

1 Numbers

Reference: [Overton](#)

In this chapter, we introduce the [Two's-complement](#) storage for integers and the [IEEE Standard for Floating-Point Arithmetic](#). There are many possible ways of representing real numbers on a computer, as well as the precise behaviour of operations such as addition, multiplication, etc. Before the 1980s each processor had potentially a different representation for real numbers, as well as different behaviour for operations. IEEE introduced in 1985 was a means to standardise this across processors so that algorithms would produce consistent and reliable results.

This chapter may seem very low level for a mathematics course but there are two important reasons to understand the behaviour of integers and floating-point numbers:

1. Integer arithmetic can suddenly start giving wrong negative answers when numbers

become large.

2. Floating-point arithmetic is very precisely defined, and can even be used

in rigorous computations as we shall see in the problem sheets. But it is not exact and its important to understand how errors in computations can accumulate.

3. Failure to understand floating-point arithmetic can cause catastrophic issues

in practice, with the extreme example being the [explosion of the Ariane 5 rocket](#).

In this chapter we discuss the following:

1. Binary representation: Any real number can be represented in binary, that is,

by an infinite sequence of 0s and 1s (bits). We review binary representation.

2. Integers: There are multiple ways of representing integers on a computer. We discuss the

the different types of integers and their representation as bits, and how arithmetic operations behave like modular arithmetic. As an advanced topic we discuss `BigInt`, which uses variable bit length storage.

2. Floating-point numbers: Real numbers are stored on a computer with a finite number of bits.

There are three types of floating-point numbers: *normal numbers*, *subnormal numbers*, and *special numbers*.

3. Arithmetic: Arithmetic operations in floating-point are exact up to rounding, and how the

rounding mode can be set. This allows us to bound errors computations.

4. High-precision floating-point numbers: As an advanced topic, we discuss how the precision of floating-point arithmetic can be increased arbitrary

using `BigFloat`.

Before we begin, we load two external packages. `SetRounding.jl` allows us to set the rounding mode of floating-point arithmetic. `ColorBitstring.jl` implements functions `printbits` (and `printlnbits`) which print the bits (and with a newline) of floating-point numbers in colour.

`using SetRounding, ColorBitstring`

1.1 Binary representation

Any integer can be presented in binary format, that is, a sequence of 0s and 1s.

Definition 1 (Binary) For $B_0, \dots, B_p \in \{0, 1\}$ denote a non-negative integer in *binary format* by:

$$(B_p \dots B_1 B_0)_2 := 2^p B_p + \dots + 2 B_1 + B_0$$

For $b_1, b_2, \dots \in \{0, 1\}$, Denote a non-negative real number in *binary format* by:

$$(B_p \dots B_0 . b_1 b_2 b_3 \dots)_2 = (B_p \dots B_0)_2 + \frac{b_1}{2} + \frac{b_2}{2^2} + \frac{b_3}{2^3} + \dots$$

First we show some examples of verifying a numbers binary representation:

Example 1 (integer in binary) A simple integer example is $5 = 2^2 + 2^0 = (101)_2$.

Example 2 (rational in binary) Consider the number $1/3$. In decimal recall that:

$$1/3 = 0.3333\dots = \sum_{k=1}^{\infty} \frac{3}{10^k}$$

We will see that in binary

$$1/3 = (0.010101\dots)_2 = \sum_{k=1}^{\infty} \frac{1}{2^{2k}}$$

Both results can be proven using the geometric series:

$$\sum_{k=0}^{\infty} z^k = \frac{1}{1-z}$$

provided $|z| < 1$. That is, with $z = \frac{1}{4}$ we verify the binary expansion:

$$\sum_{k=1}^{\infty} \frac{1}{4^k} = \frac{1}{1-1/4} - 1 = \frac{1}{3}$$

A similar argument with $z = 1/10$ shows the decimal case.

1.2 Integers

On a computer one typically represents integers by a finite number of p bits, with 2^p possible combinations of 0s and 1s. For *unsigned integers* (non-negative integers) these bits are just the first p binary digits: $(B_{p-1} \dots B_1 B_0)_2$.

Integers on a computer follow [modular arithmetic](#):

Definition 2 (ring of integers modulo m) Denote the ring

$$\mathbb{Z}_m := \{0 \pmod{m}, 1 \pmod{m}, \dots, m-1 \pmod{m}\}$$

Integers represented with p -bits on a computer actually represent elements of \mathbb{Z}_{2^p} and integer arithmetic on a computer is equivalent to arithmetic modulo 2^p .

