CS/MATH 3414 — Numerical Methods

Lecture 1
Real Numbers on a Finite Machine

Virginia Tech Department of Computer Science, Fall 2026
Instructor: Sumeet Khatri
Date: August 27, 2026

Learning Objectives

- State and explain the definition of a floating-point number.
- State and explain the IEEE 754 floating-point number formats.
- Explain overflow/underflow, rounding, and unit roundoff.
- Use numerical experiments to reveal finite-precision effects.

Primary References

- Burden and Faires, Section 1.2.
- Cheney and Kincaid, Sections 1.1, 1.3, 1.4.
- Digital Library of Mathematical Functions, NIST, Section 3.1: Arithmetics and Error Measures.

Note: much of the text here comes directly from, or is a summarized form of, the text in the above two references.

1 Numerical computation is a model of real arithmetic

The arithmetic performed by a calculator or computer is different form the arithmetic in algebra and calculus courses. You would likely expect that we always have as true statements things such as $2+2=4$, $4\times 8=32$, and $(\sqrt{3})^2=3$. However, with *computer* arithmetic, we may expect exact results for $2+2=4$ and $4\times 8=32$, but we will *not* have precisely $(\sqrt{3})^2=3$. To understand why this is true, we must explore the world of finite-digit arithmetic.

In our traditional mathematical world, we permit numbers with an infinte number of digits. The arithmetic we use in this world *defines* $\sqrt{3}$ as that unique positive number that, when multiplied by itself, produces the integer 3. In the computational world, however, each representable number has only a fixed and finite

number of digits. This means, for example, that only rational numbers — and not even all of these — can be represented exactly. Since $\sqrt{3}$ is not rational, it is given an approximate representation, one whose square will not be precisely 3, although it will likely be sufficiently close to 3 to be acceptable in most situations. In most cases, then, this machine arithmetic is satisfactory and passes without notice or concern — but, at times, problem arise because of this discrepancy.

The error that is produced when a calculator or computer is used to perform real-number calculations is called **round-off error**. It occurs because the arithmetic performed in a machine involves numbers with only a finite number of digits, with the result that calculations are performed with only approximate representations of the actual numebers. In a computer, only a relatively small subset of the real number system is used for the representation of all the real numbers. This subset contains only rational numbers, both positive and negative, and stores the fractional part, together with an exponential part. (We will see this in more detail later.)

1.1 Floating-point numbers

The standard way to represent a non-negative real number in decimal form is with an **integer part** and a **fractional part** with a **decimal point** between them, such as

$$37.21829, \qquad 0.002271828, \qquad 3000527.11059. \tag{1}$$

Another standard form, called **scientific notation**, is obtained by shifting the decimal point and supplying appropriate powers of 10. Thus, the preceding numbers have alternative representations as

$$37.21829 = 3.721829 \times 10^{1}, \tag{2}$$

$$0.002271828 = 0.002271828 \times 10^{-0}, \tag{3}$$

$$3000527.11059 = 3.00052711059 \times 10^{6}. \tag{4}$$

We can of course also represent negative numbers, such as

$$-3000527.11059 = -3.00052711059 \times 10^{6}. \tag{5}$$

Thus, in general, we can represent a number as

$$(-1)^{s} d_1.d_2 d_3 \cdots d_p \times 10^{k}, \tag{6}$$

$$s \in \{0, 1\}, \quad d_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}, \tag{7}$$

for $i \in \{2, 3, 4, \ldots, p\}$. Numbers of this form are called ***p*-digit decimal numbers**. We have that

$$d_1.d_2 d_3 \cdots d_p = d_1 \times 10^{0} + d_2 \times 10^{-1} + d_3 \times 10^{-2} + \cdots + d_p \times 10^{-(p-1)}. \tag{8}$$

Binary numbers. We can do analogous things in **binary**, i.e., in the base-2 system. In this case, our allowed digits are only 0 and 1. We can formulate numbers as follows:

$$0110, \quad 1101.1011, \quad 0111.100. \tag{9}$$

The corresponding number in decimal is given by, e.g.,

$$0110 = 0 \times 2^{0} + 1 \times 2^{1} + 1 \times 2^{2} + 0 \times 2^{3} = 6 \tag{10}$$

