# 1 – Describe the sources of computational error

## Number systems

What is a number system? Well, a rough definition (without getting too deeply into math) is a set of values and operators to do calculations with the values. So what kinds of number systems are there, and how do computers represent and work with them, and what sorts of errors are introduced?

### Boolean values

Although we may not think of Boolean values as “numbers”, they have the same properties – in fact, representing **true** as **1** and **false** as **0** is the underlying basis of our binary computers! Boolean values have operators like **AND**, **OR**, and **NOT**. How do computers represent such numbers? We’ve already pointed this out – **true** is **1** and **false** is **0**. We also know computers can perform any of the Boolean operations using gates. Is there any error introduced here? No, there shouldn’t be (subject to random power surges beyond our control), since we are just working with two values, and the Boolean operators always return a Boolean value as well.

### Integers

Let’s recall the sets of numbers that we learned in school:

Natural numbers: **{ 1, 2, 3, … }**

Whole numbers: **{0, 1, 2, 3, …}**

Integers: **{…, -3, -2, -1, 0, 1, 2, 3, …}**

And what are some of the common operations on integers? **+, -, \*, /**.

Can computers represent integers? We’ve already seen in previous classes that they can, using two’s complement notation to represent negative numbers. For instance, in Java, they can be stored in variables of type byte, short, int, or long. Are there any errors in representation? As we’ve seen, there’s the restriction that integers can only be as large as the number of bits that we assign to them, but other than that, they are just represented as 0’s and 1’s.

Are there errors introduced by operations? Again, as we’ve seen when calculating factorials, calculations on two correct numbers can result in an incorrect number if **overflow** occurs. How can we deal with overflow? We can always assign more bits to numbers – that’s basically what the BigInteger class in Java does. The tradeoff is that performance decreases as we allow larger and larger numbers.

We also have to consider what we mean by division – integer division doesn’t introduce any errors, but non-integer division takes us into an entirely different number system – the real numbers.

### Real numbers

What are the real numbers? (Remember, “real” is a math name for this set of numbers, not a distinction between “real” and “fake”!) The real numbers are all of the integers, plus any number in between the integers, including fractions like and , but also numbers such as pi and the square root of 2 that can’t be represented with fractions or decimal places. The easiest way to visualize real numbers is with a number line. Real numbers again have operators like **+, -, \*, /**, and square roots.

The number line

(from <https://en.wikipedia.org/wiki/Number_line>)

How do computers represent real numbers? Java uses float and double, floating-point number systems where we have a “mantissa” and an exponent, similar to scientific notation but in binary.

Recall that in scientific notation, numbers can always be represented with the first part being a series of digits with one digit to the left of the decimal point (the “mantissa”) multiplied by ten (the “base”) raised to some power (the “exponent”). For instance:

345.3 would be written as 3.453 × 102 (where 3.453 is the mantissa and 2 is the exponent)

0.00210 would be written as 2.10 × 10-3 (where 2.10 is the mantissa and -3 is the exponent)

Note that the decimal place moves or “floats” so that the mantissa is always in the same form. For instance, to represent 387000.0, we would move or “float” the decimal place 5 spots to the left, which is why we would use an exponent of 5: 3.870000 × 105.

Floating-point systems use a similar notation, but the mantissa and exponent are stored in binary, and the power used is the power of 2 (and to be precise, they float the binary point over one further so that numbers always start with 0). So for instance, 5.5 = 101.12 × 20 = 0.1011 × 23 would be represented as 0.10112 × 211 base 2. The computer would only store the digits – 1011 for the mantissa and 11 for the exponent (OK, technically there is also one bit stored for the positive or negative value of the mantissa, and exponents are represented with an offset). The first 0 and the base of 2 are assumed and not stored. So the value stored is just mantissa | exponent, with a fixed number of bits used for the mantissa and exponent (a total of 32 bits for a float and 64 bits for a double).

