Scientific Computing TU Berlin Winter 2021/22 © Jürgen Fuhrmann
Notebook 08

```
begin
    using PlutoUI
    using PlutoVista
    using BenchmarkTools
end
```

Number representation

Number representation

Besides of the concrete names of Julia library functions everything in this chapter is valid for most modern programming languagues and computer systems.

All data in computers are stored as sequences of bits. In Julia, for concrete number types, the `bitstring` function returns this information as a sequence of `0` and `1`. The `sizeof` function returns the number of bytes in the binary representation.

Integer numbers

```
T_int = Int8
```

```
T_int=Int8
```

```
i = 1
```

```
i=T_int(1)
```

```
1
```

```
sizeof(i)
```

```
"00000001"
```

```
bitstring(i)
```

Positive integer numbers are represented by their representation in the binary system. For negative numbers n, the binary representation of their "two's complement" $2^N - |n|$ (where N is the number of available bits) is stored. Correspondingly, the sign of an integer number is stored in the first bit.

`typemin` and `typemax` return the smallest and largest numbers which can be represented in number type.

```
(-128, 127)
```

```
typemin(T_int),typemax(T_int)
```

Unless the possible range of the representation $(-2^{N-1}, 2^{N-1})$ is exceeded, addition, multiplication and subtraction of integers are exact. If it is exceeded, operation results wrap around into the opposite sign region.

```
10
```

```
3+7
```

```
-119
```

```
typemax(T_int)+T_int(10)
```

```
3//10
```

```
1//10 + 2//10
```

Floating point numbers

```
0.30000000000000004
```

```
0.1+0.2
```

But this should be 0.3. What is happening ???

Decimal representation of $x \in \mathbb{R}$

Usually we write real numbers as decimal fractions and cut the representation off if the number of digits is too large or infinite.

Any real number $x \in \mathbb{R}$ can be expressed via the **representation formula**

$$x = \pm \sum_{i=0}^{\infty} d_i \beta^{-i} \beta^e$$

with **base** $\beta \in \mathbb{N}, \beta \geq 2$, **significand** (or **mantissa**) digits $d_i \in \mathbb{N}, 0 \leq d_i < \beta$ and **exponent** $e \in \mathbb{Z}$.

This representation is infinite for periodic decimal numbers and irrational numbers.

Scientific notation

The scientific notation of real numbers is derived from the representation formula in the case of $\beta = 10$. Let e.g. $x = 6.022 \cdot 10^{23}$= `6.022e23` . Then

- $\beta = 10$
- $d = (6, 0, 2, 2, 0 \ldots)$
- $e = 23$

This representation is not unique, e.g. $x_1 = 0.6022 \cdot 10^{24}$= `0.6022e24` $= x$ with

- $\beta = 10$
- $d = (0, 6, 0, 2, 2, 0 \ldots)$
- $e = 24$

IEEE754 standard

This is the actual standard format for storing floating point numbers. It was developed in the 1980ies. It is derived as well from the representation formula

- $\beta = 2$, therefore $d_i \in \{0, 1\}$
- Parameters:
  - t is the **significand** (mantissa) length
  - k: **exponent size** – number of bits for exponent
- Truncation to fixed size: $x = \pm \sum_{i=0}^{t-1} d_i 2^{-i} 2^e$
- Normalization: assume $d_0 = 1 \Rightarrow$ save one bit for the storage of the significand. This saved bit is also called "hidden bit" This requires a normalization step after operations which adjusts significand and exponent of the result.
- Exponent range: $L := -2^k + 1 \leq e \leq = 2^k - 1 =: U$. Actually, $e + 2^{k-1} - 1$ is stored
- Extra bit for sign
- $\Rightarrow$ storage size: $(t - 1) + k + 1$
- The storage sequence is: Sign bit, exponent, mantissa.
- Standardized for most modern languages
- Hardware support usually for 64bit and 32bit

precision	Julia	C/C++	k	t	bits
quadruple	n/a	long double	16	113	128
double	Float64	double	11	53	64
single	Float32	float	8	24	32
half	Float16	n/a	5	11	16

- See also the **Julia Documentation on floating point numbers**, **0.30000000000000004.com**, **wikipedia** and the links therein.