With decimal points:

$$1101.1011 = 1\times 2^0 + 0\times 2^1 + 1\times 2^2 + 1\times 2^3 + 1\times 2^{-1} + 0\times 2^{-2} + 1\times 2^{-3} + 1\times 2^{-4} = 13.6875, \quad (11)$$

$$0111.100 = 1\times 2^0 + 1\times 2^1 + 1\times 2^2 + 0\times 2^3 + 1\times 2^{-1} + 0\times 2^{-2} + 0\times 2^{-3} = 7.5. \quad (12)$$

Analogous to (2)–(4), we obtain the following representation of these numbers in the binary version of scientific notation:

$$0110 = (0.110)_2 \times 2^3, \quad (13)$$

$$1101.1011 = (1.1011011)_2 \times 2^3, \quad (14)$$

$$0111.100 = (0.111100)_2 \times 2^3, \quad (15)$$

where $(x)_2$ indicates that x is being written in base-2.

Example 1 (Converting from decimal to binary)**.** Let us recall how to convert numbers from decimal to binary. All we have to do is repeatedly divide the number by 2 while recording the quotient (the integer part of the division) and the remainder. The remainder will always be either 0 or 1. Once the quotient is 0, we obtain the binary form by arranging the remainders from last to first.
As an example, consider the decimal number 13.

Division	Quotient	Remainder
$\frac{13}{2}$	6	1
$\frac{6}{2}$	3	0
$\frac{3}{2}$	1	1
$\frac{1}{2}$	0	1

We thus have $(13)_{10} = (1101)_2$.

For a fraction, we repeatedly multiply the number by 2, recording the integer part and the remaining fraction. You stop once the remaining fraction is 0. The integer parts will then form the binary representation by arranging them from first to last. As an example, consider the decimal number 0.625.

Multiplication	Integer Part	Remaining Fraction
$0.625 \times 2 = 1.25$	1	0.25
$0.25 \times 2 = 0.5$	0	0.5
0.5×2	1	0

Thus, we have $(0.625)_{10} = (0.101)_2$. For a fractional number such as 13.625, we can simply convert the integer part and the fraction separately, so that $(13.625)_{10} = (1101.101)_2$.

Definition of floating-point numbers. We are now ready to formally define floating-point numbers. Intuitively, a floating-point number is just a number represented in scientific notation in a chosen base, with a particular convention for placing the decimal point, a fixed number of digits, and a fixed range of the exponent.

After formally defining floating-point numbers, we will then define common floating-point representations such as 32-, 64-, and 128-bit floating-point numbers.

Definition 1 (Floating-point number)**.** A **normal floating-point number** is a decimal number specified in the following manner:

$$(-1)^s (d_1.d_2 d_3 \cdots d_p)_b \times b^k, \tag{16}$$

where $b \in \{2, 3, \dots\}$ is the base and:

- $s \in \{0, 1\}$ determines the sign;
- $p \in \{1, 2, 3, \dots\}$ is the *precision*;
- $(d_1.d_2 d_3 \cdots d_p)_b$ is the p-digit *significand*, such that $d_1 \in \{1, 2, \dots, b-1\}$ and $d_i \in \{0, 1, \dots, b-1\}$, $i \in \{2, 3, \dots, p\}$;
- k is the exponent, with $k \in \{k_{\min}, k_{\min}+1, \dots, k_{\max}\}$ for a given $k_{\min}$ and $k_{\max}$.

A **subnormal floating-point number** is specified as

$$(-1)^s (0.d_2 d_3 \cdots d_p)_b \times b^{k_{\min}}, \tag{17}$$

where $d_i \in \{0, 1, \dots, b-1\}$ and not all the d_i are simultaneously equal to zero.
The triple $(p, k_{\min}, k_{\max})$ defines a **floating-point number format/system**. We use $\mathrm{fl}_{p,k_{\min},k_{\max}}(x)$ to denote the floating-point representation of the decimal number x within the specified format. We sometimes omit the subscript in $\mathrm{fl}_{p,k_{\min},k_{\max}}$ whenever the floating-point formatis understood from the context being considered.

