Chapter 7

Numerical Systems

7.1 Floating-point systems

We can't represent every real number on a computer. We use a "floating-point system"[1] instead — given a fixed β, fixed number of digits t, and a range $[e_{\min}, e_{\max}]$ of exponents (integers), we can represent only numbers of the form:

$$\pm d_0.d_1 \dots d_{t-1} \times \beta^e,$$

... where the $d_i \in [0, \beta - 1]$ are called the digits (and the sequence of digits is called the MANTISSA), and $e \in [e_{\min}, e_{\max}]$ is the EXPONENT. (There's also a sign, costing at least a bit).

Here's an example. If $\beta = 10$, $t = 3$, $e_{\min} = -4$, and $e_{\max} = +4$, then you can represent $1/4$ as $+0.25 \times 10^0$ or $+2.50 \times 10^{-1}$. You can represent $1/3$ as $+3.33 \times 10^{-1}$ (note that $+0.33 \times 10^0$ loses one digit of precision). Notice that there are multiple representations, so we agree on a NORMALIZED mantissa: we require that the first digit $d_0 \neq 0$ unless we are representing 0 itself.

Using this normalized floating-point system:

- The smallest positive number is $+1.00 \times 10^{-4} = 0.0001$.
- The largest positive number is $+9.99 \times 10^4 = 99900$.

Another example. Suppose $\beta = 2$, $t = 3$, $e_{\min} = -2$, $e_{\max} = +3$. Numbers (other than 0) have the form

$$\pm 1.d_1 d_2 \times 2^e.$$

- Smallest positive number: $(1.00)_2 \times 2^{-2} = 1/4$.
- Largest positive number: $(1.11)_2 \times 2^3 = 14$.

Draw these out on a number line, and note that the larger numbers are spaced further apart, since a difference of 1 in the last digit represents a larger magnitude when the exponent is larger). For example, $(1.01)_2 \times 2^{-1} - (1.00)_2 \times 2^{-1} = 1/8$, versus $(1.01)_2 \times 2^2 - (1.00)_2 \times 2^2 = 1$. However, the percentage remains constant:

$$\frac{1/8}{2^{-1}} = 1/4 = \frac{1}{2^2}.$$

7.2 Expressing real numbers

Rounding

Most numbers are not exactly representable in a floating-point system using a given base β. For example, when $\beta = 10$, you cannot represent $1/3$ exactly (no matter how large t is), so we used 3.33×10^{-1} when $t = 3$. What should we do with something like the base of natural logarithms, $e = 2.718281828\ldots$? Two approaches are used:

- Round to nearest: 2.72×10^0.
- Truncate to zero: 2.71×10^0.

Overflow

There is no way to represent a number larger than the largest floating-point number. In our first example, there is no way to represent 99901 or greater. Overflow in a floating-point system occurs when we want to use an exponent larger than $e_{\max}$ to express a number.

Underflow

There is no way to represent a positive number smaller than the smallest positive floating-point number. In our first example, there is no way to represent 0.00001 (or a smaller positive number). Underflow in a floating-point system occurs when we want to use an exponent smaller than $e_{\min}$ to express a number.

Absolute rounding error

We can calculate the difference between the true value we're trying to represent and the value of its floating-point representation. For example, the absolute error in our representation of e is $|2.71 - 2.718281828\ldots|$ $= |0.008281828\ldots|$.

Relative error

100 and 100.1 are "closer" than 1 and 1.1, even though the absolute difference is 0.1 in both cases. Look at the size of the error in terms of the size of the value being represented.

Relative error: For $x \neq 0$, the relative error between the approximate value x' and the "real" value x is

$$\frac{|x - x'|}{|x|}$$

For example, $|1.1 - 1|/|1.1| = 0.0909 \approx 9\%$. However, $|100.1 - 100|/|100| = 0.000999\cdots \approx 0.1\%$.

Relative error in round-to-nearest

When we round numbers to represent them in a floating-point system, can we bound relative error for positive numbers with no overflow or underflow?[2].

