Math 340 • Fall 2026 • NJIT • Victor Matveev

Binary system, round-off errors, loss of significance, and the "double precision" IEEE standard.

0. Why understanding representation of real numbers in a computer is important

Representation of numbers by a digital computer may sound like an uninteresting technical detail, but without knowing such details, you cannot be certain that even simple calculations you attempt to perform the task you expect. As the simplest example, consider the following calculation in MATLAB (or any other coding environment):

X = 0.1 + 0.1 + 0.1 – 0.3

MATLAB will return **X = 5.551115123125783e-17**, not zero! However, if you try

Y = 0.5 + 0.5 + 0.5 – 1.5

MATLAB will return **Y = 0**, as expected! Same goes for **0.125 + 0.125 + 0.125 – 0.375**.

You may think that the first answer above is just some arbitrary numerical "noise", but that's not the case. If you take the base-2 logarithm of the X=5.551115123125783e-17 value above (using MATLAB's built-in log2 function), you will find $\mathbf{\log_2(X) = -54}$.

If you find all this surprising, this means you do not understand how numbers are represented in a digital computer; fortunately, the next few pages will explain everything you need to know.

1. Bits and the binary number system

A "bit" is one digit in a binary representation of a number, and can assume only one of two possible values: **0** or **1**. This simplicity is precisely the reason why the binary number system is more "computer-friendly" than the usual decimal system: in order to store a binary number, it is sufficient to "string together" primitive electric circuits that have only two stable states; one of those states we can call a **0**, and the other state we can call a **1**. To store a decimal digit without converting to binary, we would have to use electric circuits with elements that have 10 possible stable states (each state representing a digit from 0 to 9); such a circuit would be extremely complicated to construct, unstable, and completely unnecessary, since binary algebra is easier than decimal algebra.

The binary (base-2) number system is very similar to the usual decimal (base-10) system. It is a **positional** system, which means that the value of each digit comprising the number depends on its position with respect to the fraction point. Namely, in a positional system each digit is multiplied by the system base (10 in the case of decimal, 2 in the case of binary), raised to the power determined by the digit's position. For instance, you will probably find the following decomposition of number 145.625 to be obvious:

$$\mathbf{145.625} = \mathbf{1}\times 10^{2} + \mathbf{4}\times 10^{1} + \mathbf{5}\times 10^{0} + \mathbf{6}\times 10^{-1} + \mathbf{2}\times 10^{-2} + \mathbf{5}\times 10^{-3}$$
2 1 0 -1-2-3

The little number under each digit indicates its position with respect to the fraction point. Now, here is what this number looks like in the binary system:

10010001.101(binary) = $2^{7}+2^{4}+2^{0}+2^{-1}+2^{-3}$ = 128 + 16 + 1 + 0.5 + 0.125 = **145.625** decimal
7 6 5 4 3 2 1 0 -1-2-3

As this calculation shows, converting from binary to decimal is simple since each binary digit is either **0** or **1**. We don't have to worry about zeros at all, and it is sufficient to collect powers of two corresponding to each **1**, with the power equal to the position of this **1** with respect to the fraction point.

Now, converting from decimal to binary is only slightly trickier: we have to figure out the highest power of 2 that "fits" within our number, which will give us the position of the highest, most significant "**1**" (the left-most non-zero digit). In the example above, such highest power contained in 145.625 is $128=2^{7}$, so there will be a 1 in position 7. Now, subtract 2^{7} from the original number, and find the highest power of 2 that fits in the remainder, 17.625. This will be $16=2^{4}$, so the next **1** is in position 4. Take the remainder, 1.625, and repeat until you are done: $\mathbf{145.625_{10} = 10010001.101_{2}}$

[Aside: for integer numbers, it is easier to convert from decimal to binary by finding the lowest digit first, instead of the highest one. The lowest digit is equal to the remainder after division by 2]

2. Exponential notation in binary system.

According to the **double precision** standard, **64 bits** are used to represent a real number in computer memory. However, only **52** of those bits are devoted to representing the significant digits, or **mantissa** of the number (also called the **significand**), while another **11** digits are devoted to the "**exponent**". The remaining one digit determines the sign of the number (it is negative if sign bit equals **1**, and positive otherwise). Now, the "exponent" refers to the standardized exponential scientific notation for real numbers:

$$\mathbf{145.625 = 1.45625\times 10^{2} = 1.45625e{+}002}$$

In this example, the decimal exponent equals two. Re-writing a number in exponential notation is called "normalizing". In the binary system, each number can be "normalized" in the same way, by carrying the fraction point to the left or to the right. However, for a binary number, each time we move the fraction point by one bit, we pick up a factor of 2, not a factor of 10. For instance, you can easily check that the following is true, by converting both sides of this equation to decimal:

10010001.101(binary) = $\mathbf{1.0010001101}\times 2^{7}$ (fraction point moved by 7 bits).
7 6 5 4 3 2 1 0 -1-2-3

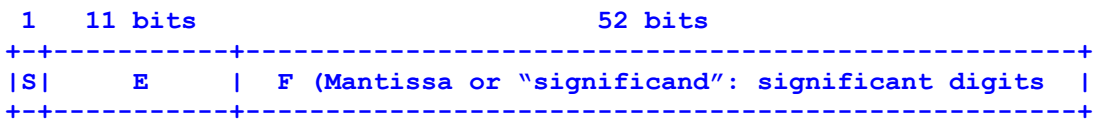
There is a definite purpose of normalizing a number before storing it in the computer: it is easy to see that this way we can store very large or very small numbers, by devoting a certain number of bits (binary digits) to the exponent. Note also that after you normalize a binary number, there will always be a **1** before the fraction point (since **1** is the only non-zero binary digit); therefore, we don't have to store this **1** – we can simply "pretend" that it's always there! The only exception is storing zero – we will discuss this below.

3. Double precision: a summary

To sum up, here is a schematic explaining the double precision number representation:

$$\textbf{Number} = (-1)^{S} \times 1.F \times 2^{E-1023}$$

Where **S** is the **1**-bit sign, **F** is a **52**-bit long binary mantissa (significand), and **E** is **11** bit-long exponent, which altogether gives **64** bits. In order to represent negative exponent values, 1023 is subtracted from **E** (in principle, one could think of reserving one bit of E for the sign, but this subtraction rule allows to define special values of E – see below)



Note that the ordering of the bits may be machine-specific: left-to-right, or right-to-left

Example: consider the double-precision representation of number 145.625 that we already converted to binary above:

$$\textbf{145.625}_{10} = \textbf{10010001.101}_{2} = \textbf{1.0010001101} \times 2^{7} = \textbf{1.0010001101} \times 2^{1030-1023}$$

The exponent in this case is $\textbf{E} = \textbf{7} + \textbf{1023} = \textbf{1030}_{10} = \textbf{10000000110}_{2}$, so we obtain the following 64-bit double-precision representation:

0 10000000110 0010001101000000000000000000000000000000000000000000000000

There are two exceptions to this representation, corresponding to the case when E contains all zeros or all ones:

1) When **E=00000000000** (all zeros), we assume that the leading binary digit is zero, not 1, and the exponent is -1022 (not -1023 as we would assume given the standard rule above)

$$\textbf{Number} = (-1)^{S} \times 0.F \times 2^{-1022}$$

Such numbers are technically termed "subnormal".

This allows us to store zero, as well as tiny non-zero numbers: the smallest non-zero positive number would have a mantissa with all zeros except for 1 in the 52nd position:

| Smallest non-zero number | = $\mathbf{2^{-52} \times 2^{-1022} = 2^{-1074} = 4.94e\text{-}324}$ (tiny!)

This corresponds to the following double-precision bit sequence:

0 00000000000 0000000000000000000000000000000000000000000000000001

2) When E=11111111111 (all ones), the number is assumed to be Inf (infinity) if mantissa is zero or NaN (not a number), if the mantissa is non-zero, allowing the computer to continue calculations that in the old days would simply crash the code.

4. Double precision: limitations

Now let us examine the limitations arising from such 64-bit representation. We already figured out the smallest (in absolute value) non-zero number that can be represented is about **4.94e-324**. Now let's think of the largest finite number. The largest 11-bit exponent is $\mathbf{2^{11}-1 = 2047}$ (for the same reason that the the largest n-digit *decimal* number is $\mathbf{10^n-1}$). However, if all bits equal 1, the number is **Inf** or **NaN**, so we have to subtract 1, which gives **E=2046**. The largest mantissa contains all 1's, which gives a value close to 2 (for the same reason that **9.999..9** is approximately **10** in the decimal system). Therefore, the largest number is approximately

| Largest number | $\approx \mathbf{2*2^{2046-1023} = 2^{1024} = 1.797e308}$ **(huge!)**

This corresponds to the following 64-bit double-precision bit sequence:

0 11111111110 1111111111111111111111111111111111111111111111111111

This shows that double precision representation is very powerful, at least for representing tiny and huge numbers: for comparison, the diameter of our Universe is about 10^{28}-10^{29} cm, so we rarely encounter numbers bigger or smaller than the above limits!