Example 3 (addition of 8-bit unsigned integers) Consider the addition of two 8-bit numbers:

$$255 + 1 = (11111111)_2 + (00000001)_2 = (100000000)_2 = 256$$

The result is impossible to store in just 8-bits! It is way too slow for a computer to increase the number of bits, or to throw an error (checks are slow). So instead it treats the integers as elements of \mathbb{Z}_{256} :

$$255 + 1 \pmod{256} = (00000000)_2 \pmod{256} = 0 \pmod{256}$$

We can see this in Julia:

```
x = UInt8(255)
y = UInt8(1)
printbits(x); println(" + "); printbits(y); println(" = ")
printbits(x + y)
```

```
11111111 +
00000001 =
00000000
```

Example 4 (multiplication of 8-bit unsigned integers) Multiplication works similarly: for example,

$$254 * 2 \pmod{256} = 252 \pmod{256} = (11111100)_2 \pmod{256}$$

We can see this behaviour in code by printing the bits:

```
x = UInt8(254) # 254 represented in 8-bits as an unsigned integer
y = UInt8(2)   # 2 represented in 8-bits as an unsigned integer
printbits(x); println(" * "); printbits(y); println(" = ")
printbits(x * y)
```

```
11111110 *
00000010 =
11111100
```

1.2.1 Signed integer

Signed integers use the [Two's complement](#) convention. The convention is if the first bit is 1 then the number is negative: the number $2^p - y$ is interpreted as $-y$. Thus for $p = 8$ we are interpreting 2^7 through $2^8 - 1$ as negative numbers.

Example 5 (converting bits to signed integers) What 8-bit integer has the bits 01001001? Adding the corresponding decimal places we get:

```
2^0 + 2^3 + 2^6
```

73

What 8-bit (signed) integer has the bits 11001001? Because the first bit is 1 we know it's a negative number, hence we need to sum the bits but then subtract 2^p :

```
2^0 + 2^3 + 2^6 + 2^7 - 2^8
```

-55

We can check the results using `printbits`:

```
printlnbits(Int8(73))
printbits(-Int8(55))
```

```
01001001
11001001
```

Arithmetic works precisely the same for signed and unsigned integers.

Example 6 (addition of 8-bit integers) Consider $(-1) + 1$ in 8-bit arithmetic. The number -1 has the same bits as $2^8 - 1 = 255$. Thus this is equivalent to the previous question and we get the correct result of 0. In other words:

$$-1 + 1 \pmod{2^p} = 2^p - 1 + 1 \pmod{2^p} = 2^p \pmod{2^p} = 0 \pmod{2^p}$$

Example 7 (multiplication of 8-bit integers) Consider $(-2) * 2$. -2 has the same bits as $2^{256} - 2 = 254$ and -4 has the same bits as $2^{256} - 4 = 252$, and hence from the previous example we get the correct result of -4 . In other words:

$$(-2) * 2 \pmod{2^p} = (2^p - 2) * 2 \pmod{2^p} = 2^{p+1} - 4 \pmod{2^p} = -4 \pmod{2^p}$$

Example 8 (overflow) We can find the largest and smallest instances of a type using `typemax` and `typemin`:

```
printlnbits(typemax{Int8}) # 2^7-1 = 127
printbits(typemin{Int8}) # -2^7 = -128
```

```
01111111
10000000
```

As explained, due to modular arithmetic, when we add 1 to the largest 8-bit integer we get the smallest:

```
typemax{Int8} + Int8(1) # returns typemin{Int8}
```

-128

This behaviour is often not desired and is known as *overflow*, and one must be wary of using integers close to their largest value.

1.2.2 Variable bit representation (advanced)

An alternative representation for integers uses a variable number of bits, with the advantage of avoiding overflow but with the disadvantage of a substantial speed penalty. In Julia these are `BigInts`, which we can create by calling `big` on an integer:

```
x = typemax{Int64} + big(1) # Too big to be an `Int64`
```

```
9223372036854775808
```

