Note that certain numbers are finite (terminating) decimals, actually are periodic in binary, e.g.

\[ 0.4_{(10)} = 0.01100110011\ldots_{(2)} = 0.00110011_{(2)} \]

**Machine numbers** (a.k.a. binary floating point numbers)

The numbers stored on the computer are, essentially, “binary numbers” in scientific notation \( x = \pm a \times 2^b \). Here, \( a \) is called the **mantissa** and \( b \) the **exponent**. We also follow the convention that \( 1 \leq a < 2 \); the idea is that, for any number \( x \), we can always divide it by an appropriate power of 2, such that the result will be within \([1, 2)\). For example:

\[ x = 5_{(10)} = 1.25_{(10)} \times 2^2 = 1.01_{(2)} \times 2^2 \]

Thus, a machine number is stored as:

\[ x = \pm1.a_1a_2\cdots a_{k-1}a_k \times 2^b \]

- **In single precision** we store \( k = 23 \) binary digits, and the exponent \( b \) ranges between \(-126 \leq b \leq 127\). The largest number we can thus represent is \((2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}\).

- **In double precision** we store \( k = 52 \) binary digits, and the exponent \( b \) ranges between \(-1022 \leq b \leq 1023\). The largest number we can thus represent is \((2 - 2^{-52}) \times 2^{1023} \approx 1.8 \times 10^{308}\).

In other words, single precision provides 23 binary significant digits; in order to translate it to familiar decimal terms we note that \( 2^{10} \approx 10^3 \), thus 10 binary significant digits are roughly equivalent to 3 decimal significant digits. Using this, we can say that single precision provides approximately 7 decimal significant digits, while double precision offers slightly more than 15.

**Absolute and relative error**

All computations on a computer are approximate by nature, due to the limited precision on the computer. As a consequence we have to tolerate some amount of **error** in our computation. Actually, the limited machine precision is only one source of error – other factors may further compromise the accuracy of our computation.
(in later lectures we will discuss modeling, truncation, measurement and roundoff errors). At any rate, in order to better understand errors in computation, we define two error measures: The absolute, and the relative error. For both definitions, we denote by $q$ the exact (analytic) quantity that we expect out of a given computation, and by $\hat{q}$ the (likely compromised) value actually generated by the computer.

**Absolute error** is defined as $e = |q - \hat{q}|$. This is useful when we want to frame the result within a certain interval, since $e \leq \delta$ implies $q \in [\hat{q} - \delta, \hat{q} + \delta]$.

**Relative error** is defined as $e = |q - \hat{q}|/|q|$. The result may be expressed as a percentile and is useful when we want to assess the error relative to the value of the exact quantity. For example, an absolute value of $10^{-3}$ may be insignificant when the intended value $q$ is in the order of $10^6$, but would be very severe if $q \approx 10^{-2}$.

**Rounding, truncation and machine $\epsilon$ (epsilon)**

When storing a number on the computer, if the number happens to contain more digits than it is possible to represent via a machine number, an approximation is made via rounding or truncation. When using truncated results, the machine number is constructed by simply discarding significant digits that cannot be stored; rounding approximates a quantity with the closest machine-precision number. For example, when approximating $\pi = 3.14159265\ldots$ to 5 decimal significant digits, truncation would give $\pi \approx 3.15159$ while the rounded result would be $\pi \approx 3.1516$. Rounding and truncation are similarly defined for binary numbers, for example $x = 0.1011011101110_{(2)}\ldots$ would be approximated to 5 binary significant digits as $x \approx 0.1011_{(2)}$ using truncation, and $x \approx 0.10111_{(2)}$ when rounded.