Remark 1. *Observe that in both* (16) *and* (17)*, the number of digits in the number is p. This is specified and fixed by the definition of the floating-point format.*

Remark 2. *The term "floating-point" comes from the fact that the range of numbers represented by the floating-point system is given by simply varying the exponent k, which moves ("floats") the decimal point. We will see this in the example below.*

Remark 3. *Note that the number 0 is not represented in our definition of a floating-point number system! This comes from the fact that in subnormal floating-point numbers the digits d_i are not allowed to all be equal to 0 simultaneously. This is largely a matter of convention.*

Example 2 (Simple toy example)**.** Let us unpack the definition of floating-point number presented above with some examples. Consider the following decimal-based floating-point system defined as follows:

$$b = 10, \quad p = 3, \quad k_{\min} = -2, \quad k_{\max} = 2. \tag{18}$$

A normal number in this system has the form $(-1)^s (d_1.d_2 d_3)_{10} \times 10^k$, where $d_1 \in \{1, 2, \dots, 9\}$, $d_2, d_3 \in \{0, 1, 2, \dots, 9\}$, and $k \in \{-2, -1, 0, 1, 2\}$. Thus, by varying d_1, d_2, d_3 and moving the decimal point (i.e., varying k within its specified range), this floating-point system can exactly represent the numbers

$$0.0d_1 d_2 d_3, \dots, d_1 d_2 d_3 \Rightarrow 0.0100, 0.0101, \dots, 999, \tag{19}$$

along with their negative counterparts. A subnormal number has the form $(-1)^s (0.d_2 d_3)_{10} \times 10^{-2}$, where $d_2, d_3 \in \{0, 1, \dots, 9\}$, both d_2 and d_3 are not simultaneously equal to zero, and $k \in \{-2, -1, 0, 1, 2\}$. With this, we can exactly represent the numbers

$$0.00d_2 d_3, \dots, d_2 d_3 \Rightarrow 0.0001, 0.0002, \dots, 0.0099, \tag{20}$$

along with their negative counterparts.
Now, consider the number 0.0123. This number can be represented exactly using our floating-point number system as the following normal number:

$$0.0123 = 1.23 \times 10^{-2}. \tag{21}$$

The number 0.0023 can be represented exactly in our system as a subnormal number:

$$0.0023 = 0.23 \times 10^{-2}. \tag{22}$$

Note that, mathematically, we could have simply moved the decimal point to write $0.0023 = 2.30 \times 10^{-3}$ — but notice that this would be outside of what is allowed in our floating-point system, because the exponent -3 is not in the valid range specified by our system.

The number line for this floating-point system is shown below.

We can see that this system has "gaps" — numbers that are not represented at all by this system.

Example 3 (A binary toy example)**.** The previous example was in decimal, but machines naturally represent numbers in binary. So let us look an analogous example in binary. Let us define a binary floating-point system as follows:

$$b = 2, \quad p = 3, \quad k_{\min} = -2, \quad k_{\max} = 2. \tag{23}$$

A normal number in this system has the form $(-1)^s (d_1.d_2d_3)_2 \times 2^k$, where $d_1 = 1$, $d_2, d_3 \in \{0, 1\}$, and $k \in \{-2, -1, 0, 1, 2\}$. Thus, by varying d_1, d_2, d_3 and moving the decimal point (i.e., varying k within its specified range), this floating-point system can exactly represent the numbers

$$(0.0d_1d_2d_3)_2, \ldots, (d_1d_2d_3)_2 \Rightarrow (0.0100)_2, (0.0101)_2, (0.0110)_2, \ldots, (111)_2, \tag{24}$$

along with their negative counterparts. In decimal, this range is

$$\frac{4}{16}, \frac{5}{16}, \frac{6}{16}, \ldots, 7. \tag{25}$$

A subnormal number has the form $(-1)^s (0.d_2d_3)_2 \times 2^{-2}$, where $d_2, d_3 \in \{0, 1\}$, both d_2 and d_3 are not simultaneously equal to zero, and $k \in \{-2, -1, 0, 1, 2\}$. With this, we can exactly represent the numbers

$$(0.00d_2d_3)_2, \ldots, (d_2d_3)_2 \Rightarrow (0.0001)_2, (0.0010)_2, (0.0011)_2, \tag{26}$$