Note in this case addition automatically promotes an `Int64` to a `BigInt`. We can create very large numbers using `BigInt`:

```
x100
```

```
308299402527763474570010682154566572137179853330569745885534227792109373198
447640470596653941241089824056172991237203850122889314192108015240464239377
659907729443406151990542412460139422694360143091643438371471672472022733159
695061370166103454894838872109766727543876375812850840329719945826027770730
120246098009381841416708056334276148239586243518509394244354072236315177002
222178324395959253133606299849420991475240801906072080512453438264605109361
381484864606203866242348750432604436120370843048930586423433380140154714002
337629571838339036072866290023067143715171661582628684226791756074958601816
573949210192042971926128564012559683306389156286526215702602395591987379284
682309585448452092050934594471287167569179082769090777848505882924858894568
168528817978796393118106206809246398429622597308249405630795808918972670167
873557636539414623207691708807594905363669045958112877309721274696727649649
601081087800063823914375007554316324004987448998664232743644123445804025448
082503822047990459461530060239055638579924527680558002493780472302931956594
201351581704871454345525023520878974570116527956902624814539521898506299183
170783021797439315846606778519958103771496882062824105186711983296636153004
791033906572655026074103671610093220596965508325771424407112022165467934046
108400156032167602544380124835543930597492387362414798072811058145280610901
173900506006060422808766749928885121870507880736423792545581389057525756998
145009099711769746929923409439498484057402540146394209901941336109623390905
611742766343976495491640159256565111157141476925718770456826870124308204483
840020135761385100647110424482884227023263774739896271187541348841577264708
857112527293249071721746826360468332593346955562978550702077536636800275361
270990152624845632820964329212289967743661388636076587788674818529924999492
184318357313040349631189661494939940979601130119128006720905325934191881396
7552543176532349157376
```

Note the number of bits is not fixed, the larger the number, the more bits required to represent it, so while overflow is impossible, it is possible to run out of memory if a number is astronomically large: go ahead and try `xx` (at your own risk).

1.3 Division

In addition to `+`, `-`, and `*` we have integer division `÷`, which rounds down:

```
5 ÷ 2 # equivalent to div(5,2)
```

```
2
```

Standard division `/` (or `\` for division on the right) creates a floating-point number, which will be discussed shortly:

```
5 / 2 # alternatively 2 \ 5
```

2.5

We can also create rational numbers using `//`:

`(1//2) + (3//4)`

`5//4`

Rational arithmetic often leads to overflow so it is often best to combine `big` with rationals:

`big(102324)//132413023 + 23434545//4243061 + 23434545//42430534435`

`26339037835007648477541540//4767804878707544364596461`

1.4 3. Floating-point numbers

Floating-point numbers are a subset of real numbers that are representable using a fixed number of bits.

Definition 3 (floating-point numbers) Given integers σ (the "exponential shift") Q (the number of exponent bits) and S (the precision), define the set of *Floating-point numbers* by dividing into *normal*, *sub-normal*, and *special number* subsets:

$$F_{\sigma,Q,S} := F_{\sigma,Q,S}^{\text{normal}} \cup F_{\sigma,Q,S}^{\text{sub}} \cup F^{\text{special}}.$$

The *normal numbers* $F_{\sigma,Q,S}^{\text{normal}} \subset \mathbb{R}$ are defined by

$$F_{\sigma,Q,S}^{\text{normal}} = \{\pm 2^{q-\sigma} \times (1.b_1b_2b_3 \dots b_S)_2 : 1 \leq q < 2^Q - 1\}.$$

The *sub-normal numbers* $F_{\sigma,Q,S}^{\text{sub}} \subset \mathbb{R}$ are defined as

$$F_{\sigma,Q,S}^{\text{sub}} = \{\pm 2^{1-\sigma} \times (0.b_1b_2b_3 \dots b_S)_2\}.$$

The *special numbers* $F^{\text{special}} \not\subset \mathbb{R}$ are defined later.

Note this set of real numbers has no nice algebraic structure: it is not closed under addition, subtraction, etc. We will therefore need to define approximate versions of algebraic operations later.

Floating-point numbers are stored in $1 + Q + S$ total number of bits, in the format

$$sq_{Q-1} \dots q_0 b_1 \dots b_S$$

The first bit (s) is the sign bit: 0 means positive and 1 means negative. The bits $q_{Q-1} \dots q_0$ are the exponent bits: they are the binary digits of the unsigned integer q :

$$q = (q_{Q-1} \dots q_0)_2.$$

Finally, the bits $b_1 \dots b_S$ are the significand bits. If $1 \leq q < 2^Q - 1$ then the bits represent the normal number

$$x = \pm 2^{q-\sigma} \times (1.b_1b_2b_3 \dots b_S)_2.$$

If $q = 0$ (i.e. all bits are 0) then the bits represent the sub-normal number

$$x = \pm 2^{1-\sigma} \times (0.b_1b_2b_3 \dots b_S)_2.$$

If $q = 2^Q - 1$ (i.e. all bits are 1) then the bits represent a special number, discussed later.