A concept that is useful in quantifying the error caused by rounding or truncation is the notion of the machine $\epsilon$ (epsilon). There are a number of (slightly different) definitions in the literature, depending on whether truncation or rounding is used, specific rounding rules, etc. Here, we will define the machine $\epsilon$ as the smallest positive machine number, such that

$$1 + \epsilon \neq 1 \text{ (on the computer)}$$

Why isn’t the above inequality always true, for any $\epsilon > 0$? The reason is that, when subject to the computer precision limitations, some numbers are “too small” to affect the result of an operation, e.g.

$$1 = 1.000\cdots000_{(2)} \times 2^{23} \text{ digits}$$

$$2^{-25} = 0.000\cdots000 01_{(2)} \times 2^{23} \text{ digits}$$
When rounding (or truncating) the last number to 23 binary significant digits corresponding to single precision, the result would be exactly the same as the representation of the number \( x = 1 \). Thus, on the computer we have, in fact, \( 1 + 2^{-25} = 1 \), and consequently \( 2^{-25} \) is smaller than the machine epsilon. We can see that the smallest positive number that would actually achieve \( 1 + \epsilon \neq 1 \) with single precision machine numbers is \( \epsilon = 2^{-24} \) (and we are even relying a “round upwards” convention for tie breaking to come up with a value this small), which will be called the machine \( \epsilon \) in this case. For double precision the machine \( \epsilon \) is \( 2^{-53} \).

The significance of the machine \( \epsilon \) is that it provides an upper bound for the relative error of representing any number to the precision available on the computer; thus, if \( q > 0 \) is the intended numerical quantity, and \( \hat{q} \) is the closest machine-precision approximation, then

\[
(1 - \epsilon)q \leq \hat{q} \leq (1 + \epsilon)q
\]

where \( \epsilon \) is the machine epsilon for the degree of precision used; a similar expression holds for \( q < 0 \).
We turn our attention to the first major focus topic of our class: techniques for solving nonlinear equations. In an earlier lecture, we actually addressed one common nonlinear equation, the quadratic equation \( ax^2 + bx + c = 0 \), and discussed the potential hazards of using the seemingly straightforward quadratic solution formula. We will start our discussion with an even simpler nonlinear equation:

\[
x^2 - a = 0, \quad a > 0
\]

The solution is obvious, \( x = \pm \sqrt{a} \) ( presuming, of course, that we have a subroutine at our disposal that computes square roots). Let us, however, consider a different approach:

- Start with \( x_0 = \text{<initial guess}> \)
- Iterate the sequence

\[
x_{k+1} = \frac{x_k^2 + a}{2x_k}
\]

We can show (and we will, via examples) that this method is quite effective at generating remarkably good approximations of \( \sqrt{a} \) after just a few iterations. Let us, however, attempt to analyze this process from a theoretical standpoint:

If we assume that the sequence \( x_0, x_1, x_2, \ldots \) defined by this method has a limit, how does that limit relate to the problem at hand? Assume \( \lim x_k = A \). Then, taking limits on equation (1) we get

\[
\lim x_{k+1} = \lim \frac{x_k^2 + a}{2x_k} \Rightarrow A = \frac{A^2 + a}{2A} \Rightarrow 2A^2 = A^2 + a \Rightarrow A^2 = a \Rightarrow A = \pm \sqrt{a}
\]

Thus, if the iteration converges, the limit is the solution of the nonlinear equation \( x^2 - a = 0 \). The second question is whether it may be possible to guarantee that the described iteration will converge. For this, we manipulate (1) as follows

\[
x_{k+1} = \frac{x_k^2 + a}{2x_k} \Rightarrow x_{k+1} - \sqrt{a} = \frac{x_k^2 + a}{2x_k} - \sqrt{a} = \frac{x_k^2 - 2x_k\sqrt{a} + a}{2x_k} = \frac{[x_k - \sqrt{a}]^2}{2x_k}
\]

If we denote by \( e_k = x_k - \sqrt{a} \) the error (or discrepancy) from the exact solution of the approximate value \( x_k \), the previous equation reads

\[
e_{k+1} = \frac{1}{2x_k}e_k^2 = \frac{1}{2} \frac{e_k^2}{e_k + \sqrt{a}}
\]

For example, if we were approximating the square root of \( a = 2 \), and at some point we had \( e_k = 10^{-3} \), the previous equation would suggest that \( e_{k+1} < 10^{-6} \). One