I.2 Reals

Reference: [Overton (https://cs.nyu.edu/~overton/book/)](https://cs.nyu.edu/~overton/book/)

In this chapter, we introduce the [IEEE Standard for Floating-Point Arithmetic (https://en.wikipedia.org/wiki/IEEE_754)](https://en.wikipedia.org/wiki/IEEE_754). There are multiplies ways of representing real numbers on a computer, as well as the precise behaviour of operations such as addition, multiplication, etc.: one can use

1. [Fixed-point arithmetic (https://en.wikipedia.org/wiki/Fixed-point_arithmetic)](https://en.wikipedia.org/wiki/Fixed-point_arithmetic): essentially representing a real number as integer where a decimal point is inserted at a fixed point. This turns out to be impractical in most applications, e.g., due to loss of relative accuracy for small numbers.
2. [Floating-point arithmetic (https://en.wikipedia.org/wiki/Floating-point_arithmetic)](https://en.wikipedia.org/wiki/Floating-point_arithmetic): essentially scientific notation where an exponent is stored alongside a fixed number of digits. This is what is used in practice.
3. [Level-index arithmetic (https://en.wikipedia.org/wiki/Symmetric_level-index_arithmetic)](https://en.wikipedia.org/wiki/Symmetric_level-index_arithmetic): stores numbers as iterated exponents. This is the most beautiful mathematically but unfortunately is not as useful for most applications and is not implemented in hardware.

Before the 1980s each processor had potentially a different representation for floating-point 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 floating-point numbers in details:

1. Floating-point arithmetic is very precisely defined, and can even be used in rigorous computations as we shall see in the labs. But it is not exact and its important to understand how errors in computations can accumulate.
2. Failure to understand floating-point arithmetic can cause catastrophic issues in practice, with the extreme example being the [explosion of the Ariane 5 rocket (https://youtu.be/N6PWATvLQCY?t=86)](https://youtu.be/N6PWATvLQCY?t=86).

In this chapter we discuss the following:

1. Real numbers in binary: we discuss how binary digits can be used to represent real numbers.
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 (non-examinable) 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.

In [1]:
```
using SetRounding, ColorBitstring
```

1. Real numbers in binary

Reals can also be presented in binary format, that is, a sequence of `0` s and `1` s alongside a decimal point:

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

$$(B_p \ldots B_0.b_1 b_2 b_3 \ldots)_2 := (B_p \ldots B_0)_2 + \sum_{k=1}^{\infty} \frac{b_k}{2^k}.$$

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

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

We will see that in binary

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

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.

2. Floating-point numbers

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

Definition 2 (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:

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

The *normal numbers* $F^{\text{normal}}_{\sigma,Q,S} \subset \mathbb{R}$ are

$$F^{\text{normal}}_{\sigma,Q,S} := \{\pm 2^{q-\sigma} \times (1.b_1 b_2 b_3 \ldots b_S)_2 : 1 \leq q < 2^Q - 1\}.$$

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

$$F^{\text{sub}}_{\sigma,Q,S} := \{\pm 2^{1-\sigma} \times (0.b_1 b_2 b_3 \ldots b_S)_2\}.$$

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

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

where NaN is a special symbol representing "not a number", essentially an error flag.

Note this set of real numbers has no nice *algebraic structure*: it is not closed under addition, subtraction, etc. On the other hand, we can control errors effectively hence it is extremely useful for analysis.

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

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

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

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

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

$$x = \pm 2^{q-\sigma} \times (1.b_1 b_2 b_3 \ldots b_S)_2.$$

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

$$x = \pm 2^{1-\sigma} \times (0.b_1 b_2 b_3 \ldots b_S)_2.$$

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

IEEE floating-point numbers

Definition 3 (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:

$$F_{16} := F_{15,5,10}$$
$$F_{32} := F_{127,8,23}$$
$$F_{64} := F_{1023,11,52}$$

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`. Alternatively, one can use scientific notation: `1e0`. `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` (this is in fact scientific notation so `1f1 ≡ 10f0`). `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 2 (rational in `Float32`) How is the number $1/3$ stored in `Float32`? Recall that

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

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

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

and the significand bits are 01010101010101010101011. Thus the `Float32` bits for $1/3$ are:

In [2]:
```
printbits(1f0/3)
```

```
00111110101010101010101010101011
```

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

In [3]:
```
printbits(0.0)
```

```
0000000000000000000000000000000000000000000000000000000000000000
```