The real limitation of the double precision standard arises from its accuracy in representing real numbers (the mantissa part, **1.F**). Let's see what the value of the lowest (least significant) 52nd digit in **F** is: $\mathbf{2^{-52}} = \mathbf{2.2 \cdot 10^{-16}}$. Thus, we have about **16** decimal digits of accuracy to work with. Another way to get the number of significant *decimal* digits in double accuracy is to simply divide **53**, the number of bits, by $\mathbf{\log_2(10)=3.322}$, which yields about **16** for the corresponding number of ***significant decimal digits***.

5. Round-off errors.

For the sake of simplicity, we will ignore the binary nature of all computer numbers for a moment, and concentrate on the almost equivalent limitations arising from using **16 digits** to represent decimal numbers. Consider for instance the following **20-digit** integer:

98765432109876543210

Now, this is an example of a number that cannot be stored precisely. The large size of the number is not a problem – the problem is the number of significant digits. Since we can only keep 16, we have to "round off" the remaining 4:

98765432109876540000

It is easy to see that the **round-off error** equals **3210** in our example. You can check that **MATLAB** cannot tell the difference between these two numbers: try calculating their difference.

When will this be a problem? If we denote the above number by **N**, you can easily see what the result of the following calculation would be:

(N + 1234) - N

We would want to get **1234** as our result, but we will naturally get **zero** instead, since the last four digits are rounded off:

98765432109876543210 + 1234 = 9876543210987654**4444**

Rounds off to: 9876543210987654**0000**

Naturally, the size of the round-off error changes with the size of the number - in other words, if **N=9.8765432109876543210e-109**, then the round-off error will be **3.210e-125**. Using these two examples, you can now understand the following upper bound on the round-off error:

$$|\,\textbf{Error of } N\,| \;\leq\; 10^{-16} \cdot |\,N\,|$$

The expression on the right is simply a rough estimate of the 17th digit of N, the one that is thrown out due to round-off. Equivalently, we can write:

$$\frac{|\,\textbf{Error of } N\,|}{|N|} \;\leq\; 10^{-16}$$

Therefore, expression of form $A + a$ is rounded off to simply A if (approximately)

$$\left|\frac{a}{A}\right| \;\leq\; 10^{-16}$$

Loss of significance: an example.

Let us examine for instance how the round-off errors may affect a function evaluation, such as $\mathbf{sin}(\boldsymbol{x}+\mathbf{2}\boldsymbol{\pi n})$. We know that the result should equal $\mathbf{sin}(\boldsymbol{x})$.
However, if $\mathbf{|}\boldsymbol{x}\mathbf{|}$ is less than the round-off error of $\mathbf{2}\boldsymbol{\pi n}$, there is no way that **MATLAB** (or any other computer program) can recover the value of $\boldsymbol{x}$ from the sum $\boldsymbol{x}+\mathbf{2}\boldsymbol{\pi n}$. In other words, all digits of $\boldsymbol{x}$ will be rounded off when x is added to $\mathbf{2}\boldsymbol{\pi n}$. This is a classic example of **complete loss of significance of $\boldsymbol{x}$.** Let's find when this will happen, given $\boldsymbol{x}$**=1:**

$$\frac{\mathbf{Error}(\mathbf{2}\boldsymbol{\pi n})}{\mathbf{2}\boldsymbol{\pi n}} \approx \frac{\mathbf{1}}{\mathbf{2}\boldsymbol{\pi n}} \approx \mathbf{10}^{-\mathbf{16}} \;\rightarrow\; \boldsymbol{n} \approx \frac{\mathbf{10}^{\mathbf{16}}}{\mathbf{2}\boldsymbol{\pi}} \approx \mathbf{2}\cdot\mathbf{10}^{\mathbf{15}}$$

Let us verify this result using MATLAB: $\mathbf{sin}(\mathbf{2}\boldsymbol{\pi} * \mathbf{2}\boldsymbol{e}\mathbf{15}+\mathbf{1}) - \mathbf{sin}\,\mathbf{1} = -\mathbf{1.657}$, which clearly represents complete failure, since this error is larger (in absolute magnitude) than **sin(1) = 0.841**.

