I.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.
3. 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*.
4. 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.
5. 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.

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

1. Binary representation

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

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

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

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

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

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

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

Example (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. 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: $(B_{p-1} \ldots B_1 B_0)_2$.

Integers on a computer follow modular arithmetic:

Definition (ring of integers modulo m) Denote the ring

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

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 (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:

In [2]:
```
x = UInt8(255)
y = UInt8(1)
printbits(x); println(" + "); printbits(y); println(" = ")
printbits(x + y)
```

```
11111111 +
00000001 =
00000000
```

Example (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:

In [3]:
```
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
```

Signed integer

Signed integers use the Two's complemement 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 (converting bits to signed integers) What 8-bit integer has the bits `01001001`? Adding the corresponding decimal places we get:

In [4]:
```
2^0 + 2^3 + 2^6
```

Out[4]:
```
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`:

In [5]:
```
2^0 + 2^3 + 2^6 + 2^7 - 2^8
```

Out[5]:
```
-55
```

We can check the results using `printbits`:

In [6]:
```
printlnbits(Int8(73))
printbits(-Int8(55))
```

```
01001001
11001001
```

Arithmetic works precisely the same for signed and unsigned integers.

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

In [7]:
```
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:

In [8]:
```
typemax(Int8) + Int8(1) # returns typemin(Int8)
```

Out[8]:
```
-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.

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 `BigInt`s, which we can create by calling `big` on an integer:

In [9]:
```
x = typemax(Int64) + big(1) # Too big to be an `Int64`
```

Out[9]:
```
9223372036854775808
```

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

In [10]:
```
x^100
```

Out[10]:
```
30829940252776347457001068215456657213717985333056974588553422779210937319844764047059665394124108982405617299123720385012288931419210801524046423937765990772944340615199054241246013942269436014309164343837147167247202273315969506137016610345489483887210976672754387637581285084032971994582602777073012024609800938184141670805633427614823958624351850939424435407223631517700222217832439595925313360629984942099147524080190607208051245343826460510936138148486460620386624234875043260443612037084304893058642343338014014547140023376295718383390360728662900230671437151716615826286842267917560749586018165739492101920429719261285640125596833063891562865262157026023955919873792846823095854484520920509345944712871675691790827690907778485058829248588945681685288179787963931181062068092463984296225973082494056307958089189726701678735576365394146232076917088075949053636690459581128773097212746967276496496010810878000638239143750075543163240049874489986642327436441234458040254480825038220479904594615300602390556385799245276805580024937804723029319565942013515817048714543455250235208789745701165279569026248145395218985062991831707830217974393158466067785199581037714968820628241051867119832966361530047910339065726550260741036716100932205969655083257714244071120221654679340461084001560321676025443801248355439305974923873624147980728110581452806109011739005060060604228087667499288851218705078807364237925455813890575257569981450090997117697469299234094394984840574025401463942099019413361096233909056117427663439764954916401592565651111571414769257187704568268701243082044838400201357613851006471104244828842270232637747398962711875413488415772647088571125272932490717217468263604683325933469555629785507020775366368002753612709901526248456328209643292122899677436613886360765877886748185299249994921843183573130403496311896614949399409796011301191280067209053259341918813967552543176532349157376
```

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 `x^x` (at your own risk).

Division

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

In [11]:
```
5 ÷ 2 # equivalent to div(5,2)
```

Out[11]:
```
2
```

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

In [12]:
```
5 / 2 # alternatively 2 \ 5
```

Out[12]:
```
2.5
```

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

In [13]:
```
(1//2) + (3//4)
```

Out[13]:
```
5//4
```

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

In [14]:
```
big(102324)//132413023 + 23434545//4243061 + 23434545//42430534435
```

Out[14]:
```
263390378350076484775415 40//476780487870754436459646 1
```

3. Floating-point numbers

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

Definition (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 defined by

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

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

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

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

$$sq_{Q-1}\dots q_0b_1\dots b_S$$

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.

IEEE floating-point numbers

Definition (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`. `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 limitted 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 (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

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

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

In [15]:
```
printbits(1f0/3)
```

```
00111111010101010101010101011
```

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

In [16]:
```
printbits(0.0)
```

```
0000000000000000000000000000000000000000000000000000000000000000
```

Unlike integers, we also have a negative zero:

In [17]:
```
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 (machine epsilon/smallest positive normal number/largest normal number) *Machine epsilon* is denoted

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

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:

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

where $|A| :=\{|x| : x \backslash in\ A\ \}. The_l argest(positive) normal number_i s$
$\max F^{\mathrm{normal}}_{\sigma,Q,S} = 2^{2^Q-2-\sigma}(1.11\ldots1)_2 = 2^{2^Q-2-\sigma}(2-\epsilon_{\mathrm{m}})\$$

We confirm the simple bit representations:

In [18]:
```
σ,Q,S = 127,23,8 # Float32
εₘ = 2.0^(-S)
println bits(Float32(2.0^(1-σ))) # smallest positive Float32
println bits(Float32(2.0^(2^Q-2-σ) * (2-εₘ))) # largest Float32
```

```
00000000100000000000000000000000
01111111000000000000000000000000
```

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

In [19]:
```
eps(Float32),floatmin(Float32),floatmax(Float32)
```

Out[19]:
```
(1.1920929f-7, 1.1754944f-38, 3.4028235f38)
```

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

In [20]:
```
mn = floatmin(Float32) # smallest normal Float32
println bits(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".

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

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

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):

In [21]:
```
println bits(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 [22]:
```
printbits(NaN16)
```

```
0111111000000000
```

These are needed for undefined algebraic operations such as:

In [23]:
```
0/0
```

Out[23]:
```
NaN
```

Example (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 [24]:
```
i = parse(UInt16, "0111110000010001"; base=2)
reinterpret(Float16, i)
```

Out[24]:
```
NaN16
```

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

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:

Definition (rounding) $\mathrm{fl}^{\mathrm{up}}_{\sigma,Q,S} : \mathbb{R} \to F_{\sigma,Q,S}$ denotes the function that rounds a real number up to the nearest floating-point number that is greater or equal. $\mathrm{fl}^{\mathrm{down}}_{\sigma,Q,S} : \mathbb{R} \to F_{\sigma,Q,S}$ denotes the function that rounds a real number down to the nearest floating-point number that is greater or equal. $\mathrm{fl}^{\mathrm{nearest}}_{\sigma,Q,S} : \mathbb{R} \to F_{\sigma,Q,S}$ 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 $\mathrm{fl}^{\mathrm{nearest}}$ 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.)

WARNING (rounding performance, advanced) These rounding modes are part of the FPU instruction set so will be (roughly) equally fast as the default, `RoundNearest`. Unfortunately, changing the rounding mode is expensive, and is not thread-safe.

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

In [25]:
```
printlnbits(1/3)  # 64 bits
printbits(Float32(1/3))  # round to nearest 32-bit
```

```
00111111110101010101010101010101010101010101010101010101010101010101
00111110101010101010101010101011
```

The default rounding mode can be changed:

In [26]:
```
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 [27]:
```
x = 1f0
setrounding(Float32, RoundDown) do
    x/3
end,
setrounding(Float32, RoundUp) do
    x/3
end
```

Out[27]:
```
(0.3333333f0, 0.33333334f0)
```

WARNING (compiled constants, advanced): 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 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.

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

In [28]:
```
x = 1.1
y = 0.1
x + y - 1.2 # Not Zero?!?
```

Out[28]:
```
2.220446049250313e-16
```

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

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

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:

In [29]:
```
(1.1 + 1.2) + 1.3, 1.1 + (1.2 + 1.3)
```

```
(3.5999999999999996, 3.6)
```

Out[29]:

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 (absolute/relative error) If $\tilde{x} = x + \delta_{rma} = x(1 + \delta_{\mathrm{r}})$ 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 (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

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

where the *relative error* is

$$|\delta_x^{\mathrm{nearest}}| \leq \frac{\epsilon_{\mathrm{m}}}{2}$$
$$|\delta_x^{\mathrm{up/down}}| < \epsilon_{\mathrm{m}}.$$

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

Example (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 $\mathrm{fl}(1.1)$. Thus this computation becomes

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

First we find

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

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

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

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

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

Indeed, this bound is bigger than the observed error:

In [30]:
```
abs(3.6 - (1.1+1.2+1.3)), 4.8eps()
```

Out[30]:
```
(4.440892098500626e-16, 1.0658141036401502e-15)
```

Arithmetic and special numbers

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

In [31]:
```
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[31]:
```
true
```

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

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

where $|\delta| < c\epsilon_{\mathrm{m}}$ for a reasonably small $c > 0$, *provided* that $x \in \mathrm{F}^{\mathrm{normal}}$. 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

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

In [32]:
```
π₆₄ = Float64(π)
πᵦ = big(π₆₄) # Convert 64-bit approximation of π to higher precision. Note its the
abs(sin(π₆₄)), abs(sin(π₆₄) - sin(πᵦ)) # only has relative accuracy compared to sin(
```

Out[32]:
```
(1.2246467991473532e-16, 2.99476980971833986075426382233777881143079984105459688279415867658134246764335
5e-33)
```

Another issue is when x is very large:

In [33]:
```
ε = eps() # machine epsilon, 2^(-52)
x = 2*10.0^100
abs(sin(x) - sin(big(x)))  ≤  abs(sin(big(x))) * ε
```

Out[33]:
```
true
```

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

In [34]:
```
x̃ = 2*big(10.0)^100
sin(x), sin(x̃)
```

Out[34]:
```
(-0.703969872087777, 0.691191084503746221962375159497891426040396639271694499036093734000130024296540
8)
```

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

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

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

Example (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 [35]:
```
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[35]:
```
(6.80819412685456854527155350312500164052811023329692119432365871034562587738037 1e-19, 0.0019623283540971669935970567166805267333984375000000000000000000000000000000000000000000000008)
```

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):

$$(x \otimes x \oplus 4x) + 1 = (x^2(1+\delta_1) + 4x)(1+\delta_2) + 1 = x^2 + 4x + 1 + \delta_1 x^2 + 4x\delta_2 + x^2\delta_1\delta_2$$

Using a simple bound $|x| < 1$ we get a (pessimistic) bound on the absolute error of $3\epsilon_{\mathrm{m}}$. Here `f(x)` itself is less than $2\epsilon_{\mathrm{m}}$ 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.)

5. High-precision floating-point numbers (advanced)

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 [36]:

```
big(1)/3
```

Out[36]:

```
0.3333333333333333333333333333333333333333333333333333333333333333333333333333348
```

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

In [37]:

```
setrounding(BigFloat, RoundDown) do
  big(1)/3
end, setrounding(BigFloat, RoundUp) do
  big(1)/3
end
```

Out[37]:

```
(0.3333333333333333333333333333333333333333333333333333333333333333333333333333305,
0.3333333333333333333333333333333333333333333333333333333333333333333333333333348)
```

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

In [38]:

```
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[38]:

```
(0.333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333308, 0.33333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
333333333333333333333333333333333333333333333333333333333333333333333333333333333333
3333333333333333333333333333333333333333333333333333333333333333346)
```

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