1.4.1 IEEE floating-point numbers

Definition 4 (IEEE floating-point numbers) IEEE has 3 standard floating-point formats: 16-bit (half precision), 32-bit (single precision) and 64-bit (double precision) defined by:

$$\begin{aligned} F_{16} &:= F_{15,5,10} \\ F_{32} &:= F_{127,8,23} \\ F_{64} &:= F_{1023,11,52} \end{aligned}$$

In Julia these correspond to 3 different floating-point types:

1. `Float64` is a type representing double precision (F_{64}).

We can create a `Float64` by including a decimal point when writing the number: `1.0` is a `Float64`. `Float64` is the default format for scientific computing (on the *Floating-Point Unit*, FPU).

2. `Float32` is a type representing single precision (F_{32}). We can create a `Float32` by including a

`f0` when writing the number: `1f0` is a `Float32`. `Float32` is generally the default format for graphics (on the *Graphics Processing Unit*, GPU), as the difference between 32 bits and 64 bits is indistinguishable to the eye in visualisation, and more data can be fit into a GPU's limited memory.

3. `Float16` is a type representing half-precision (F_{16}).

It is important in machine learning where one wants to maximise the amount of data and high accuracy is not necessarily helpful.

Example 9 (rational in Float32) How is the number $1/3$ stored in `Float32`? Recall that

$$1/3 = (0.010101\dots)_2 = 2^{-2}(1.0101\dots)_2 = 2^{125-127}(1.0101\dots)_2$$

and since $125 = (1111101)_2$ the exponent bits are `01111101`. For the significand we round the last bit to the nearest element of F_{32} , (this is explained in detail in the section on rounding), so we have

$$1.010101010101010101010101\dots \approx 1.01010101010101010101011 \in F_{32}$$

and the significand bits are `010101010101010101011`. Thus the `Float32` bits for $1/3$ are:

```
printbits(1f0/3)
```

```
001111101010101010101010101011
```

For sub-normal numbers, the simplest example is zero, which has $q = 0$ and all significand bits zero:

```
printbits(0.0)
```

```
0000000000000000000000000000000000000000000000000000000000000000
```

Unlike integers, we also have a negative zero:

```
printbits(-0.0)
```

```
1000000000000000000000000000000000000000000000000000000000000000
```

This is treated as identical to 0.0 (except for degenerate operations as explained in special numbers).

1.4.2 Special normal numbers

When dealing with normal numbers there are some important constants that we will use to bound errors.

Definition 5 (machine epsilon/smallest positive normal number/largest normal number) *Machine epsilon* is denoted

$$\epsilon_{m,S} := 2^{-S}.$$

When S is implied by context we use the notation ϵ_m . The *smallest positive normal number* is $q = 1$ and b_k all zero:

$$\min |F_{\sigma,Q,S}^{\text{normal}}| = 2^{1-\sigma}$$

where $|A| := \{|x| : x \in A\}$. The *largest (positive) normal number* is

$$\max F_{\sigma,Q,S}^{\text{normal}} = 2^{2^Q-2-\sigma}(1.11\dots 1)_2 = 2^{2^Q-2-\sigma}(2 - \epsilon_m)$$

We confirm the simple bit representations:

```
 $\sigma, Q, S = 127, 23, 8$  # Float32
```

```
 $\epsilon_m = 2.0^{-(S)}$ 
```

```
printlnbits(Float32(2.0^(1- $\sigma$ ))) # smallest positive Float32
```

```
printlnbits(Float32(2.0^(2Q-2- $\sigma$ ) * (2- $\epsilon_m$ ))) # largest Float32
```

```
00000000100000000000000000000000
```

```
01111111100000000000000000000000
```

For a given floating-point type, we can find these constants using the following functions:

```
eps(Float32), floatmin(Float32), floatmax(Float32)
```

```
(1.1920929f-7, 1.1754944f-38, 3.4028235f38)
```

Example 10 (creating a sub-normal number) If we divide the smallest normal number by two, we get a subnormal number:


```
mn = floatmin(Float32) # smallest normal Float32
printlnbits(mn)
printbits(mn/2)
```

```
00000000100000000000000000000000
00000000010000000000000000000000
```

Can you explain the bits?

1.4.3 Special numbers

The special numbers extend the real line by adding $\pm\infty$ but also a notion of "not-a-number".