Storage layout for a normalized Float32 number ($d_0 = 1$):

- bit 1: sign, $0 \to +, \quad 1 \to -$
- bit $2 \ldots 9$: $k = 8$ exponent bits
  - the value $e + 2^{k-1} - 1 = e + 127$ is stored $\Rightarrow$ no need for sign bit in exponent
- bit $10 \ldots 32$: $23 = t - 1$ mantissa bits $d_1 \ldots d_{23}$
- $d_0 = 1$ not stored $\equiv$ "hidden bit"

Prettyprintinig bit representations

Julia allows to obtain the signifcand and the exponent of a floating point number

```
x0 = 100.0
```

```
x0=100.0
```

```
(1.5625, 6)
```

```
significand(x0),exponent(x0)
```

Calculate k – the length of the exponent – from the maximum representable floating point number by taking the base-2 logarithm of its exponent:

```
exponent_length(T::Type{<:AbstractFloat})=Int(log2(exponent(floatmax(T))+1)+1);
```

t – the size of the significand – is calculated from the overall size of the representation minus the size of the exponent and the size of the sign bit + 1 for the "hidden bit".

```
significand_length(T::Type{<:AbstractFloat})=8*sizeof(T)-exponent_length(T)-1+1;
```

This allows to define a more readable variant of the bitstring representation for floats.

The sign bit is the first bit in the representation:

```
signbit(x::AbstractFloat)=bitstring(x)[1:1];
```

Next comes the exponent:

```
exponent_bits (generic function with 1 method)
```

```
exponent_bits(x::AbstractFloat)=bitstring(x)[2:exponent_length(typeof(x))+1]
```

And finally, the significand:

```
significand_bits(x::AbstractFloat)=bitstring(x)
[exponent_length(typeof(x))+2:8*sizeof(x)];
```

Put them together:

```
floatbits(x::AbstractFloat)=signbit(x)*"_"*exponent_bits(x)*"_"*significand_bits(x
);
```

Julia floating point types

```
T = Float64
```

```
T=Float64
```

```
x = 0.1
```

```
x=T(0.1)
```

- Binary representation:
  0_01111111011_1001100110011001100110011001100110011001100110011010
- Exponent e=-4
  - stored: e+1023 = 1019
- $d_0 = 1$ assumed implicitely.

Numbers which are exactly represented in the decimal system may not be exactly represented in the binary system! Such numbers are always rounded to a finite (in the binary system) approximation.

```
x_per = 0.30000000000000004
```

```
x_per=T(0.1)+T(0.2)
```

```
"0_01111111101_0011001100110011001100110011001100110011001100110100"
```

```
floatbits(x_per)
```

Floating point limits

Finite size of representation $\Rightarrow$ there are minimal and maximal possible numbers which can be represented

Smallest positive denormalized number: $d_i = 0, i = 0 \ldots t-2, d_{t-1} = 1 \Rightarrow x_{min} = 2^{1-t}2^L$

```
(5.0e-324, "0_00000000000_0000000000000000000000000000000000000000000000000001")
```

```
nextfloat(zero(T)), floatbits(nextfloat(zero(T)))
```

Smallest positive normalized number: $d_0 = 1, d_i = 0, i = 1 \ldots t-1 \Rightarrow x_{min} = 2^L$

```
(2.22507e-308, "0_00000000001_0000000000000000000000000000000000000000000000000000")
```

```
floatmin(T),floatbits(floatmin(T))
```

Largest positive normalized number: $d_i = 1, 0 \ldots t-1 \Rightarrow x_{max} = 2(1-2^{1-t})2^U$

```
(1.79769e308, "0_11111111110_1111111111111111111111111111111111111111111111111111")
```

```
floatmax(T), floatbits(floatmax(T))
```

