INTRODUCTION TO FLOATING POINT NUMBERS

Martin Hořeňovský

INTRODUCTION TO FLOATING POINT NUMBERS

Martin Hořeňovský

DISCLAIMERS

Through the talk I will use *floats* as a shorthand for *floating-point numbers*.

Through the talk I will use *floats* as a shorthand for *floating-point numbers*.

This talk is about IEEE 754 *binary* floats.

We will not get into the full nitty-gritty details.

We will not get into the full nitty-gritty details.

We will look at floats through the lens of users.

```
assert(0.1 + 0.2 == 0.3);
assert(0.1 + 0.2 == 0.3);
```

FAIL

```cpp
assert(0.1 + 0.2 == 0.3);
```

FAIL

```cpp
assert(20'000'000.f + 1 == 20'000'000.f);
assert(0.1 + 0.2 == 0.3);
```

FAIL

```
assert(20'000'000.f + 1 == 20'000'000.f);
```

PASS

```
assert(0.1 + 0.2 == 0.3);
```

FAIL

```
assert(20'000'000.f + 1 == 20'000'000.f);
```

PASS

```
assert(20'000'000.f + 1.f + 1.f
                  ==
       1.f + 1.f + 20'000'000.f);
```

```
    assert(0.1 + 0.2 == 0.3);
```

FAIL

```
    assert(20'000'000.f + 1 == 20'000'000.f);
```

PASS

```
    assert(20'000'000.f + 1.f + 1.f
                  ==
           1.f + 1.f + 20'000'000.f);
```

FAIL

???

All the results make perfect sense …

All the results make perfect sense …

… if you understand how floats work.

```
assert(0.1 + 0.2 == 0.3);
```

FAIL

```
assert(20'000'000.f + 1 == 20'000'000.f);
```

PASS

```
assert(20'000'000.f + 1.f + 1.f
                  ==
       1.f + 1.f + 20'000'000.f);
```

FAIL

```cpp
TEST_CASE() {
    double a = 1;
    auto b = std::nextafter(a, 2);

    REQUIRE(a == b);
}
```

```cpp
TEST_CASE() {
    double a = 1;
    auto b = std::nextafter(a, 2);

    REQUIRE(a == b);
}
```

```
example.cpp:9: FAILED:
  REQUIRE( a == b )
with expansion:
  1.0 == 1.0
```

CONTENTS

CONTENTS

- Decimal floats

CONTENTS

- Decimal floats
- Binary floats

CONTENTS

- Decimal floats
- Binary floats
- IEEE-754 guarantees for math

CONTENTS

- Decimal floats
- Binary floats
- IEEE-754 guarantees for math
- Comparing floats

CONTENTS

- Decimal floats
- Binary floats
- IEEE-754 guarantees for math
- Comparing floats
- When does it all break down?

WHY ARE FLOATS WEIRD?

DESIGN CONSTRAINTS

DESIGN CONSTRAINTS

- Usable for computations in $\mathbb{R}$.

DESIGN CONSTRAINTS

- Usable for computations in $\mathbb{R}$.
- Must support both tiny and large numbers in the same operation, e.g. $0.000\,000\,000\,067$ and $2\,000\,000\,000\,000\,000\,000\,000\,000\,000\,000$.

DESIGN CONSTRAINTS

- Usable for computations in $\mathbb{R}$.
- Must support both tiny and large numbers in the same operation, e.g. $0.000\,000\,000\,067$ and $2\,000\,000\,000\,000\,000\,000\,000\,000\,000\,000$.
- Efficiently implementable in HW.

DECIMAL FLOATS

(NORMALIZED) SCIENTIFIC NOTATION

$$m * 10^n$$

$$m * 10^n$$

$$m \in \mathbb{R}, 1 \leq |m| < 10$$
$$n \in \mathbb{Z}$$

$$\begin{array}{ll} 3.5 & *10^{2} \\ 1.234 & *10^{7} \\ 1.22 & *10^{5} \end{array}$$

FIXED LENGTH SCIENTIFIC NOTATION

What if we can only use fixed number of digits?

What if we can only use fixed number of digits?

$$X.YYY * 10^Z$$

What if we can only use fixed number of digits?

$$X.YYY * 10^Z$$

Negative numbers will be handled by separately storing the minus sign.

What if we can only use fixed number of digits?

$$X.YYY * 10^{Z}$$

Negative numbers will be handled by separately storing the minus sign.

The same goes for negative exponents.

We just defined a decimal floating point format.

$$X.YYY * 10^Z$$

$$X.YYY * 10^Z$$

$X.YYY$ is called the *mantissa*.

$$X.YYY * 10^Z$$

$X.YYY$ is called the *mantissa*.

Z is called the *exponent*.

The format can represent interval

$$[-9\,999\,000\,000, 9\,999\,000\,000]$$

The smallest representable positive number is

$$1.000 * 10^{-9} \ (0.000\,000\,001)$$

or is it?

or is it?

$$0.001 * 10^{-9} \ (0.000\,000\,000\,001)$$

Note that the absolute difference between two closest numbers increases with their magnitude.

Note that the absolute difference between two closest numbers increases with their magnitude.

Relative difference remains the same.

$$2.435 * 10^4 - 2.434 * 10^4 = 1.000 * 10^1$$

vs

$$2.223 * 10^8 - 2.222 * 10^8 = 1.000 * 10^5$$

Other issues in the design:

Other issues in the design:

- We have both positive and negative zero

Other issues in the design:

- We have both positive and negative zero
- Zero doesn't have unique representation

There are 19 different ways to represent 0:

There are 19 different ways to represent 0:

$$0.000 * 10^{-9}$$
$$0.000 * 10^{-8}$$
$$\dots$$
$$0.000 * 10^{8}$$
$$0.000 * 10^{9}$$

And the same goes for -0.