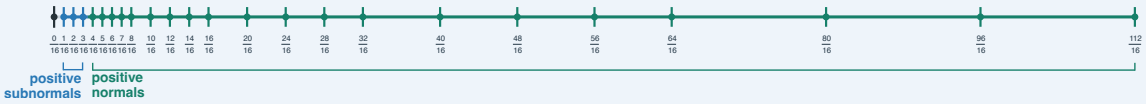
along with their negative counterparts, which in decimal is

$$\frac{1}{16}, \frac{2}{16}, \frac{3}{16}. \tag{27}$$

The full range of positive subnormal and normal values is not too many here, and we can straightforwardly write them all down:

$$\frac{1}{16}, \frac{2}{16}, \frac{3}{16}, \frac{4}{16}, \frac{5}{16}, \frac{6}{16}, \frac{7}{16}, \frac{8}{16}, \frac{10}{16}, \frac{12}{16}, \frac{14}{16}, \frac{16}{16}, \frac{20}{16}, \frac{24}{16}, \frac{28}{16}, \frac{32}{16}, \frac{40}{16}, \frac{48}{16}, \frac{56}{16}, \frac{64}{16}, \frac{80}{16}, \frac{96}{16}, \frac{112}{16}. \quad (28)$$

The number line for this floating-point system is shown below.



In the above two examples, we have seen explicitly that floating-point systems exactly represent only discrete points on the real number line. What happens if we wish to represent a number that is not exactly represented by the given floating-point systems? If, in the course of a computation, a number is produced that is outside the computer's permissible range (because it is greater that the largest value that can be represented within the corresponding floating-point system), then we say an **overflow** has occurred. Generally, an overflow results in a fatal error/exception, and the normal execution of the program stops. An **underflow** is usually treated automatically by setting the number to zero without any interruption of the program and without a warning message in most computers. When we want to represent a number that lies within the range of the floating-point system (but not exactly represented by the system), then we must resort to rounding and/or chopping. We will discuss this in Section 2.2.

1.2 Standard floating-point systems

In 1985, the IEEE (Institute for Electrical and Electronic Engineers) published a report called *Binary Floating Point Arithmetic Standard 754–1985*. An updated version was published in 2008 as *IEEE 754–2008*. This provides standards for binary and decimal floating point numbers, formats for data interchange, algorithms for rounding arithmetic operations, and the handling of exceptions. Formats are specified for single, double, and extended precisions, and these standards are generally followed by all microcomputer manufacturers using floating-point hardware. **The current IEEE standard is IEEE 754–2019.**

Definition 2 (Common floating-point formats)**.** The most common IEEE 754 floating point formats are the `binary16`, `binary32` (sometimes referred to as `float`), and `binary64` (sometimes referred to as `double`) formats. The following table summarizes all of them.

Format	Sign | Significand | Exponent Bits	Precision	Decimal Precision
`binary16` (half)	1 | 10 | 5	11 bits	≈ 3-4 digits
`binary32` (single)	1 | 23 | 8	24 bits	≈ 7 digits
`binary64` (double)	1 | 52 | 11	53 bits	≈ 16 digits
`binary128` (quadruple)	1 | 112 | 15	113 bits	≈ 34 digits

There are some useful **special numbers** in the IEEE standard. Instead of terminating with an overflow when dividing a nonzero number by 0, the machine representation for ∞ is stored. In fact, both $+\infty$ and $-\infty$ are explicitly stored, and the reason is to distinguish between a large positive result ($+\infty$) and a large negative

result ($-\infty$). Also, as already alluded to earlier, our standard definition of floating-point numbers does not include zero. The IEEE standard includes this, and in particular it includes $+0$ and -0 — again, both signs, and in this case the signs distinguish between a small positive number ($+0$) and a small negative number (-0). There is also `NaN`, which stands for *Not a Number*, and it is really an error pattern rather than a number.