Largest representable number:

```
(Inf, "0_11111111111_0000000000000000000000000000000000000000000000000000", 1.79769e3
```

```
typemax(T),floatbits(typemax(T)),prevfloat(typemax(T)),
floatbits(prevfloat(typemax(T)))
```

Machine precision

- There cannot be more than 2^{t+k} floating point numbers $\Rightarrow$ almost all real numbers have to be approximated
- Let x be an exact value and $\tilde{x}$ be its approximation. Then $|\frac{\tilde{x}-x}{x}| < \epsilon$ is the best accuracy estimate we can get, where
  - $\epsilon = 2^{1-t}$ (truncation)
  - $\epsilon = \frac{1}{2}2^{1-t}$ (rounding)
- Also: ϵ is the smallest representable number such that $1 + \epsilon > 1$.
- Relative errors show up in particular when
  - subtracting two close numbers
  - adding smaller numbers to larger ones

How do operations work?

E.g. Addition

- Adjust exponent of number to be added:
  - Until both exponents are equal, add 1 to exponent, shift mantissa to right bit by bit
- Add both numbers
- Normalize result

The smallest number one can add to 1 can have at most t bit shifts of normalized mantissa until mantissa becomes 0, so its value must be 2^{-t}.

Machine epsilon

- Smallest floating point number ϵ such that $1 + \epsilon > 1$ in floating point arithmetic
- In exact math it is true that from $1 + \varepsilon = 1$ it follows that $0 + \varepsilon = 0$ and vice versa. In floating point computations this is not true

```
ϵ = 2.220446049250313e-16
```

```
ϵ=eps(T)
```

```
"0_01111001011_0000000000000000000000000000000000000000000000000000"
```

```
floatbits(ϵ)
```

```
(1.0, "0_01111111111_0000000000000000000000000000000000000000000000000000", "0_011111
```

```
one(T)+ϵ/2,floatbits(one(T)+ϵ/2), floatbits(one(T))
```

```
(1.0, "0_01111111111_0000000000000000000000000000000000000000000000000001")
```

```
 one(T)+ϵ,floatbits(one(T)+ϵ)
```

```
(2.22045e-16, "0_01111001011_0000000000000000000000000000000000000000000000000000")
```

```
nextfloat(one(T))-one(T),floatbits(nextfloat(one(T))-one(T))
```

BigFloat

- the `BigFloat` type wraps the GNU multiprecision library and allows for floating point operations with high accuracy
- one can control its precision
- However it cannot be exact: periodic dual representations must be cut off nevetheless…
- Here, we just use the default precision

```
1.727233711018888925077270372560079914223200072887256277004740694033718360632485e-77
```

```julia
eps(BigFloat)
```

```
0.2000000000000000000000000000000000000000000000000000000000000000000000000000004
```

```julia
abig=big"0.1"; bbig=big"0.2"
```

```
0.3000000000000000000000000000000000000000000000000000000000000000000000000000017
```

```julia
abig+bbig
```

Calculations with BigFloat are expensive

```julia
a64=0.1; b64=0.2;
```

```
BenchmarkTools.Trial: 10000 samples with 1000 evaluations.
 Range (min … max):  0.015 ns … 20.957 ns  ┊ GC (min … max): 0.00% … 0.00%
 Time  (median):     0.017 ns              ┊ GC (median):    0.00%
 Time  (mean ± σ):   0.020 ns ±  0.212 ns  ┊ GC (mean ± σ):  0.00% ± 0.00%
```

```
  0.015 ns        Histogram: frequency by time        0.028 ns <

 Memory estimate: 0 bytes, allocs estimate: 0.
```

```julia
@benchmark $a64+$b64
```

```
BenchmarkTools.Trial: 10000 samples with 987 evaluations.
 Range (min … max):  51.370 ns …   3.425 μs  ┊ GC (min … max): 0.00% … 97.93%
 Time  (median):     53.698 ns               ┊ GC (median):    0.00%
 Time  (mean ± σ):   62.262 ns ± 125.440 ns  ┊ GC (mean ± σ):  9.77% ±  4.76%
```

```
  51.4 ns       Histogram: log(frequency) by time      99.1 ns <

 Memory estimate: 112 bytes, allocs estimate: 2.
```