Unlike integers, we also have a negative zero:

In [4]:
```
printbits(-0.0)
```

```
1000000000000000000000000000000000000000000000000000000000000000
```

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

Special normal numbers

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

Definition 4 (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:

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

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

$$\max F^{\text{normal}}_{\sigma,Q,S} = 2^{2^Q-2-\sigma}(1.11\ldots)_2 = 2^{2^Q-2-\sigma}(2-\epsilon_m)$$

We confirm the simple bit representations:

In [5]:
```
σ,Q,S = 127,8,23 # Float32
εₘ = 2.0^(-S)
printlnbits(Float32(2.0^(1-σ))) # smallest positive normal Float32
printlnbits(Float32(2.0^(2^Q-2-σ) * (2-εₘ))) # largest normal Float
```

```
00000000100000000000000000000000
01111111011111111111111111111111
```

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

In [6]:
```
eps(Float32), floatmin(Float32), floatmax(Float32)
```

Out[6]:
```
(1.1920929f-7, 1.1754944f-38, 3.4028235f38)
```

Example 3 (creating a sub-normal number) If we divide the smallest normal number by two, we get a subnormal number:

In [7]:
```
mn = floatmin(Float32) # smallest normal Float32
printlnbits(mn)
printbits(mn/2)
```

```
00000000100000000000000000000000
00000000010000000000000000000000
```

Can you explain the bits?

Special numbers

The special numbers extend the real line by adding $\pm\infty$ but also a notion of "not-a-number" NaN. Whenever the bits of q of a floating-point number are all 1 then they represent an element of $F^{\rm special}$. 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):

In [8]:
```
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):

In [9]:
```
printbits(NaN16)
```

```
0111111000000000
```

These are needed for undefined algebraic operations such as:

In [10]:
```
0/0
```

Out[10]: NaN

Essentially it is a CPU's way of indicating an error has occurred.

Example 4 (many `NaN` s) What happens if we change some other b_k to be nonzero? We can create bits as a string and see:

In [11]:
```
i = 0b0111110000010001 # an UInt16
reinterpret(Float16, i)
```

Out[11]: NaN16

Thus, there are more than one `NaN` s on a computer.

3. 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:

Definition 6 (rounding) $\mathrm{fl}^{\rm up}_{\sigma,Q,S} : \mathbb{R} \rightarrow F_{\sigma,Q,S}$ denotes the function that rounds a real number up to the nearest floating-point number that is greater or equal. $\mathrm{fl}^{\rm down}_{\sigma,Q,S} : \mathbb{R} \rightarrow F_{\sigma,Q,S}$ denotes the function that rounds a real number down to the nearest floating-point number that is greater or equal. $\mathrm{fl}^{\rm nearest}_{\sigma,Q,S} : \mathbb{R} \rightarrow F_{\sigma,Q,S}$ denotes the function that rounds a real number to the nearest floating-point number. In case of a tie, it returns the floating-point number whose least significant bit is equal to zero. We use the notation fl when σ, Q, S and the rounding mode are implied by context, with $\mathrm{fl}^{\rm nearest}$ being the default rounding mode.

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` .

In [12]:
```
printlnbits(1/3)  # 64 bits
printbits(Float32(1/3))  # round to nearest 32-bit
```

```
0011111111010101010101010101010101010101010101010101010101010101
00111110101010101010101010101011
```

The default rounding mode can be changed:

In [13]:
```
printbits(Float32(1/3,RoundDown) )
```