Floating-point number systems have a number of drawbacks:

* They suffer from **overflow** as numbers get too big for the exponent (number can’t be stored in the bits assigned to the exponent) – Java displays such numbers as Infinity
* They suffer from **underflow** as numbers get too small for the exponent (number can’t be stored in the bits assigned to the exponent) – Java displays such numbers as 0.
* They suffer from a lack of precision when numbers have too many digits (number can’t be stored in the bits assigned to the mantissa) – Java rounds such numbers off
* They can’t represent certain numbers – for instance, in decimal, 1/3 = 0.333… and can’t be written precisely using any number of digits. Computer suffer from more problems, since they represent numbers in binary, and numbers such as 1/10 = 0.110 (no error) = 0.0001100110011…2 cannot be represented precisely as well.

(To see this, note that binary numbers use powers of two for their placeholders, so the placeholders to the left of the binary point are 1, 2, 4, 8, etc. and the placeholders to the right of the binary point are , , , etc. One tenth < and and but > , so its first four binary digits are 0.0001, with the remainder between one tenth and making up the rest of the binary digits as we go along, which turn out to be 0.0001100110011…. You can also double any fraction and remove at the integer part to see what the corresponding binary bit would be, so 0.1 becomes 0.2, 0.4, 0.8, 1.6, 1.2, 0.4, 0.8, 1.6, 1.2, etc.)

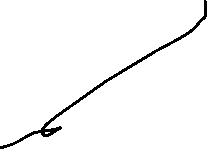
This representation error explains why 0.1 + 0.1 + 0.1 != 0.3 on the computer; each of the 0.1 values is rounded off, and the accumulated round-off error makes the sum different from 0.3.

* Calculations with operators can introduce problems with underflow and overflow (as with factorials), but also with precision. This can happen even when we have two precise numbers to begin with. Rounding errors are commonly how such precision problems are introduced.

How do we overcome such errors? We can round off, so we at least have a controlled amount of error. We can always introduce more digits to work with, but even that can’t deal with numbers like pi. We can switch from binary to decimal representations using solutions such as the BigDecimal class in Java, but we suffer from decreased performance and we still only have a defined number of digits to work with, so precision continues to be a problem. And calculations continue to suffer from the problem of introducing more error (such as 0.1 + 0.1 + 0.1).

*Example:* Suppose you are working with a decimal machine (rather than binary) that stores numbers in the form #.## × 10±# (so three decimal digits are stored for the mantissa and one decimal digit along with a ± sign are stored for the exponent). Calculations can be done with exact precision, but each step of the calculations must be rounded off to three digits after being done. Consider the following:

* The largest possible number in this system is 9.99 × 109 or 9990000000.



* The smallest possible positive number in this system is 0.01 × 10-9 or 0.00000000001.
* 312 can be accurately stored as 3.12 × 102
* 11.3 can be accurately stored as 1.13 × 101
* 312 + 11.3 = 323.3 which would be stored as 3.23 × 102 = 323, introducing error from a calculation involving accurate numbers
* 312 \* 11.3 = 3525.6 which would be stored as 3.53 × 103 = 3530, introducing error from a calculation involving accurate numbers

The best we can do with floating-point systems is accept the round-off errors and deal with them to try to limit the error introduced.

### Other number systems

While there are other mathematical number systems such as complex numbers, imaginary numbers, and rational numbers, they are not as commonly used in computing, so we will not focus on them. From this point on, our focus will be primarily on the real numbers and how we can deal with the errors introduced by the real number system. These problems will also need to be dealt with in future learning outcomes as we try to calculate various different kinds of values.

## Types of error

Some of the types of error that show up in computations include:

Representation error – e.g. limited precision, or numbers like 0.1 that can’t be represented exactly

Measurement error – e.g. temperature (introduced by humans or by limits of equipment) – for instance, a temperature of 18°C could actually be 18.4° or 17.813576° but be reported as 18° due to round-off errors introduced by humans, or by limitations of the thermometer being used to do the measurement

Round-off error – from computer representation, measurement, or calculation; can sometimes be quantified e.g. 42 kg ± 0.5 kg.

Truncation error – when we have a formula with many or infinite terms and we only add some of them. For instance, *e* (Euler’s number) is an irrational number where *e* = 1/0! + 1/1! + 1/2! + …, so sum_(n=0)^â 1/(n!) = e in ***summation notation*** where the Greek capital letter **Σ** (sigma) indicates that we are adding up similar terms, and the n = 0 to ∞ for calculating 1/*n*! is shown by the numbers below and above the sigma. Since we cannot add up infinite terms, we always introduce some truncation error into the calculation.