In general, a number of the form

$$d_0.d_1 \cdots d_{t-1} d_t \cdots \times \beta^e$$

...gets rounded either up or down to one of

$$d_0.d_1 \cdots (d_{t-1}+1) \times \beta^e$$
$$d_0.d_1 \ldots d_{t-1} \times \beta^e$$

The representation of the first number may be different (the $+1$ may cause a number of "carries" in the addition), but the value is the same. The difference between these two numbers is simply a 1 in the position occupied by d_{t-1}, for a difference of $0.00 \cdots 1 \times \beta^e = \beta^{e-(t-1)}$. Since we round to nearest, our error is at most half this value:

$$\frac{\beta^{e-(t-1)}}{2}.$$

The relative error can be calculated, since we use the convention that the leading digit, d_0, is non-zero, so the smallest denominator (hence the largest bound) is when $d_0 = 1$, giving a relative error of:

$$\frac{|\beta^{e-(t-1)}/2|}{|1.0 \cdots 0 \times \beta^e|} = \frac{\beta^{1-t}}{2}$$

This matches our intuition that by increasing t (the number of digits) we get more precision.

In our example of a floating-point system with $\beta = 2, t = 3$, this gives a bound on the relative error of round-to-nearest of $2^{1-3}/2 = 1/8$. This is also clear from the number line or the 24 values in this representation.

The IEEE standard

You may be familiar with using floating-point numbers in Java or other programming languages. Typically, modern computers support the IEEE 754 standard that specifies the systems we use (sets β, t and $[e_{\min}, e_{\max}]$) and how they are physically stored in memory.

You will typically see two types of floating-point numbers: `float` or single-precision, which use 32 bits of memory to represent a number, and `double` or double-precision, which use 64 bits. Single-precision numbers have $\beta = 2$, $t = 24$, $e_{\min} = -126$, and $e_{\max} = +127$. Double-precision numbers have $\beta = 2$, $t = 53$, $e_{\min} = -1022$, and $e_{\max} = +1023$.

7.3 Computing with floating point numbers

Addition

Addition in a floating-point system (and subtraction, which is just adding the negation) operates much like schoolhouse addition. We need to line up the radix point, then do regular addition. For example, take $\beta = 10$, $t = 3$, $e \in [-2, +2]$, and consider the sum $x + y$ where $x = 1.65 \times 10^2$ and $y = 2.71 \times 10^1$. First we need to get a common exponent (meaning usually one of the numbers must be denormalized) before doing the addition:

$$\begin{array}{rll} & 1.65 & \times 10^2 \\ + & 0.271 & \times 10^2 \\ \hline & 1.921 & \times 10^2 \end{array}$$

... and the answer must be rounded (and, if necessary, normalized) to 1.92×10^2. Note that during normalization the exponent can change (for example, if a carry happened in the first digit or if the first digit becomes zero).

Multiplication

For multiplication, we don't need the exponents to agree: we can find the product of the mantissas first, then add the exponents together. To compute $x \cdot y$ in this system:

$$\begin{array}{rll} & 1\,.\,6\,5 & \times 10^2 \\ \times & 2\,.\,7\,1 & \times 10^1 \\ \hline & 4\,.\,4\,7\,1\,5 & \times 10^3 \end{array}$$

... which is rounded (and, if necessary, normalized) to 4.47×10^3. In practice, we usually only need to "remember" one extra digit (beyond the t digits) to figure out the rounding direction, so the exact answer is never really computed (in this example, the "...15" won't be computed).

Accumulation of error

Since we can't represent all real numbers exactly in a floating point representation, what happens when we repeat operations? For example, take $\beta = 10$, $t = 3$, $e \in [-2, +2]$, and consider the sum

$$100 + 0.1 + 0.1 + \cdots + 0.1 \qquad [n \text{ times}]$$

The first part of the sum, $100 + 0.1$ is represented by 1.00×10^2. As we add additional terms with value 0.1, the result is still 1.00×10^2, since our representation cannot represent that additional 0.1. The relative error can become arbitrarily large, given large enough n. If you play around with `FloatExample.cumulativeError` (on the web page), with say `t1` $= 1.0$, $n \geq 10$, and `t2` $= 10^{-16}$, you'll see this demonstrated.

The easy way to work around this particular problem (at least when n is not too big) is to add the 0.1 terms first, and then add the 100 — in other words, addition is NOT associative for floating-point numbers: $(a + b) + c \neq a + (b + c)$.