```
00111110101010101010101010101010
```

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

In [14]:
```
x = 1f0
setrounding(Float32, RoundDown) do
    x/3
end,
setrounding(Float32, RoundUp) do
    x/3
end
```

Out[14]: (0.3333333f0, 0.33333334f0)

WARNING (compiled constants, non-examinable): Why did we first create a variable `x` instead of typing `1f0/3`? This is due to a very subtle issue where the compiler is *too clever for it's own good*: it recognises `1f0/3` can be computed at compile time, but failed to recognise the rounding mode was changed.

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

$$
\begin{aligned}
x \oplus y &:= \mathrm{fl}(x + y) \\
x \ominus y &:= \mathrm{fl}(x - y) \\
x \otimes y &:= \mathrm{fl}(x * y) \\
x \oslash y &:= \mathrm{fl}(x/y)
\end{aligned}
$$

Note also that `^` and `sqrt` are similarly exact up to rounding. Also, note that when we convert a Julia command with constants specified by decimal expansions we first round the constants to floats, e.g., `1.1 + 0.1` is actually reduced to

$$\mathrm{fl}(1.1) \oplus \mathrm{fl}(0.1)$$

This includes the case where the constants are integers (which are normally exactly floats but may be rounded if extremely large).

Example 5 (decimal is not exact) The Julia command `1.1+0.1` gives a different result than `1.2`:

In [15]:
```
x = 1.1
y = 0.1
x + y - 1.2 # Not Zero?!?
```

Out[15]: 2.220446049250313e-16

This is because $\mathrm{fl}(1.1) \neq 1 + 1/10$ and $\mathrm{fl}(1.1) \neq 1/10$ since their expansion in *binary* is not finite, but rather:

$$\mathrm{fl}(1.1) = (1.0001100110011001100110011001100110011001100110011010)_2$$

$$\begin{aligned}\mathrm{fl}(0.1) &= 2^{-4} * (1.100110011001100110011001100110011001100110011001101\\ &= (0.00011001100110011001100110011001100110011001100110011010\end{aligned}$$

Thus when we add them we get

$$\mathrm{fl}(1.1) + \mathrm{fl}(1.1) = (1.001100110011001100110011001100110011001100110011001100110$$

where the red digits indicate those beyond the 52 representable in F_{54}. In this case we round up and get

$$\mathrm{fl}(1.1) \oplus \mathrm{fl}(1.1) = (1.0011001100110011001100110011001100110011001100110$$

On the other hand,

$$\mathrm{fl}(1.2) = (1.0011001100110011001100110011001100110011001100110011)_2$$

which differs by 1 bit.

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:

In [16]:
```
(1.1 + 1.2) + 1.3, 1.1 + (1.2 + 1.3)
```

Out[16]:
```
(3.5999999999999996, 3.6)
```

Can you explain this in terms of bits?

Bounding errors in floating point arithmetic

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

Definition 7 (absolute/relative error) If $\tilde{x} = x + \delta_{\mathrm{a}} = x(1 + \delta_{\mathrm{r}})$ then $|\delta_{\mathrm{a}}|$ is called the *absolute error* and $|\delta_{\mathrm{r}}|$ 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 8 (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:

$$\mathcal{N}_{\sigma,Q,S} := \{x : \min |F^{\mathrm{normal}}_{\sigma,Q,S}| \leq |x| \leq \max F^{\mathrm{normal}}_{\sigma,Q,S}\}$$

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 1 (round bound) If $x \in \mathcal{N}$ then

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

where the *relative error* is

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

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) $|\delta_1| \leq \frac{\epsilon_{\text{m}}}{2}$.

Example 6 (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

$$(\text{fl}(1.1) \oplus \text{fl}(1.2)) = (1.1(1 + \delta_1) + 1.2(1 + \delta_2))(1 + \delta_3) = 2.3 + \underbrace{1.1\delta_1 + 1.2\delta_2 + 2}_{}$$

In this module we will never ask for precise bounds: that is, we will always want bounds of the form $C\epsilon_{\text{m}}$ for a specified constant C but the choice of C need not be sharp. Thus we will tend to round up to integers. Further, while $\delta_1\delta_3$ and $\delta_2\delta_3$ are absolutely tiny we will tend to bound them rather naïvely by $|\epsilon_{\text{m}}/2|$. Using these rules we have the bound

$$|\delta_4| \leq (1 + 1 + 2 + 1 + 1)\epsilon_{\text{m}} = 6\epsilon_{\text{m}}$$

Thus the computation becomes

$$((2.3 + \delta_4) + 1.3(1 + \delta_5))(1 + \delta_6) = 3.6 + \underbrace{\delta_4 + 1.3\delta_5 + 3.6\delta_6 + \delta_4\delta_6 + 1.3\delta_5\delta_6}_{\delta_7}$$

where the *absolute error* is

$$|\delta_7| \leq (6 + 1 + 2 + 1 + 1)\epsilon_{\text{m}} = 11\epsilon_{\text{m}}$$

Indeed, this bound is bigger than the observed error:

In [17]:
```
abs(3.6 - (1.1+1.2+1.3)), 11eps()
```

Out[17]:
```
(4.440892098500626e-16, 2.4424906541753444e-15)
```

Arithmetic and special numbers

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

In [18]:
```
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
```

Out[18]:
```
true
```

Special functions (non-examinable)

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, `s = sin(x)` (that is, the Julia implementation of $\sin x$) satisfies

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

where $|\delta| < c\epsilon_{\rm m}$ for a reasonably small $c > 0$, *provided* that $x \in F^{\rm normal}$. Note these special functions are written in (advanced) Julia code, for example, [sin (https://github.com/JuliaLang/julia/blob/d08b05df6f01cf4ec6e4c28ad94cedda76cc62e8/ba](https://github.com/JuliaLang/julia/blob/d08b05df6f01cf4ec6e4c28ad94cedda76cc62e8/ba)

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 ${\rm fl}(\pi) \neq \pi$. Thus we only have relative accuracy compared to the floating point approximation:

In [19]:
```
π₆₄ = Float64(π)
πᵦ = big(π₆₄) # Convert 64-bit approximation of π to higher precisi
abs(sin(π₆₄)), abs(sin(π₆₄) - sin(πᵦ)) # only has relative accuracy
```

Out[19]:
```
(1.2246467991473532e-16, 2.994769809718339860754263822337778811430
7998410545968827941586765813424676433355e-33)
```

Another issue is when x is very large:

In [20]:
```
ε = eps() # machine epsilon, 2^(-52)
x = 2*10.0^100
abs(sin(x) - sin(big(x)))  ≤  abs(sin(big(x))) * ε
```

Out[20]:
```
true
```

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

In [21]:
```
x̃ = 2*big(10.0)^100
sin(x), sin(x̃)
```

Out[21]:
```
(-0.703969872087777, 0.6911910845037462219623751594978914260403966
392716944990360937340001300242965408)
```

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

$$2 * 10^{100} * \epsilon_{\rm m} \approx 4.44 * 10^{84}$$

when we round $2 * 10^{100}$ to the nearest float.

Example 7 (polynomial near root) For general functions we do not generally have relative accuracy. For example, consider a simple polynomial $1 + 4x + x^2$ which has a root at $\sqrt{3} - 2$. But

In [22]:
```
f = x -> 1 + 4x + x^2
x = sqrt(3) - 2
abserr = abs(f(big(x)) - f(x))
relerr = abserr/abs(f(x))
abserr, relerr # very large relative error
```

Out[22]:
```
(6.808194126854568545271553503125001640528110233296921194323658710
345625877380371e-19, 0.001962328354097166993597056716680526733398
4375000000000000000000000000000000008)
```

We can see this in the error bound (note that $4x$ is exact for floating point numbers and adding 1 is exact for this particular x):
$(x \otimes x) \oplus 4x + 1 = (x^2(1 + \delta_1) + 4x)(1 + \delta_2) + 1 = x^2 + 4x + 1 + \delta_1 x^2 + 4x\delta_2 +$
Using a simple bound $|x| < 1$ we get a (pessimistic) bound on the absolute error of $3\epsilon_{\rm m}$. Here `f(x)` itself is less than $2\epsilon_{\rm m}$ so this does not imply relative accuracy. (Of course, a bad upper bound is not the same as a proof of inaccuracy, but here we observe the inaccuracy in practice.)

4. High-precision floating-point numbers (non-examinable)

It is possible to set the precision of a floating-point number using the `BigFloat` type, which results from the usage of `big` when the result is not an integer. For example, here is an approximation of 1/3 accurate to 77 decimal digits:

In [23]:
```
big(1)/3
```

Out[23]:
```
0.3333333333333333333333333333333333333333333333333333333333333333
333333333333348
```

Note we can set the rounding mode as in `Float64`, e.g., this gives (rigorous) bounds on `1/3`:

In [24]:
```
setrounding(BigFloat, RoundDown) do
  big(1)/3
end, setrounding(BigFloat, RoundUp) do
  big(1)/3
end
```

Out[24]:
```
(0.333333333333333333333333333333333333333333333333333333333333333
3333333333333305, 0.3333333333333333333333333333333333333333333333
3333333333333333333333333333333348)
```

We can also increase the precision, e.g., this finds bounds on `1/3` accurate to more than 1000 decimal places:

In [25]:
```julia
setprecision(4_000) do # 4000 bit precision
  setrounding(BigFloat, RoundDown) do
    big(1)/3
  end, setrounding(BigFloat, RoundUp) do
    big(1)/3
  end
end
```

Out[25]:
```
(0.333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333308, 0.33333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333346)
```

In the labs we shall see how this can be used to rigorously bound e, accurate to 1000 digits.