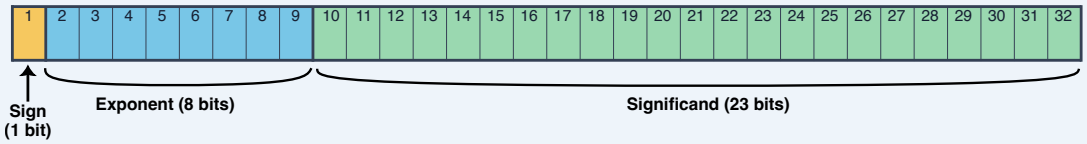
For example, we have things like

$$\infty - \infty = \text{NaN}, \quad 0 \times \infty = \text{NaN}, \quad \frac{0}{0} = \frac{\infty}{\infty} = \text{NaN}. \tag{29}$$

We also have

$$\frac{3}{+0} = +\infty, \quad \frac{3}{-0} = -\infty, \quad \frac{-3}{+0} = -\infty, \quad \frac{-3}{-0} = +\infty. \tag{30}$$

Example 4 (Understanding `binary32`)**.** In the case of `binary32`, we have: $b = 2$, $p = 24$, $k_{\min} = -126$, and $k_{\max} = 127$. This is by definition. To see why this is 32 bits overall, observe that *storing the sign requires one bit*. The exponent k can be any integer between -126 and 127. The total number of permitted exponents is therefore $127 - (-126) + 1 = 254$. As mentioned above, the IEEE standard adds 0 and ∞, which are encoded in the exponent value. This brings the total number of exponents to $254 + 2 = 256 = 2^8$. Therefore, *8 bits are needed to store the exponent*. Finally, every normal number has the form $(1.d_1 d_2 \cdots d_{23})_2$, $d_i \in \{0, 1\}$. Note that the leading 1 is always there. For subnormal numbers, the exponent is fixed to $k_{\min}$ and the leading digit is always zero. As such, *storing the significand requires only 23 bits.* Thus, the total number of bits is $1 + 8 + 23 = 32$, which we can summarize in the following diagram.



Note that NaN can already be encoded within the 32 bits, without adding more bits. A commonly-used encoding for NaN is

$$\texttt{NaN} \equiv 0|11111111|10000000000000000000000. \tag{31}$$

The signed zeros are given by:

$$+0 \equiv 0|00000000|00000000000000000000000, \tag{32}$$

$$-0 \equiv 1|00000000|00000000000000000000000. \tag{33}$$

The signed infinities are:

$$+\infty \equiv 0|11111111|00000000000000000000000, \tag{34}$$

$$-\infty \equiv 1|11111111|00000000000000000000000. \tag{35}$$

Example 5. Let us determine the `binary32` (i.e., single-precision) representation of the decimal number -52.234375.

First, it is straightforward to show that

$$(52.234375)_{10} = (110100.001111)_2 = (1.10100001111)_2 \times 2^5. \tag{36}$$

Therefore, we have

$$1|10000100|10100001111000000000000. \tag{37}$$

2 Errors and how to quantify them

We have seen that floating-point systems cannot represent every real number exactly — any given floating-point system will exactly represent only a discrete, finite set of points on the real line. Dealing with intermediate values between the discrete points will introduce errors. Below we define two common error metrics.

Definition 3 (Absolute and relative error)**.** If x is exact and $\widehat{x}$ is an approximation, define

$$\varepsilon = x - \widehat{x} \quad \text{(Actual error)}, \tag{38}$$

$$\varepsilon_{\text{abs}} = |x - \widehat{x}| \quad \text{(Absolute error)}, \tag{39}$$

$$\varepsilon_{\text{rel}} = \frac{|x - \widehat{x}|}{|x|} \quad (x \neq 0) \quad \text{(Relative error)}. \tag{40}$$

Absolute error carries the units of the quantity; relative error measures the error compared with the scale of the answer.

Example 6. Determine the actual, absolute, and relative errors when approximating x by $\hat{x}$ in the following cases.

(a) $x = 0.3000 \times 10^1, \widehat{x} = 0.3100 \times 10^1$

(b) $x = 0.3000 \times 10^{-3}, \widehat{x} = 0.3100 \times 10^{-3}$

(c) $x = 0.3000 \times 10^4, \widehat{x} = 0.3100 \times 10^4$

Solution

(a) The actual error is -0.1, the absolute error is 0.1, and the relative error is $0.3333\cdots \times 10^{-1}$.