For lower $\boldsymbol{n} = \mathbf{10}^{\mathbf{14}}$, Matlab gives $\mathbf{sin}(\mathbf{2}\boldsymbol{\pi} * \mathbf{1}\boldsymbol{e}\mathbf{14}+\mathbf{1}) - \mathbf{sin}(\mathbf{1}) = -\mathbf{0.012}$, which is only borderline "tolerable".

6. Numbers that are represented exactly, with no round-off errors.

From the earlier discussion of the double precision standard it follows that any binary numbers that can "fit" entirely into the 53 bits allocated for storing its fractional part (which includes the "implied" highest bit**)** are represented exactly, with no loss of accuracy. Therefore, a number which is given by a sum of powers of two is represented without round-off errors, provided that the difference between the highest and lowest powers is less or equal 52. This includes all integers which are smaller than $2^{53}-1 \approx 9 * 10^{15}$, since such integers can be represented as a binary number with 53 bits or less ($2^{53}-1$ is a binary number with all 53 bits equal to 1: 1111111111…..1111).

Apart from integers, many fractional numbers are also represented exactly, for instance:

0.5 (decimal) = 2^{-1} = 0.1 (binary)
0.75 (decimal) = $2^{-1}+2^{-2}$ = 0.11 (binary)
0.625 (decimal) = $2^{-1}+2^{-3}$ = 0.101 (binary)

Surprisingly however, some simple decimal numbers which can be exactly represented using only *one decimal* digit, require an *infinite* number of *binary* bits to be represented without error! For instance, you can easily check the validity of the following binary representation of 1/10=0.1:

0.1 (decimal) = 0.00011001100110011…(binary) = $2^{-4}+2^{-5}+2^{-8}+2^{-9}+2^{-12}+2^{-13}+\ldots$

The explanation for this fact is very simple: since $\mathbf{0.1} = \frac{\mathbf{1}}{\mathbf{2 \cdot 5}}$, it can never be represented as a sum of fractions that have only factors of 2 in the denominator (because 5 is not divisible by 2).

As mentioned in the very beginning of this lecture, this has serious consequences for calculations involving decimal fractions like **0.1** (as we saw when evaluating **0.1 + 0.1 + 0.1 – 0.3**).

7. Things to avoid.

As was shown, the round-off errors are usually small (1 part in 10^{16}). Now, apart from the sine calculation example above, when should we in general worry about this error? The answer is simple to formulate: as with any other type of error, **we should worry about the round-off error when it is comparable in absolute value to the result of a calculation that we seek.** In other words, it is not the absolute value of the error which is important, but the ***relative*** error, which is the ratio of the error to the true result value. For instance, here are several such examples where the error is comparable to the result, resulting in a failure when evaluated numerically using Matlab or some other program:

- $(1 + x^8) - 1 = x^8$ (fails when $x \le 0.01$)
- $(10-9)^{14} = \sum_{k=0}^{14} \frac{14!}{k!(14-k)!} 10^k (-9)^{14-k} = 1$ (summation will fail)
- $\cos(100) = \sum_{n=0}^{\infty} (-1)^n \frac{100^{2n}}{(2n)!} \approx 0.8623$ (truncated sum will fail)
- $\frac{x^2}{2}\left(\frac{1}{x-1} - \frac{1}{x+1}\right) = \frac{x^2}{x^2-1}$ (equality fails when $x \ge 10^{16}$)
- $\sqrt{x^8+1} - \sqrt{x^8-1}$ (calculation fails when $x \ge 100$)

You may not notice at first, but there is something in common between all five examples. Namely, **they all involve precise cancellation between terms to achieve accurate results** (notice that there is a minus sign in each of these formulas). While the error in calculating each of the terms in a given expression may be small compared to the size of that term (1 part in 10^{16}), it is large when compared to the result, since the result is much smaller than each term taken separately.