Try calculating *e* with differing numbers of terms – you can see the truncation error in Java by comparing it to the constant Math.E, which is the best possible floating-point representation of *e*. Note that even the order of calculation can affect things – adding smallest to largest may produce better results (try it with 19 terms).

*Example:* Suppose we are working with the machine described earlier. We wish to add up the numbers 8080 = 8.08 × 103, 4 = 4.00 × 100, and 3 = 3.00 × 100. Adding them up in order gives us 8080 + 4 = 8084 which must be rounded off to 8080 = 8.08 × 103 since the results of any calculation must be stored in the system with only 3 digits. Then we have 8080 + 3 = 8083 rounded off again to 8080 = 8.08 × 103 as our final result. But adding the numbers from smallest to largest gives us 3 + 4 = 7 = 7.00 × 100, and then adding 7 + 8080 = 8087 which rounds off to 8090 = 8.09 × 103.

So why don’t we always worry about error?

* Usually, we don’t need complete precision
* We can usually get more precision if needed
* Roundoff errors may cancel – e.g. 1/3 + 2/3 ≈ .33 + .67 = 1

But why should we worry about error?

* Awareness – reporting e.g. 18°C = 64.4°F should be reported as rounded off to 64°F so that we don’t introduce extra precision; also don’t expect doubles to equal exactly (look for a range)
* Extra precision in calculations can be slow (try 1000000! with the BigInteger class)
* Errors may accumulate – e.g. 1/3 + 1/3 + 1/3 ≈ 33% + 33% + 33% = 99%

## Calculating errors in computation

Since we know that errors in computation exist, we should try to understand what happens when error is introduced into our calculations. So let’s see how error propagates as we do calculations.

Suppose we start with an exact number *x*. It is represented in the computer by some approximation *xa*, which may contain errors for the reasons we’ve already discussed. Let’s designate the amount of error in the representation as *Ex*. Then we can represent this relationship as follows:

exact = approximate + error, so *x* = xa + *Ex*

or equivalently

error = exact - approximate, so *Ex* = *x* - *xa*

*Ex* may be positive or negative (depending on whether the approximation is less than or greater than the exact value). Often we are only interested in the size of the error, so we take the absolute value: | *Ex* |.

While *Ex* is the amount of error, we are often more interested in how the error relates to the whole number, so we can calculate the relative (percentage) error as follows:

Relative error = amount of error / value = *Ex* / *x*

Which is approximately equal to *Ex* / *xa*, since we often don’t know the exact value

And can also be expressed in absolute terms: | *Ex* / *x* |

Similarly, for a variable *y*,

*y* = *ya* + *Ey* and relative error =

*Example:* Suppose we know that *x* = 10.3, but are dealing with the approximation *xa* = 10. Then:

* The error in *x* = 10.3 - 10 = 0.3
* The absolute error | *Ex* | = 0.3 as well
* The relative error is 0.3 / 10.3 ≈ 0.029 or 2.9%

If we also know that *y* = 99.7 but we are using the approximation *ya* = 100, then:

* The error in *y* = 99.7 - 100 = -0.3
* The absolute error | *Ey* | = 0.3 (with no negative sign) – note that this is the same as the absolute error in *x*, but it seems “unfair” to say they are the same when *y* is so much bigger than *x*
* The relative error is -0.3 / 99.7 ≈ -0.003 or -0.3% – note that the relative error does a better job of expressing that the error in *y* is smaller in proportion to its value than the error in *x*

Note that often, we don’t have the exact values; then we can approximate relative error using the approximate values. For instance, if we know a measurement has been rounded to 42 ± 0.5, we can approximate the (maximum) relative error as ±0.5 / 42 ≈ ±0.0119 or ±1.2%.

What happens when we do calculations with values? In particular, how does error grow or propagate?

### Addition

Suppose we have *x* and *y* as above. What happens to the error as we add them?