(b) The actual error is -0.1×10^{-4}, the absolute error is 0.1×10^{-4}, and the relative error is $0.3333\cdots \times 10^{-1}$.

(c) The actual error is -0.1×10^3, the absolute error is 0.1×10^3, and the relative error is again $0.3333\cdots \times 10^{-1}$.

This example shows that the same relative error, $0.3333\cdots \times 10^{-1}$, occurs for widely varying absolute errors. As a measure of accuracy, the absolute error can be misleading and the relative error more meaningful, because the relative error takes into consideration the size of the value.

Example 7. Let $x_1 = 1.333$, $\widehat{x_1} = 1.334$, and $x_2 = 0.001$, $\widehat{x_2} = 0.002$. What are the absolute errors and relative errors of $\widehat{x_i}$ as an approximation of x_i?

Solution: The absolute error is the same in both cases, namely, 10^{-3}. However, the relative errors are approximately 0.75×10^{-3} for $\widehat{x_1}$ approximating x_1 and 1 for $\widehat{x_2}$ approximating x_2. The relative error clearly indicates that $\widehat{x_1}$ is a good approximation to x_1, while $\widehat{x_2}$ is a poor approximation to x_2.

2.1 Significant digits

Significant digits (also sometimes "significant figures") describe the number of digits of a number when written in normalized scientific notation.

Consider a base $b \in \{2, 3, \dots\}$. A non-zero finite numeral in normalized form can always be expressed as

$$(x)_b = \pm(d_0.d_1 d_2 \cdots d_{n-1})_b \times b^k, \tag{41}$$

where $d_0 \in \{1, 2, \dots, b-1\}$ and $d_i \in \{0, 1, \dots, b-1\}$ for $i \in \{1, 2, \dots, n-1\}$. In this case, by definition, the number x as n significant digits. (Note that, unlike in our definition of floating-point system, the exponent k is not restricted; so we can indeed always express any number in this form, by appropriately choosing the exponent k.)

Suppose that a real number is expressed in normalized scientific notation in the decimal system as follows:

$$3.721498 \times 10^{-5}. \tag{42}$$

The digits 3, 7, 2, 1, 4, 9, 8 do not all have the same significance because they represent different powers of 10. Thus, we say that 3 is the **most significant digit**, and the significance of the digits diminishes from left to right. In this example, 8 is the **least significant digit**. The total number of significant digits here is seven.

Remark 4. *Note: sometimes the concepts of significant digits and precision are used synonymously. For us, precision is always defined within the context of some floating-point number system. The number of significant digits, on the other hand, is purely a function of the literal numerical representation — it does not necessarily convey, on its own, anything about precision or accuracy.*

2.2 Rounding and chopping

We have seen that floating-point numbers are specified by a particular precision p. Given a real number x, we have defined $\mathrm{fl}(x)$ to be its floating-point representation within the specified floating-point number system. To completely specify what $\mathrm{fl}(x)$ does, we must fix the rounding or chopping rule, in addition to the floating-point system.

Chopping is perhaps the simplest method. Suppose the number x (say, in decimal) is expressed as

$$x = d_1.d_2 d_3 \cdots d_p d_{p+1} d_{p+2} \cdots \times 10^k, \tag{43}$$

where p is the precision given by our floating-point system and k is an exponent within the admissible range of values (which is also given by definition of the floating-point system), then **chopping** produces the floating-point number

$$x = d_1.d_2 d_3 \cdots d_p d_{p+1} d_{p+2} \cdots^k \Rightarrow \mathrm{fl}(x) = d_1.d_2 d_3 \cdots d_p \times 10^k \quad \text{(chopping)}. \tag{44}$$

On the other hand, **rounding** works differently. If $d_{p+1} \geq 5$, then we add 1 to d_p to obtain $\mathrm{fl}(x)$ — that is, we **round up**. When $d_{p+1} < 5$, then we simply chop off all but the first p digits — that is, we **round down**.