Catastrophic cancellation

Use the same floating-point system as in the previous example to compute $b^2 - 4ac$ for $b = 3.34$, $a = 1.22$, and $c = 2.28$. The exact value is $0.0292 = 2.92 \times 10^{-2}$, and this exact value is representable in our floating-point system. Look at how the value is calculated, though:

$$\begin{aligned} b^2 &= (3.34)^2 \\ &= 11.1556 \approx 1.12 \times 10^1 \\ 4ac &= 4 \times 1.22 \times 2.28 \\ &= 4.88 \times 2.28 \\ &= 11.1264 \approx 1.11 \times 10^1 \\ b^2 - 4ac &\approx 1.12 \times 10^1 - 1.11 \times 10^1 \\ &= 0.01 \times 10^1 = 1.00 \times 10^{-1} \end{aligned}$$

Compared to our exact answer of 2.92×10^{-2}, this has a relative error of

$$\frac{|0.0292 - 0.1|}{0.0292} = \frac{0.0708}{0.0292} = 2.424\ldots > 240\%.$$

Subtracting two floating-point numbers that are very close together leaves very few significant digits — a great deal of information is lost. Since the true value is very small, the round-off error becomes much more significant, and sometimes becomes much larger than the value being computed (see above).

The expression $b^2 - 4ac$ crops up in the solution to the quadratic equation $ax^2 + bx + c = 0$. The general form of the solution for the two roots x_1 and x_2 is

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

We may not have to worry about the large relative error in $b^2 - 4ac$, since it may be small in absolute value compared to $-b$. Here's a case where computing x_1 (using the values that lead to catastrophic cancellation above) gives a fairly acceptable value using floating-point operations:

$$x_1 = \frac{-3.34 + \sqrt{0.1}}{2 \times 1.22} = \frac{-3.34 + 0.316}{2.44} = -1.24$$

Compare this to the result if there were no error in the computation of $b^2 - 4ac$, which is:

$$x_1 = \frac{-3.34 + \sqrt{0.0292}}{2 \times 1.22} = \frac{-3.34 + 0.171}{2.44} = -1.30$$

...for a relative error of less than 5%.

7.4 Stability

There is a built-in problem with the formulas used above to compute the roots of a quadratic equation: if $b^2 - 4ac$ is close to b^2, then there will be catastrophic cancellation between $-b$ and $+\sqrt{b^2 - 4ac}$. This is separate from the catastrophic cancellation that may happen if b^2 is close to $4ac$.

Definition: a formula (or algorithm) is called unstable iff errors in the input values get magnified during the computation (i.e., iff the relative error in the final answer can be larger than the relative error in the input values).

Dealing with instability

Our first example $(100 + 0.1 + \cdots)$ was unstable, but there was an easy way to use a different algorithm that is stable: perform the operations in a different order. Typically, if we're adding up a series of numbers, we always want to add the smallest ones together first (and repeat!).

Our second example $(b^2 - 4ac)$ was also unstable, because of potential catastrophic cancellation, and there is, unfortunately, no easy fix. You could increase the number of significant digits to make the round-off error smaller, but you will still have the potential for catastrophic cancellation when you subtract numbers that are very close, even with your increased precision. The formula is unstable.

Our third example is also unstable (it includes the instability of example 2, plus its own instability), but the possibility of cancellation between $-b$ and $+\sqrt{b^2 - 4ac}$ can be avoid by changing the formula. Suppose $b > 0$ (otherwise swap the role of x_1 and x_2 below if $b < 0$). Then avoid the subtraction in the numerator of the quadratic formula:

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}.$$

Since both $-b$ and $-\sqrt{b^2 - 4ac}$ have the same (negative) sign, there will be no catastrophic cancellation. Now compute x_1 using x_2

$$x_1 = \frac{c}{ax_2} \qquad \text{(multiply the two roots to see this)}$$

This formula involves no subtraction, so there is no catastrophic cancellation.

In general, there are two ways to deal with unstable formulas or algorithms:

- Increase the precision (the number of significant digits). This does not change the fact that the formula or algorithm is unstable, but can help minimize the magnitude of the errors, for some inputs.
- Use a different, more stable algorithm or formula to compute the result. When possible, this is preferred.

Remember: floating-point numbers are *not* real numbers. They are very useful, but always be aware of their limitations and when bad things might happen. Of course, not all cancellation or rounding errors are catastrophic: you need to check the values you (plan to) use to see whether a problem will occur, and how much you can trust your answer. Often unstable formulae are faster and work the majority of the time, so are good enough in most cases.

