponents) are similar. So, if you are performing a chain calculation involving addition or subtraction, you should attempt to group the operations so that you can add or subtract values whose magnitudes are close to each other, before adding or subtracting values whose magnitudes are not so close.

FP addition is not guaranteed to by associative; that is, given three FP numbers x, y, and z, equality may not hold when performing the following arithmetic:

$$x + (y + z) = (x + y) + z.$$

Problems occur when adding two large numbers of opposite signs plus a small number. For example, suppose $x = -1.5\text{e}38$, $y = 1.5\text{e}38$, and $z = 1.0$ are single-precision floating point numbers. Recall that we have 23 bits to store the mantissa. Then,

$$\begin{aligned} x + (y + z) &= -1.5 \times 10^{38} + (1.5 \times 10^{38} + 1.0) \\ &= -1.5 \times 10^{38} + 1.5 \times 10^{38} \\ &= 0.0 \end{aligned}$$

On the other hand,

$$\begin{aligned} (x + y) + z &= (-1.5 \times 10^{38} + 1.5 \times 10^{38}) + 1.0 \\ &= 0.0 + 1.0 \\ &= 1.0 \end{aligned}$$

Since the mantissa field has limited precision and since 1.5e38 is so much larger than 1.0, the sum 1.5e38 + 1.0 is still 1.5e38. That is why the sum of x, y, and z is 0.0 or 1.0 depending on the order of the FP additions.

Another problem with addition and subtraction is a phenomenon called *false precision*. Consider the computation 1.23e0 - 1.22e0 which produces 0.01e0. Although this is mathematically equivalent to 1.00e-2, this form suggests that the last two digits are both exactly zero. This is not the case: the two non significant zeros have simply been appended. We actually only have a single significant digit after this computation. This is related to another phenomenon called *subtractive cancellation* in which the difference will not have as much precision of either the subtrahend or the minuend. Let us use a hypothetical decimal computer with a 4-digit mantissa and a 1-digit exponent to illustrate this issue. Consider the following difference of two nearly equal numbers: 3.593 - 3.574 = 0.019. All three numbers appear to have the same precision: four decimal digits in the mantissa. However, because 3.593 represents any number in the inclusive range [3.5925, 3.5934], and 3.574 represents any number in the range [3.5735, 3.5744], the actual difference may

be any number in the interval [0.018, 0.019], and therefore while we have an apparent precision of four digits, we actually have zero significant digits. As another example, let us use four decimal digits of precision to subtract 3.523 from 3.537. The answer is 0.014. However, since 3.523 could represent any number in [3.5225, 3.5234] and 3.537 could represent any number in [3.5365, 3.5374], and thus the actual answer could be any number in the range [0.013, 0.014], and thus, while the original numbers had a maximum relative error of 0.00014, the answer has a relative error of 0.071 which is larger by a factor of 500. As a final example for subtractive cancellation, consider the numerical formula for the derivative of a function $f(x)$. The derivative is given by $(f(x+h)-f(x))/h$. If we use $f(x) = x^2$ with $x = 3.253$ and $h = 0.002$, we get an approximation $(3.255^2 - 3.253^2)/0.002 = (10.60 - 10.58)/0.002 = 10$, which is a poor approximation for the actual analytical value of the derivative, which is $df(x)/dx = 2x = 6.506$.

The foregoing discussion brings us to the second important rule concerning FP arithmetic: *whenever subtracting two numbers with the same signs or adding two numbers with different signs, the accuracy of the result may be less than the precision available in the FP format*.