*x* + *y* = (*xa* + *Ex*) + (*ya* + *Ey*) by substitution

= (*xa* + *ya*) + (*Ex* + *Ey*) by rearranging terms

exact = approximate + error

so the amount of error in addition *Ex*+*y* = *Ex* + *Ey*. Now the error can be either positive or negative, so in the worst case, the error grows, while in the best case, the errors cancel. Consider the following:

A poll says 10 people prefer vanilla ice cream while 20 prefer chocolate. What are the percentages?

10/30 = 1/3 ≈ approximately 0.33 or 33% with some rounding error  
20/30 = 2/3 ≈ approximately 0.67 or 67% with some rounding error  
33% + 67% = 100% of poll respondents – exactly right since the errors cancel out

But another poll says 10 people prefer cones, 10 prefer bowls, and 10 are undecided.

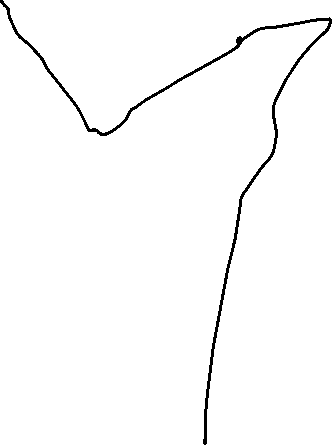
10/30 ≈ 33% for each category, so 33% + 33% + 33% = 99%; error has increased

How does this affect relative error?

Relative error = error / value = (*Ex* + *Ey*) / (*x* + *y*)

Let’s assume for now that *x* and *y* are both positive numbers. (When we consider subtraction, it will handle the case when one of them is negative, since subtraction is just adding a negative number). Then while the error may grow, the value will also grow, so the relative error remains fairly constant.

*Example:* Suppose *x* = 101 and *xa* = 100, and *y* = 11 and *ya* = 10. Then *Ex* = 1 and *Ey* = 1. Thus the error in addition is *Ex*+*y* = *Ex* + *Ey* = 1 + 1 = 2. Since we have the exact values, we could also simply compare the exact answer (*x* + *y* = 101 + 11 = 112) and the approximate answer (*xa* + *ya* = 100 + 10 = 110) to determine the error (*Ex* + *Ey* = exact - approximate = 112 - 110 = 2). What about the relative error?



relative error in *x* = 1 / 101 ≈ 0.0099 ≈ 1%  
 relative error in *y* = 1 / 11 ≈ 0.0909 ≈ 9%

relative error in *x* + *y* = error / exact = 2 / 112 ≈ 0.017857 ≈ 1.8%

We see that the while the absolute error in *x* + *y* grew, the relative error in *x* + *y* is in between the relative errors in *x* and *y* (in this case, closer to the relative error in *x* since *x* is a bigger number).

We may not know the exact values, but we can use the best approximations to approximate the relative error in *x* and *y*.

*Example:* Suppose that we don’t know the exact values for *x* and *y*, but we do know the approximate values and boundaries for the error: *x* = 40 ± 1 and *y* = 50 ± 1. To calculate the relative error:  
 relative error in *x* ≈ ±1 / 40 = ±0.025 = ±2.5%  
 relative error in *y* ≈ ±1 / 50 = ±0.02 = ±2%

What happens when we add *x* and *y*? Using the best approximations for addition, *x* + *y* = 40 + 50 = 90. We know that the error in addition is *Ex*+*y* = *Ex* + *Ey* = (±1) + (±1). Although the pluses and minuses could cancel out, we are looking for the **maximum** possible error, so we will add the errors: *Ex*+*y* = (±2). You could also determine this by looking at the range of values for *x* and *y*:  
 x = 39…41  
 y = 49…51

*x* + *y* = 88…92 = 90 ± 2

Note that the amount of error has increased; but what happens to the relative error?  
 relative error in *x* + *y* = error / exact ≈ ±2 / 90 = ±0.0222… ≈ ±2.2%

The relative error in addition does not grow, but remains similar to the relative error of initial values (a weighted average of the initial relative errors, in this case slightly closer to the relative error in *y* because *y* is greater than *x*).

### Subtraction

Again, let’s suppose that *x* and *y* are positive numbers, represented by some approximations *xa* and *ya*. Then:

*x* - *y* = (*xa* + *Ex*) - (*ya* + *Ey*) by substitution

= (*xa* - *ya*) + (*Ex* - *Ey*) by rearranging terms

exact = approximate + error

So the error in subtraction *Ex*-*y* = *Ex* - *Ey*.