Definition 6 (not a number) Let NaN represent "not a number" and define

$$F^{\text{special}} := \{\infty, -\infty, \text{NaN}\}$$

Whenever the bits of q of a floating-point number are all 1 then they represent an element of F^{special} . If all $b_k = 0$, then the number represents either $\pm\infty$, called **Inf** and **-Inf** for 64-bit floating-point numbers (or **Inf16**, **Inf32** for 16-bit and 32-bit, respectively):

```
printlnbits(Inf16)
printbits(-Inf16)
```

```
0111110000000000
1111110000000000
```

All other special floating-point numbers represent NaN. One particular representation of NaN is denoted by **NaN** for 64-bit floating-point numbers (or **NaN16**, **NaN32** for 16-bit and 32-bit, respectively):

```
printbits(NaN16)
```

```
0111111000000000
```

These are needed for undefined algebraic operations such as:

```
0/0
```

```
NaN
```

Example 11 (many NaNs) What happens if we change some other b_k to be nonzero? We can create bits as a string and see:

```
i = parse(UInt16, "0111110000010001"; base=2)
reinterpret(Float16, i)
```

```
NaN16
```

Thus, there are more than one NaNs on a computer.

1.5 4. Arithmetic

Arithmetic operations on floating-point numbers are *exact up to rounding*. There are three basic rounding strategies: round up/down/nearest. Mathematically we introduce a function to capture the notion of rounding:

In Julia, the rounding mode is specified by tags `RoundUp`, `RoundDown`, and `RoundNearest`. (There are also more exotic rounding strategies `RoundToZero`, `RoundNearestTiesAway` and `RoundNearestTiesUp` that we won't use.)

Let's try rounding a `Float64` to a `Float32`.

[illegible]

```
printbits(Float32(1/3, RoundDown) )
```

Or alternatively we can change the rounding mode for a chunk of code using `setrounding`. The following computes upper and lower bounds for `/:`

(0.3333333f0, 0.33333334f0)

In IEEE arithmetic, the arithmetic operations $+$, $-$, $*$, $/$ are defined by the property that they are exact up to rounding. Mathematically we denote these operations as follows:

$$x \oslash y := \mathfrak{fl}(x/y)$$

Note also that `~` and `sqrt` are similarly exact up to rounding.

Example 12 (decimal is not exact) `1.1+0.1` gives a different result than `1.2`:

```
x = 1.1
y = 0.1
x + y - 1.2 # Not Zero?!?

2.220446049250313e-16
```

This is because $\text{fl}(1.1) \neq 1 + 1/10$, but rather:

$$\text{fl}(1.1) = 1 + 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + \dots + 2^{-48} + 2^{-49} + 2^{-51}$$

WARNING (non-associative) These operations are not associative! E.g. $(x \oplus y) \oplus z$ is not necessarily equal to $x \oplus (y \oplus z)$. Commutativity is preserved, at least. Here is a surprising example of non-associativity:

```
(1.1 + 1.2) + 1.3, 1.1 + (1.2 + 1.3)

(3.5999999999999996, 3.6)
```

Can you explain this in terms of bits?

1.5.1 Bounding errors in floating point arithmetic

Before we discuss bounds on errors, we need to talk about the two notions of errors:

Definition 8 (absolute/relative error) If $\tilde{x} = x + \delta_{rma} = x(1 + \delta_r)$ then $|\delta_a|$ is called the *absolute error* and $|\delta_r|$ is called the *relative error* in approximating x by \tilde{x} .

We can bound the error of basic arithmetic operations in terms of machine epsilon, provided a real number is close to a normal number:

Definition 9 (normalised range) The *normalised range* $\mathcal{N}_{\sigma,Q,S} \subset \mathbb{R}$ is the subset of real numbers that lies between the smallest and largest normal floating-point number:

$$\mathcal{N}_{\sigma,Q,S} := \{x : \min |F_{\sigma,Q,S}| \leq |x| \leq \max F_{\sigma,Q,S}\}$$

When σ, Q, S are implied by context we use the notation \mathcal{N} .

We can use machine epsilon to determine bounds on rounding:

Proposition (rounding arithmetic) If $x \in \mathcal{N}$ then

$$\text{fl}^{\text{mode}}(x) = x(1 + \delta_x^{\text{mode}})$$

where the *relative error* is

$$\begin{aligned} |\delta_x^{\text{nearest}}| &\leq \frac{\epsilon_m}{2} \\ |\delta_x^{\text{up/down}}| &< \epsilon_m. \end{aligned}$$

This immediately implies relative error bounds on all IEEE arithmetic operations, e.g., if $x + y \in \mathcal{N}$ then we have

$$x \oplus y = (x + y)(1 + \delta_1)$$

where (assuming the default nearest rounding) $|\delta_1| \leq \epsilon_{\text{m}}/2$.