Let's take for instance the last example. If our calculation only involves $\sqrt{\boldsymbol{x+1}}$, there is no problem: even though the $\mathbf{1}$ will get completely rounded off when $\boldsymbol{x \ge 10^{16}}$, why should we care? The result will be accurate enough; the error relative to the answer will be small (1 part in 10^{16}, as usual). However, the value of expression $\sqrt{\boldsymbol{x+1}} - \sqrt{\boldsymbol{x+1}}$ is much smaller than each of the two terms taken separately, and when $\boldsymbol{x \ge 10^{16}}$, yields a

zero, which is incorrect. In fact, the exact limiting behavior of this difference is $1/\sqrt{x}$. So the result decreases as *x* grows, while the round-off error increases; it is easy to see that eventually there will be a problem.

The conclusion is simple: whenever possible, **avoid subtracting numbers that are very close in magnitude!** For instance, the last example may be re-written by multiplying by a conjugate the numerator and the denominator:

$$\sqrt{x+1}-\sqrt{x-1}=\left(\sqrt{x+1}-\sqrt{x-1}\right)\frac{\sqrt{x+1}+\sqrt{x-1}}{\sqrt{x+1}+\sqrt{x-1}}=\frac{(x+1)-(x-1)}{\sqrt{x+1}+\sqrt{x-1}}=\frac{2}{\sqrt{x+1}+\sqrt{x-1}}$$

Although these two formulas are equivalent, the newly derived one is much more accurate when evaluated numerically. For instance, it is easy to see that it gives the correct limiting behavior, $1/\sqrt{x}$, as $x \to +\infty$. Use MATLAB to compare these two formulas, for values of *x* which are greater or equal $\mathbf{10^{16}}$.

8. Rule of Thumb

When determining at which point an expression fails completely, you only have to examine the original expression, and see if it may return **zero** or **Inf** or **NaN** as a result of the following general roundoff:

$$\boldsymbol{A \pm a \to A}$$

which happens when

$$\left|\frac{\boldsymbol{a}}{\boldsymbol{A}}\right| \leq \mathbf{10^{-16}}$$

Example. For instance, consider the following expression, with $\boldsymbol{x < 1}$:

$$\boldsymbol{f(x)=\frac{1}{100+x^8}-\frac{1}{100-x^8}}$$

It will fail (produce zero, which has no significant digits) when the following roundoff happens:

$$\mathbf{100 \pm x^8 \to 100}$$

According to above, this happens when

$$\left|\frac{\boldsymbol{a}}{\boldsymbol{A}}\right|=\frac{x^8}{100} \leq \mathbf{10^{-16}}$$

which is equivalent to

$$\boldsymbol{x^8 \leq 10^{-14}}$$

Therefore, the problem occurs when $\boldsymbol{|x| \leq 0.01}$.

OPTIONAL: more precise round-off threshold

Given binary number representation of floating-point numbers, round-off thresholds can be described more accurately, as follows:

Case 1: A is not a power of 2:

Roudoff

$$\boldsymbol{A \pm a \to A}$$

happens when

$$\left|\frac{\boldsymbol{a}}{\boldsymbol{\overline{A}}}\right| \leq \boldsymbol{2^{-53} \approx 1.11 \cdot 10^{-16}}$$

The denominator in above expression, $\boldsymbol{\overline{A}}$, is **the largest power of 2 contained in** *A* (determined by the leading binary digit in *A*), that is:

$$\boldsymbol{\overline{A} = 2^{\mathrm{floor}(\log_2 A)}}$$

Example. The roundoff in $\boldsymbol{100 \pm x^8 \to 100}$ happens when (noting that $\overline{100} = 64$)

$$\boldsymbol{\frac{x^8}{64} \leq 2^{-53}}$$

Therefore, the problematic range for the above expression is

$$\boldsymbol{x^8 \leq 2^{-47} \;\; \therefore \;\; |x| \leq 2^{-\frac{47}{8}} \approx 0.017}$$

Case 2: A is a power of 2

However, when A is exactly a power of 2, the threshold for **subtraction** from *A* is different, since the leading digit will become zero after subtracting even a tiny number:

$$\boldsymbol{A - a \to A}$$

happens when

$$\left|\frac{\boldsymbol{a}}{\boldsymbol{A}}\right| \leq \boldsymbol{2^{-54} \approx 5.55 \cdot 10^{-17}}$$

Example. For instance, the roundoff in $\boldsymbol{64 - x^8 \to 64}$ happens when

$$\boldsymbol{\frac{x^8}{64} \leq 2^{-54}}$$

Therefore, the problematic range for this expression is slightly narrower:

$$\boldsymbol{x^8 \leq 2^{-48} \;\; \therefore \;\; |x| \leq 2^{-6} = 0.015625}$$