If both *x* and *y* are positive, then we are guaranteed that |*x - y*| is smaller than at least one of the terms. Note that because the errors can be either positive or negative, we are not guaranteed that the resulting error (*Ex* - *Ey*) will be smaller. Thus the relative error can increase greatly with subtraction, since we are subtracting a lot of the “correct part” and leaving lots of “error part”. Consider the following examples:

Example 1 – error cancels out, so both the amount of error and the relative error shrink:



Example 2 - error does not cancel out, so both the amount of error and relative error grow:



*Example:* Suppose that we don’t know the exact values for *x* and *y*, but we do know the approximate values and boundaries for the error: *x* = 50 ± 1 and *y* = 40 ± 1. To calculate the relative error:  
 relative error in *x* ≈ ±1 / 50 = ±0.02 = ±2%  
 relative error in *y* ≈ ±1 / 40 = ±0.025 = ±2.5%

What happens when we subtract *y* from *x*? Using the best approximations, *x* - *y* = 50 + 40 = 10. We know that the error in subtraction is *Ex*-*y* = *Ex* - *Ey* = (±1) - (±1). This may look like it cancels out (and it possibly could), but we are looking for the **maximum** possible error, which would occur when we have +1 - -1 = 2 or -1 - +1 = -2, so we will add the errors: *Ex*-*y* = (±2). You could also determine this by looking at the range of values for *x* and *y*, but be careful about the arrangement of the values:  
 x = 49…51  
 y = 39…41

*x* - *y* = 10…10 = 10 ± 0 makes it look like the error cancelled out, but

x = 49…51  
 y = 41…39

*x* - *y* = 8…12 = 10 ± 2

Note that the amount of error has increased; but what happens to the relative error?  
 relative error in *x* - *y* = error / exact ≈ ±2 / 10 = ±0.2 = ±20%

The relative error in subtractions grows and can be much bigger than the relative errors in *x* and *y*.

### Multiplication

Again, let’s suppose that *x* and *y* are positive numbers, represented by some approximations *xa* and *ya*. Then:

*x* \* *y* = (*xa* + *Ex*) \* (*ya* + *Ey*)

= *xaya* + (*xaEy* + *yaEx* + *ExEy*)

exact = approximate + error

So the error in multiplication is *Exy* = (*xaEy* + *yaEx* + *ExEy*). Again, this might grow or shrink depending on the amounts and signs of the values and the errors, since the errors may partially cancel out. Consider:

 Error = 11\*1 + 19\*(-1) + 1\*(-1) partially cancels

while

 Error = 9\*1 + 19\*1 + 1\*1 has no cancellation

How does the relative error behave? Consider that *ExEy* should be relatively small compared to *xy*, since the starting errors must be relatively small for our calculations to have any value at all. So *ExEy* can be ignored. Further consider that *xy* ≈ *xaya*. Thus, the relative error can be approximated as follows:

Relative error =

So the relative error is approximately the relative error in *x* + relative error in *y*. So relative error tends to grow with multiplication (unless there is cancellation occurring).



This applies when we don’t know the exact values as well:

*Example:* Suppose we have xa = 100 ± 1 (relative error of 1%) and ya = 10 ± 1 (relative error of 10%). Then our best answer for the product of *x* and *y* is *xaya* = 100 \* 10 = 1000. The amount of error is given by the formula (*xaEy* + *yaEx* + *ExEy*) = 100\*(±1) + 10\*(±1) + (±1)\*(±1) = ±100 + ±10 + ±1 = ±111, and the relative error is approximately ±111 / 1000 = 11.1%. Note that the relative error is roughly the relative error in *x* plus the relative error in *y* (1% + 10% = 11%).

This can also be seen using ranges of values:

*x* = 99…101  
 *y* = 9…11

*xy* = 99\*9…101\*11 = 891…1111 which is roughly 1000 ± 111

Since relative error can grow relatively quickly as we do calculations, we should try to minimize the number of calculations that we do, especially calculations such as subtraction and multiplication.