Multiplication and division do not suffer from the precision problems experienced by addition and subtraction since we do not have to adjust the exponents before these operations. For multiplication, we add the exponents and multiply the mantissa values, whereas for division we subtract the exponents and divide the mantissa values. Multiplication and division, however, will exacerbate any accuracy error that already exists in a value. For example, if we multiply 1.23e0 by 2 when we should be multiplying 1.24e0 by 2, the result becomes even less accurate than it was. So, the third important rule when working with limited-precision arithmetic is: *when performing a chain of calculations involving addition, subtraction, multiplication, and division, try to perform the multiplication and division operations first*. One can usually arrange a calculation, by applying normal algebraic transformations, such that multiplication and division occur first. For example, suppose we wish to compute: $x \times (y + z)$. Normally we would add $y + z$ and then multiply the result by x. However, we will get a little more accuracy if we first transform $x \times (y + z)$ to $x \times y + x \times z$, and this way we can perform the multiplication operations first before the addition operation. The tradeoff here is speed: we need to perform two multiplications rather than one.

Finally, comparing two FP numbers must be done with great care. Given the inaccuracies present in any computation, you should never compare two FP values to check if they are equal. In other words, if you perform a calculation and then do this comparison:

```
if (result == expectedResult) then ...
```

then it is unlikely that the comparison will be true. This is because in a binary FP format, different computations that produce the same mathematical result may differ in their least significant bits. For example, adding 1.31e0 + 1.69e0 should produce 3.00e0 and so should adding 1.50e0 + 1.50e0. However if we were to compare (1.31e0 + 1.69e0) to (1.50e0 + 1.50e0) we may find that these sums are not equal to one another. The test for equality succeeds if and only if all the bits or digits in the two operands are the same, that is, if the bit pattern is identical. Since it is not necessarily true that two seemingly equivalent FP computations will produce exactly the same bit pattern, a straight comparison for equality may fail, when algebraically such a comparison ought to succeed.

One way to test for equality between FP numbers is to determine how much error or tolerance you will allow in a comparison and then check to see if the values are within this error range. If you

decide, based on some error analysis, that the result should always be within some `epsilon` of the expected result then you can change your comparison to this:

```
if (fabs(result - expectedResult) < epsilon) then ...
```

You must take care when choosing a value for `epsilon`. This value should be greater than the largest amount of error that may creep into your calculations. Say you perform a calculation that has an expected answer of about 10,000 and choose an `epsilon` value of 0.00001. Because FP math is imperfect you may not get an answer of exactly 10,000; you may be off by one or two in the least significant bits of your result. If you are using 4-byte floats and are off by one in the least significant bit of your result, then instead of `expectedResult = 10,000` you will get `result = 10000.000977` which is the closest 4-byte float to 10,000 without being 10,000. Now,

```
diff = fabs(result - expectedResult) = 0.000977
```

which is 97.7 times larger than our epsilon value. So, our comparison tells us that `result` and `expectedResult` are not nearly equal, even though they are adjacent FP numbers. So, using an epsilon value 0.00001 for float calculations in this range is meaningless.

An `epsilon` value of 0.00001 is appropriate for numbers around one, too big for numbers around 0.00001, and too small for numbers around 10,000. A more generic way of comparing two FP numbers that works regardless of their range is to check the *relative error* obtained as:

```
relativeError = fabs((result - expectedResult) / expectedResult)
```

Comparing for less than or greater than creates similar problems encountered when checking for equality. For example, suppose that some chain of calculations in our simplified decimal representation in (1) produces a result of $a = 1.25$, which is only accurate to plus or minus 0.05. That is, the real value could be somewhere between 1.20 and 1.30. Also assume that a second chain of equivalent calculations produces the result $b = 1.27$, which is accurate to the full precision of our FP result. That is, the actual value, before rounding, is somewhere between 1.265 and 1.275. Now if we compare the result of the first calculation (1.25) with the value of the second calculation (1.27), then we conclude that $a < b$. Unfortunately, given the inaccuracy present in the calculation of a, this might not be true. That is, if a happens to be in the closed interval 1.27 to 1.30, then reporting that $a < b$ is false. The only reasonable test is to see if the two values are within the error tolerance of one another. If so, treat the values as equal (so one wouldn't be considered less than or greater than the other). If you determine that the values are not equal to one another within the desired tolerance, then compare them to see if one value is less than or greater than the other.