7.5 Conditioning

In the previous example we examined the error produced in calculating the roots of $ax^2 + bx + c$. The numbers a, b, and c may come from measurement and already have some error associated with them. By using a stable algorithm we get a relatively correct answer for slightly incorrect inputs.

But, independently of the particular algorithm used to compute roots, what can be said about the effect that errors in the measurement of a, b, and c have on values of roots? That is, if the values of a, b, or c change slightly, could the roots change dramatically?

To simply the discussion, let's look at the special case of a quadratic formula $x^2 - c = 0$ (so we're finding $\sqrt{c}$).

Suppose that $c = 0.25$, but we use a bad approximation $c' = 0.36$ instead of the true value. Then our answer will be 0.6 instead of 0.5. The relative error in the input is $0.11/0.25 = 0.44$, while the relative error in the result is $0.1/0.5 = 0.2$. Taking the square root makes the relative error smaller!

This isn't something special about the particular case we chose. Let's work the algebra to see the general case of computing $\sqrt{c}$ using an approximation c' of c. The ratio of the relative error of the result to the relative error of the input is (assuming c and c' are close enough to have the same sign):

$$\begin{aligned}\frac{|\sqrt{c}-\sqrt{c'}|/|\sqrt{c}|}{|c-c'|/|c|} &= \sqrt{c}\frac{|\sqrt{c}-\sqrt{c'}|}{|c-c'|} \quad [\text{since } |c|/\sqrt{c} = \sqrt{c}] \\ &= \frac{\sqrt{c}}{\sqrt{c}+\sqrt{c'}} \quad [\text{since } |c-c'| = (\sqrt{c}+\sqrt{c'})(\sqrt{c}-\sqrt{c'})].\end{aligned}$$

This ratio is always less than 1 (so the error improves), and when c' is very close to c, the ratio is close to 1/2. So the error is never increased, and as the measurements improve it is reduced to almost 1/2.

When computing the function $f(x)$ using the approximation x' instead of the true value x, we say that the CONDITION NUMBER is equal to the ratio of the relative error of the result and the relative error of the input, i.e.:

$$\frac{|f(x)-f(x')|/|f(x)|}{|x-x'|/|x|}.$$

If you take the limit as $x' \to x$ (and assume that f is differentiable, and that $f(x) \neq 0$), this is:

$$\lim_{x'\to x}\frac{|f(x)-f(x')|/|f(x)|}{|x-x'|/|x|} = \frac{|x|}{|f(x)|}\lim_{x'\to x}\frac{|f(x)-f(x')|}{|x-x'|} = \frac{|xf'(x)|}{|f(x)|}$$

For $f(x) = \sqrt{x}$, the condition number is (well, do the derivative)![3] Not all functions have good condition numbers. Compute the condition number for $\cos(x)$,[4] and you'll see that you can get a huge condition number by choosing an appropriate x.[5]

Relationship between conditioning and stability

A problem f (that is, computing some function $f(n)$) is WELL-CONDITIONED iff *exact* computation of the output doesn't magnify the relative error of the input much.

An algorithm A for a problem f (that is, a method for computing $f(n)$) is STABLE iff it doesn't magnify the relative error of the input much.

See the difference? If an algorithm A is stable, then f is well-conditioned. (What's the contrapositive?[6]) The converse is not true: f might be well-conditioned but we can't find a stable algorithm to compute it.

7.6 Truncation

We have looked at some errors caused by the necessity of computing on inexact inputs (either due to finite precision of measurement, or finite precision of representation). However, even if our inputs were completely exact, most algorithms do not compute exact values and then round them.

In practice, to add two numbers on a computer, the number with the smallest exponent is rounded to "align"[7] the decimal point with the number with the larger exponent, and then addition is carried out, for example:

$$9.8765 \times 10^3 + 7.6543 \times 10^0 \approx (9876.5 + 7.7) \times 10^0 = 9.8842 \times 10^3$$

The true value is never computed. All arithmetic is implemented as a compromise between using small amounts of time and storage, and losing as little information as possible. There are libraries that allow arbitrary amounts of precision, but in the end this compromise is still necessary.

