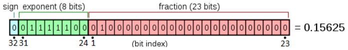
Algorithms and Data Structures

Encoding
Machine numbers for $\mathbb{R}$



Learning goals

- IEEE 745
- Types in C
- Floating point numbers in $\mathbb{R}$
- Distance

REALS ON A MACHINE

- Floating point numbers on a machine don't correspond to $\mathbb{R}$ in a mathematical sense. They are merely an approximation.
- Because there is only a finite number of machine numbers, there are no arbitrarily small or large numbers and also no arbitrarily close numbers.
- A finite subset of the real numbers cannot be closed w.r.t. rational operations (+,-,*,:)

We would like to have:

- Operations as similar as possible to those in $\mathbb{R}$,
- fast and easy implementation on digital computers.

MACHINE NUMBERS FOR $\mathbb{R}$: IEEE 754

IEEE (**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers) 754 defines standard representations for floating point numbers in computers.

Characterized by:

- Sign bit $S \in \{-1, +1\}$
- Base $b > 1$, common are 2, 8, 10 and 16 (usually 2)
- Mantissa of length m, the significant bits / digits
- Smallest and largest exponent $e_{\text{min}} < 0$ and $e_{\text{max}} > 0$
- Mantissa, exponent and S are coded as bits u_i

Representation:

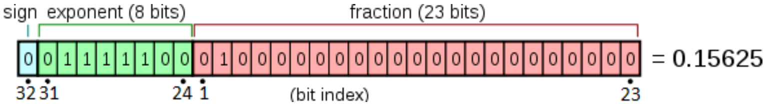
$$x = S \cdot b^e \cdot (1 + \sum_{i=1}^{m} u_i b^{-i})$$

Thus: a sign bit, an exponent and the significant digits (binary coded with mantissa bits u_i, $i = 1, ..., m$).

IEEE 754

Single precision, 32 bit:

- $b = 2$; u_{32}: sign bit
- e is 8 bits $u_{24}, \ldots, u_{31}$ (excess coding with bias 127)
- $m = 23$, the first 23 bits are used for the mantissa.



Source: `https://en.wikipedia.org/wiki/Single-precision_floating-point_format`

Converter: `https://www.h-schmidt.net/FloatConverter/IEEE754.html`

Example:

- sign bit $u_{32} = 0 \Rightarrow S = +1$
- $e = \sum_{i=1}^{8} u_{i+23} 2^{i-1} - 127 = 2^2 + 2^3 + 2^4 + 2^5 + 2^6 - 127 = -3$
- mantissa bits: $u_2 = 1, u_1 = u_3 = \ldots = u_{23} = 0$

$\Rightarrow x = S \cdot b^e \cdot (1 + \sum_{i=1}^{23} u_i b^{-i}) = 1 \cdot 2^{-3} \cdot (1 + 2^{-2}) = 0.15625$

IEEE 754 / 2

Double precision, 64 bit:

- $b = 2$; u_{64}: sign bit
- e is 11 bits $u_{53}, \ldots, u_{63}$ (excess coding with bias 1023)
- $m = 52$

Other representations:
In addition to single and double precision, IEEE 754 also has single extended and double extended. Here only a minimum number of bits is required - the exact number of bits is the implementor's choice.

IEEE 754 / 3

Normalized Number:
To guarantee a unique representation, most systems require that the first bit of the mantissa is $\neq 0$. In case of $b = 2$, the first bit of the mantissa does not need to be stored in a normalized representation. Hence, one gains one extra bit of precision (hidden bit).

A number is considered normalized if at least one exponent bit is 1.

$$x = S \cdot b^e \cdot (1 + \sum_{i=1}^{m} u_i b^{-i})$$

IEEE 754 / 4

Special cases:

- **0**: If all mantissa bits and all exponent bits are 0, then $x = \pm 0$.
- ∞: If all mantissa bits are 0 and all exponent bits are 1, then $x = \pm\infty$. This results from the division by 0, or if the result is too large or too small.
- **NaN**: If all exponent bits are 1 and at least one mantissa bit is 1, then $x =$ NaN ("Not a Number"). E.g.: $0/0$ or $\infty - \infty$.

If all exponent bits are 0, a **denormalized** number is stored. The mantissa before the "decimal" point is then 0. In this case:

$$x = S \cdot b^{e_{\min}} \cdot \left(\sum_{i=1}^{m} u_i b^{-i} \right)$$

This allows very small numbers close to 0.