Example 13 (bounding a simple computation) We show how to bound the error in computing

$$(1.1 + 1.2) + 1.3$$

using floating-point arithmetic. First note that 1.1 on a computer is in fact $\text{fl}(1.1)$. Thus this computation becomes

$$(\text{fl}(1.1) \oplus \text{fl}(1.2)) \oplus \text{fl}(1.3)$$

First we find

$$(\text{fl}(1.1) \oplus \text{fl}(1.2)) = (1.1(1 + \delta_1) + 1.2(1 + \delta_2))(1 + \delta_3) = 2.3 + 1.1\delta_1 + 1.2\delta_2 + 2.3\delta_3 + 1.1\delta_1\delta_3 + 1.2\delta_2\delta_3 = 2.3 + \delta_4$$

where (note $\delta_1\delta_3$ and $\delta_2\delta_3$ are tiny so we just round up our bound to the nearest decimal)

$$|\delta_4| \leq 2.3\epsilon_{\text{m}}$$

Thus the computation becomes

$$((2.3 + \delta_4) + 1.3(1 + \delta_5))(1 + \delta_6) = 3.6 + \delta_4 + 1.3\delta_5 + 3.6\delta_6 + \delta_4\delta_6 + 1.3\delta_5\delta_6 = 3.6 + \delta_7$$

where the *absolute error* is

$$|\delta_7| \leq 4.8\epsilon_{\text{m}}$$

Indeed, this bound is bigger than the observed error:

```
abs(3.6 - (1.1+1.2+1.3)), 4.8eps()
```

```
(4.440892098500626e-16, 1.0658141036401502e-15)
```

1.5.2 Arithmetic and special numbers

Arithmetic works differently on `Inf` and `NaN` and for undefined operations. In particular we have:

```
1/0.0      # Inf
1/(-0.0)   # -Inf
0.0/0.0    # NaN

Inf*0      # NaN
Inf+5      # Inf
(-1)*Inf   # -Inf
1/Inf      # 0.0
1/(-Inf)   # -0.0
Inf - Inf  # NaN
```

```

Inf == Inf # true
Inf == -Inf # false

NaN*0 # NaN
NaN+5 # NaN
1/NaN # NaN
NaN == NaN # false
NaN != NaN # true

true

```

1.5.3 Special functions (advanced)

Other special functions like `cos`, `sin`, `exp`, etc. are *not* part of the IEEE standard. Instead, they are implemented by composing the basic arithmetic operations, which accumulate errors. Fortunately many are designed to have *relative accuracy*, that is, $\mathbf{s} = \sin(x)$ (that is, the Julia implementation of $\sin x$) satisfies

$$\mathbf{s} = (\sin x)(1 + \delta)$$

where $|\delta| < c\epsilon_m$ for a reasonably small $c > 0$, *provided* that $x \in \mathbb{F}^{\text{normal}}$. Note these special functions are written in (advanced) Julia code, for example, `sin`.

WARNING (`sin(fl(x))` is not always close to `sin(x)`) This is possibly a misleading statement when one thinks of x as a real number. Consider $x = \pi$ so that $\sin x = 0$. However, as $\text{fl}(\pi) \neq \pi$. Thus we only have relative accuracy compared to the floating point approximation:

```

π_6_4 = Float64(π)
π_beta = big(π_6_4) # Convert 64-bit approximation of π to higher precision. Note its
the same number.
abs(sin(π_6_4)), abs(sin(π_6_4) - sin(π_beta)) # only has relative accuracy compared to
sin(π_beta), not sin(π)

(1.2246467991473532e-16, 2.994769809718339860754263822337778811430799841054
596882794158676581342467643355e-33)

```

Another issue is when x is very large:

```

ε = eps() # machine epsilon, 2^(-52)
x = 2*10.0^100
abs(sin(x) - sin(big(x))) ≤ abs(sin(big(x))) * ε

true

```

But if we instead compute 10^{100} using `BigFloat` we get a completely different answer that even has the wrong sign!

```

xt = 2*big(10.0)^100
sin(x), sin(xt)

(-0.703969872087777, 0.6911910845037462219623751594978914260403966392716944
990360937340001300242965408)

```

This is because we commit an error on the order of roughly

[illegible]

15