For other functions, like $\sqrt{1+x}$ or e^x or $\sin x$, we borrow tools from calculus, such as the Taylor series:[8]

$$\begin{aligned}
\sqrt{1+x} &= 1 + \frac{x}{2} - \frac{x^2}{8} + \frac{x^3}{16} - \frac{5x^4}{128} + \cdots \\
e^x &= 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots \\
\sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots
\end{aligned}$$

Eventually computation has to stop, so these series are TRUNCATED. Take care with the order of operations to avoid accumulated error and cancellation. In all cases, we're stuck with an approximation of an infinite series. TRUNCATION is defined as using an approximate formula (for example, a finite sub-series of a Taylor series) to compute a value. Calculus will provide you with a bound on how bad the approximation can be.

The truncation error is independent of the rounding error. Rounding error affects the input (use rounded x' instead of true x for the input) and the representation of the result. Truncation affects the computations performed (instead of computing true function $f(x')$, compute truncated function $\widehat{f}(x')$). Both sources contribute to give your total error.

7.7 Error summary

WE WANT	WE GET
exact value: x	approximate value: x'
exact function: f	approximate function: $\widehat{f}$
exact result: $f(x)$	computed result: $\widehat{f}(x')$

The absolute error for the entire computation has two sources of error. The first term is contributed by truncation or instability, the second term is due to rounding (in measurement or representation):

$$|\widehat{f}(x') - f(x)| = |(\widehat{f}(x') - f(x')) + (f(x') - f(x))|.$$

Exercises

1. Suppose you have a floating-point representation that has $\beta = 10$ (base, or radix, 10), $t = 5$ digits, and $e \in \{-5, 5\}$. Suppose x and y are non-zero numbers that are represented by x' and y' (respectively) without overflow in your representation, using round-to-nearest. (In the questions below, you may express your result as a fraction, or use a calculator.)

   (a) What is the largest possible relative error, $|x - x'|/|x|$. Explain your answer.[9]

   (b) If $x = 3.0000500000$ and $y = 3.0000499999$, what is the relative error $|(x-y)-(x'-y')|/|x-y|$?[10]

   (c) What is the relative error if, in the previous part, we increase precision from $t = 5$ to $t = 8$ digits?[11]

2. Consider the IEEE floating-point standard systems.

   (a) What is the maximum relative error in representing a real number in single-precision, when neither overflow nor underflow occurs?

   (b) How many decimal digits can we guarantee will be represented exactly when using single-precision numbers? How about when using double-precision numbers?

   (c) Notice that, when moving from single-precision to double-precision, the number of mantissa digits more than doubles, but the range of exponents only increases slightly (adding just a few bits). Why do you think this choice might have been made, allocating nearly all of the new bits to the mantissa?

3. Computing the expression $x^2 - 100$ is susceptible to error for certain values of x.

   (a) Explain what kind of error can occur and for what values of x it occurs. Illustrate your claim with an example.

   (b) Is computing this expression (in this form) stable? Explain why or why not.

   (c) Reformulate the expression to reduce the error you described in (a). Explain how this error is reduced in your new formulation.

Chapter 7 Notes

[1]The name “floating-point system” comes from the fact that the actual location of radix point (decimal point or binary point) “floats” depending on the exponent. A “fixed-point system” has the radix point always in the same location, something that's useful for, say, accounting software (to represent dollars and cents).

[2]In fact, with overflow or underflow, the error can be arbitrarily large.

[3]1/2.

[4]Did you get $|x \tan(x)|$?

[5]Such as x close to $\pi/2$ radians.

[6]If f is not well-conditioned, then there is no stable algorithm for computing it.

[7]Reflecting the limit, t, on the number of digits.

[8]You'll probably see more about Taylor series in your calculus course. We won't study it here.

[9]The smallest numerator, for a given exponent e, is $x \geq 1.0000 \times 10^e$. Since x is rounded to the nearest value, it differs from x' by, at most, half the increment of the last digit, or $0.0001 \times 10^e/2 = 10^{e-4}/2$. This means that the relative error is, at most, $10^{-4}/2$, or .005%.

[10]x' rounds up x, so $x' = 3.0001$, y' rounds down y, so $y' = 3.0000$, so $x' - y' = 0.0001$, whereas $x - y = 0.0000000001$, so the relative error is

$$\frac{|0.0000000001 - 0.0001|}{|0.0000000001|} = 999999 = 99999900\%$$

[11]In this case x' represents x perfectly, so $x' = 3.0000500$, while y' rounds y up, so $y' = 3.0000500$. Thus, $x' - y' = 0.0000000$, and the relative error is

$$\frac{|0.0000000001|}{|0.0000000001|} = 1 = 100\%$$