IEEE 754 / 5

Binary representation of smallest and largest numbers (IEEE 754, single precision, 32 bit):

	Sign Bit	Exponent bits	Mantissa bits
Smallest number (normalized)	0	00000001	00000000000000000000000
Smallest number (denormalized)	0	00000000	00000000000000000000001
Largest number	0	11111110	11111111111111111111111

The corresponding values in the decimal system:

	exact value	scient. notation
Smallest number (normalized)	2^{-126}	$1.175494 \cdot 10^{-38}$
Smallest number (denormalized)	$2^{-126} \cdot 2^{-23}$	$1.401298 \cdot 10^{-45}$
Largest number	$2^{127} \cdot (1 + \sum_{i=1}^{23} 2^{-i})$	$3.4028235 \cdot 10^{38}$

TYPES IN `C` (PROGRAMMING LANGUAGE)

Most programming languages provide several fixed-point and floating point representations. `C` has:

Fixed: signed short int, unsigned short int, signed long int, . . .

Float: float, double, long double

The compiler translates them for the CPU. Standard PCs (usually) have hardware support for floating-point arithmetic in single and double accuracy. CPUs of different architecture (with the same nominal clock rate) can strongly differ in computing power.

FLOATING POINT NUMBERS IN R

- By default, `R` displays 6 decimal places. This can be adjusted using the command `options(digits = m)`.
- Internally, `R` calculates all floating point operations in double precision (IEEE 754, double precision, 64 bit).
- `.Machine` contains all information about the encoding

FLOATING POINT NUMBERS IN R / 2

```
.Machine$double.base      # base
## [1] 2

.Machine$double.digits    # number of mantissa bits
## [1] 53

.Machine$double.exponent  # number of exponent bits
## [1] 11

.Machine$double.xmin      # smallest float
## [1] 2.225074e-308

.Machine$double.xmax      # largest float
## [1] 1.797693e+308
```

FLOATING POINT NUMBERS IN R / 3

```
0.1 + 0.2 == 0.3
## [1] FALSE
```

- `sprintf` (wrapper for the corresponding `C` function) outputs a formatted string. Both numbers cannot be represented exactly (hence, also not their sum):

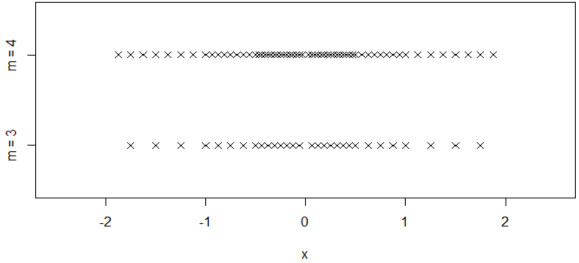
```
sprintf("%.20f", 0.1)    # decimal notation (20 digits)
## [1] "0.10000000000000000555"

sprintf("%.20f", 0.2)
## [1] "0.20000000000000001110"

sprintf("%.20f", 0.1 + 0.2)
## [1] "0.30000000000000004441"
```

- This problem can be avoided by using the comparison **with tolerance** `all.equal` instead of the exact comparison ==.

```
all.equal(0.1 + 0.2, 0.3)
## [1] TRUE
```

DISTANCE

The machine floats $\mathcal{M}$ are **not uniformly distributed** in the domain. The interval $[b^{i-1}, b^i]$ contains the same quantity of numbers as the interval $[b^i, b^{i+1}]$, even though the latter is b times as big.



DISTANCE / 2

The distance between the representable numbers is important:

- The smallest numbers around 0 are $\pm b^{e_{\min} - m}$.
- The smallest number greater than 1 is $1 + b^{-m}$
  ($e = 0$, $u_0 = \ldots = u_{m-1} = 0$ and $u_m = 1$).
- The largest real number less than 1 is $1 - b^{-m-1}$
  ($e = -1$, $u_0 = 0$ and $u_1 = \ldots = u_m = 1$).
- This results in important constants called "machine epsilons":

$$\epsilon_{\min} = b^{-m-1} \qquad \epsilon_{\max} = b^{-m}$$

- For numbers greater than b^{m+1}, the distance between numbers is > 1 and even integer parts are no longer exact.

DISTANCE / 3

Machine epsilons can be used to estimate the distance between the numbers in the entire domain.

In general: Around a number $x \neq 0$, the **relative distance** between machine numbers is approximately ϵ_{max}, and the **absolute distance** is thus $x \cdot \epsilon_{\mathsf{max}}$ (approximately, since neither x nor the product with ϵ_{max} need to be representable).

The estimate with ϵ_{max} is conservative and therefore usually preferred.

From now on, we define $\epsilon_m := \epsilon_{\mathsf{max}}$.

This machine epsilon is our minimal accuracy.

DISTANCE / 4

```
options(digits = 20)
1 + 1 / (2^53)
## [1] 1
```

```
1 + 1 / (2^52)
## [1] 1.0000000000000002
```

```
1 / (2^52)
## [1] 2.220446049250313e-16
```

```
.Machine$double.eps
## [1] 2.220446049250313e-16
```