There is another rounding rule for when $d_{p+1} = 5$, which is referred to as the **round-to-even** (or **ties-to-even**) rule. This specifies that if $d_{p+1} = 5$ and d_p is even, then d_p remains as it is; if d_p is odd, then we add one to it. This means that

$$0.365 \to 0.36, \quad 0.475 \to 0.48 \qquad \text{(rounds-to-even rule)}. \tag{45}$$

The rounds-to-even rule is often preferred, because for large data this rule tends to reduce the total rounding error due to the (on average) equal portion of numbers being rounded up and rounded down. This is the rule used in the IEEE 754 standard.

Example 8. Determine the five-digit (a) chopping; and (b) rounding values of the irrational number π.

Solution: The number π has an infinite decimal expansion of the form $\pi = 3.14159265\ldots$.

(a) The floating-point form of π using five-digit chopping is 3.1415.

(b) The sixth digit of the decimal expansion of π is 9, so the floating-point form of π using five-digit rounding is 3.1416.

Example 9 (Simple toy example revisited)**.** Let us revisit the decimal-based toy example from Example 2. Recall that the spacing for subnormal numbers is 0.0001. For normal numbers, the spacing depends on the exponent. For example, around 37, the adjacent values are 37.2, 37.3, 37.4, etc.

- If we consider the number $x = 37.21829$, then its two nearest floating-point neighbors are 37.2 and 37.3. Using the rounding rule from above, we would obtain $\mathrm{fl}(37.21829) = 37.2$. We would get the same in this case from chopping.
- Next, consider the number $x = 0.00234$. The neighboring subnormal numbers are 0.0023 and 0.0024. Thus, using rounding, we obtain $\mathrm{fl}(0.00234) = 0.0023$.
- Now consider the case $x = 37.25$. It lies exactly halfway between the two normal numbers 37.2 and 37.3. From the rounds-to-even rule, we obtain $\mathrm{fl}(37.25) = 37.2$, while with rounding up we would obtain 37.3. Similarly, for $x = 0.00235$, which is halfway between 0.0023 and 0.0024, we would obtain in this case $\mathrm{fl}(0.00235) = 0.0024$, both from the rounds-to-even rule and from rounding up.
- Finally, if have a number like $x = 0.00004$, then because that number is closer to zero than to the first subnormal number in our floating-point system, namely 0.0001, we obtain $\mathrm{fl}(0.00004) = 0$. On the other hand, by rounding up, we obtain $\mathrm{fl}(0.00006) = 0.0001$.

Example 10 (Binary toy example revisited)**.** Let us go through the same type of examples for our binary toy example from Example 3. Recall that the finest spacing is $\frac{1}{16}$. Near zero, the positive representable values are $\frac{1}{16}, \frac{2}{16}, \frac{3}{!6}, \frac{4}{16}, \frac{5}{16}, \ldots$. The first three are subnormal; $\frac{4}{16}$ is the smallest positive normal.

- The number 0.1 lies between $\frac{1}{16} = 0.0625$ and $\frac{2}{16} = 0.125$. Its distances are $0.1 - 0.0625 = 0.0375$ and $0.125 - 0.1 = 0.025$. Therefore, $\mathrm{fl}(0.1) = \frac{2}{16} = 0.125$.
- Considering 0.2, it lies between $\frac{3}{16} = 0.1875$ and $\frac{4}{16} = 0.25$; therefore, being closer to $\frac{3}{16}$, we obtain

$\mathrm{fl}(0.2) = \frac{3}{16} = 0.1875$.

- Now consider $\frac{9}{32} = 0.28125$. This is exactly halfway between $\frac{4}{16} = 0.25$ and $\frac{5}{16} = 0.3125$. Also note that $\frac{9}{32} = (1.001)_2 \times 2^{-2}$, while $\frac{4}{16} = (1.00)_2 \times 2^{-2}$ and $(1.01)_2 \times 2^{-2}$. The first one, containing a 0 in the last digit, is even, while the second one is odd; therefore, the rounds-to-even rule establishes that $\mathrm{fl}(\frac{9}{32}) = \frac{4}{16} = \frac{1}{4}$.
- If we take $\frac{11}{32} = (1.011)_2 \times 2^{-2}$, then the closest neighboring representable values are $\frac{5}{16} = \frac{10}{32}$ and $\frac{6}{16} = \frac{12}{32}$, with the distances again being equal. Noting that $\frac{5}{16} = (1.01)_2 \times 2^{-2}$ and $\frac{6}{16} = (1.10)_2 \times 2^{-2}$, the rounds-to-even rule tells us that $\mathrm{fl}(\frac{11}{32}) = \frac{6}{16} = \frac{3}{8}$.