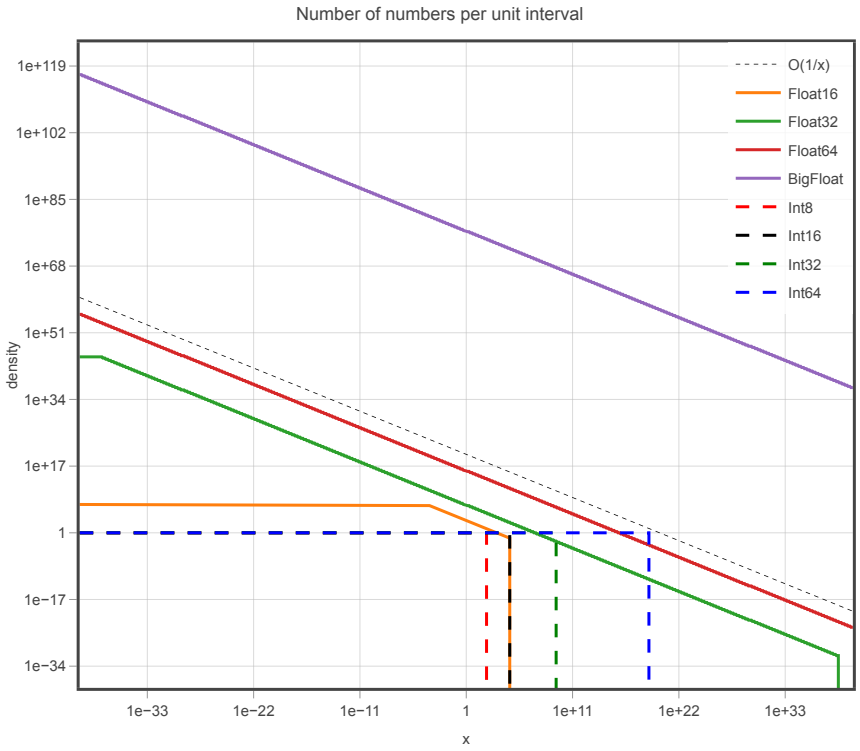
```julia
@benchmark $abig+$bbig
```

Density of numbers

How dense are numbers on the real axis?

```julia
function dens(x, ::Type{T}; sample_size=1000) where T<:AbstractFloat
    xleft=T(x)
    xright=T(x)
    for i=1:sample_size
        xleft=prevfloat(xleft)
        xright=nextfloat(xright)
    end
    return Float64(prevfloat(2.0*sample_size/(xright-xleft)))
end;
```

```julia
dens(x, ::Type{T}) where T<:Integer = x < typemax(T) ? 1 : 0 ;
```




```
let
    X=10.0.^collect(-40:0.1:40)
    p=plot(resolution=(600,500),
        title="Number of numbers per unit interval",
        xscale=:log,yscale=:log,xlabel="x",ylabel="density",legend=:rt)
    plot!(p,X,map(x->1.0e20/x,X),
label="O(1/x)",linewidth=0.5,color=:black,linestyle=:dot)
    plot!(p,X,dens.(X,Float16),label="Float16")
    plot!(p,X,dens.(X,Float32),label="Float32")
    plot!(p,X,dens.(X,Float64),label="Float64")
    plot!(p,X,dens.(X,BigFloat),label="BigFloat")
    plot!(p,X,dens.(X,Int8),label="Int8",linestyle=:dash,color=:red)
    plot!(p,X,dens.(X,Int16),label="Int16",linestyle=:dash,color=:black)
    plot!(p,X,dens.(X,Int32),label="Int32",linestyle=:dash,color=:green)
    plot!(p,X,dens.(X,Int64),label="Int64",linestyle=:dash,color=:blue)
 end
```