Theorem 5 (Floating-point relative error)**.** Consider a base-b floating-point number system that specifies $p \in \{1, 2, \dots\}$ digits of precision, where $b \in \{2, 3, \dots\}$. Let $x \neq 0$ be an arbitrary real number satisfying $x_{\min}^{\text{normal}} \leq |x| \leq x_{\max}$, where $x_{\min}^{\text{normal}}$ is the smallest normal number represented by the floating-point system and $x_{\max}$ is the largest such number. Consider the floating-point approximation $\mathrm{fl}(x)$ of x, given by rounding to the nearest representable number using, e.g., the rounds-to-even rule, and assume that the rounding does not over/underflow to $\pm\infty$. Then, the relative error in the floating-point approximation of x satisfies the following upper bound:

$$\frac{|x - \mathrm{fl}(x)|}{|x|} \leq u \coloneqq \frac{1}{2} b^{-p+1}. \tag{46}$$

Futhermore, there exists $\delta \in \mathbb{R}$ such that

$$\mathrm{fl}(x) = x(1 + \delta), \quad |\delta| \leq u. \tag{47}$$

Proof. Consider the exponent value $k \in \{k_{\min}, k_{\min} + 1, \dots, k_{\max}\}$, with $k_{\min}$ and $k_{\max}$ specified by the floating-point system, such that $b^k \leq |x| \leq b^{k+1}$. (This value is simply $k = \lfloor \log_b |x| \rfloor$.) Within this exponent range, consecutive p-digit floating-point numbers are separated by $\Delta = b^{k-p+1}$. Because $\mathrm{fl}(x)$ is a nearest floating-point number, it holds that

$$|x - \mathrm{fl}(x)| \leq \frac{\Delta}{2} = \frac{1}{2} b^{k-p+1}. \tag{48}$$

At an exact midpoint, the rounds-to-even rule chooses one of the two endpoints, but the error is still exactly $\frac{\Delta}{2}$. Therefore, using the fact that $|x| \geq b^k$,

$$\frac{|x - \mathrm{fl}(x)|}{|x|} \leq \frac{\frac{1}{2} b^{k-p+1}}{|x|} \tag{49}$$

$$\leq \frac{\frac{1}{2} b^{k-p+1}}{b^k} \tag{50}$$

$$= \frac{1}{2} b^{-p+1} \tag{51}$$

$$= u, \tag{52}$$

as desired. Now, define

$$\delta = \frac{\mathrm{fl}(x) - x}{x}. \tag{53}$$

Then, $\mathrm{fl}(x) - x = x\delta$, which implies that $\mathrm{fl}(x) = x(1+\delta)$. Moreover, using the steps shown above, we have that $|\delta| = \left|\frac{\mathrm{fl}(x)-x}{x}\right| \leq u$. This concludes the proof. ■

Definition 4 (Unit roundoff and machine epsilon)**.** For a given floating-point system, the quantity $u = \frac{1}{2}b^{-p+1}$ in Theorem 5 is referred to as the **unit roundoff error**. It is the smallest value of δ such that $\mathrm{fl}(1+\delta) > 1$; equivalently,

$$u = \inf\left\{\delta : \mathrm{fl}(1+\delta) > 1\right\}. \tag{54}$$

A related quantity is the **machine epsilon**. This quantity is defined as the distance from 1 to the next larger floating-point number. This quantity is $\varepsilon_{\text{machine}} = b^{1-p} = 2u$.

Example 11 (The error in IEEE 754 floating-point systems)**.** Using Theorem 5, we can easily determine the unit roundoff and machine epsilon for the IEEE 754 floating-point formats. For example for `binary32` (single) we have $p = 24$, so $\varepsilon_{\text{machine}} = 2^{-23}$. For `binary64` (double) we have $p = 53$, so $\varepsilon_{\text{machine}} = 2^{-52}$.