And the same goes for -0.

We could use them for e.g. infinity or various errors.

Decimal floats exhibit the same weirdness as binary:

Decimal floats exhibit the same weirdness as binary:

$$\frac{1}{9} * 9 \neq 1$$

Decimal floats exhibit the same weirdness as binary:

$$\frac{1}{9} * 9 \neq 1$$

$$10\,000 + 4 = 10\,000$$

Decimal floats exhibit the same weirdness as binary:

$$\frac{1}{9} * 9 \neq 1$$

$$10\,000 + 4 = 10\,000$$

$$10\,000 + 4 + 4 \neq 4 + 4 + 10\,000$$

Why $10\,000 + 4 = 10\,000$?

Why $10\,000 + 4 = 10\,000$?

$10\,000 + 4 = 10\,004$, or $1.0004 * 10^4$

Why $10\,000 + 4 = 10\,000$?

$10\,000 + 4 = 10\,004$, or $1.0004 * 10^4$

But we can only use 3 digits after decimal point …

Why $10\,000 + 4 = 10\,000$?

$10\,000 + 4 = 10\,004$, or $1.0004 * 10^4$

But we can only use 3 digits after decimal point …

… and $10\,004$ is closer to $10\,000$ than to $10\,010$.

Why $\frac{1}{9} * 9 \neq 1$?

Why $\frac{1}{9} * 9 \neq 1$?

$$\frac{1}{9} = 0.\overline{1}$$

Why $\frac{1}{9} * 9 \neq 1$?

$$\frac{1}{9} = 0.\overline{1}$$

$$0.\overline{1} \rightarrow 1.111 * 10^{-1}$$

Why $\frac{1}{9} * 9 \neq 1$?

$$\frac{1}{9} = 0.\overline{1}$$

$$0.\overline{1} \rightarrow 1.111 * 10^{-1}$$

$$1.111 * 10^{-1} * 9 = 9.999 * 10^{-1}$$

Why $\frac{1}{9} * 9 \neq 1$?

$$\frac{1}{9} = 0.\overline{1}$$

$$0.\overline{1} \rightarrow 1.111 * 10^{-1}$$

$$1.111 * 10^{-1} * 9 = 9.999 * 10^{-1}$$

$$9.999 * 10^{-1} \neq 1.000 * 10^{0}$$

Why $10\,000 + 4 + 4 \neq 4 + 4 + 10\,000$?

Why $10\,000 + 4 + 4 \neq 4 + 4 + 10\,000$?

Summation is done left-to-right.

Why $10\,000 + 4 + 4 \neq 4 + 4 + 10\,000$?

Summation is done left-to-right.

$$10\,000 + 4 + 4 \neq 4 + 4 + 10\,000$$

Why $10\,000 + 4 + 4 \neq 4 + 4 + 10\,000$?

Summation is done left-to-right.

$$10\,000 + 4 + 4 \neq 4 + 4 + 10\,000$$

$$10\,000 \qquad + 4 \neq \qquad 8 + 10\,000$$

Why $10\,000 + 4 + 4 \neq 4 + 4 + 10\,000$?

Summation is done left-to-right.

$$10\,000 + 4 + 4 \neq 4 + 4 + 10\,000$$

$$10\,000 \qquad + 4 \neq \qquad 8 + 10\,000$$

$$10\,000 \neq 10\,010$$

RECAP

RECAP

Float addition and multiplication is not associative.

RECAP

Float addition and multiplication is not associative.

Floats will behave ~~weirdly~~ unintuitively in any base.

RECAP

Float addition and multiplication is not associative.

Floats will behave ~~weirdly~~ unintuitively in any base.

It is the result of using fixed size representation for $\mathbb{R}$.

BINARY FLOATS

Uses base-2 in representation.

Field	**32 bit float**	**64 bit float**
sign	1	1
mantissa	23	52
exponent	8	11

size	min positive value	max positive value
32 bits	$1 * 10^{-45}$	$3.4 * 10^{38}$
64 bits	$5 * 10^{-324}$	$1.8 * 10^{308}$

IEEE 754 guarantees the results of basic operations

IEEE 754 guarantees the results of basic operations

- addition

IEEE 754 guarantees the results of basic operations

- addition
- subtraction

IEEE 754 guarantees the results of basic operations

- addition
- subtraction
- multiplication

IEEE 754 guarantees the results of basic operations

- addition
- subtraction
- multiplication
- division

If you take the same numbers and add them on different platforms, you will get the same result.

This is different from getting useful result!

$$20\,000\,000 + 1 + 1 \neq 1 + 1 + 20\,000\,000$$

$$20\,000\,000 + 1 + 1 \neq 1 + 1 + 20\,000\,000$$

How can we fix this?

The naive option: sum smaller numbers first.

The naive option: sum smaller numbers first.

The advanced option: Kahan summation algorithm.

Kahan summation isn't a cure-all.

Kahan summation isn't a cure-all.

$$1 + 10^{100} + 1 - 10^{100} = 0$$

Kahan summation isn't a cure-all.

$$1 + 10^{100} + 1 - 10^{100} = 0$$

Other options are available, but they are computationally even more expensive.

What about multiplication?

Try to normalize your numbers to be around 1 before multiplication. Then denormalize them back.

Try to normalize your numbers to be around 1 before multiplication. Then denormalize them back.

You can also look for alternative formulations.

IEEE 754 does not provide guarantees on transcendental functions, such sine or logarithm.

Some implementations of performant CR (Correctly Rounded) math libraries exist, you can use them.

RECAP

RECAP

- Basic math operations should be reproducible.

RECAP

- Basic math operations should be reproducible.
- Numerical error can be reduced using the right approach.

RECAP

- Basic math operations should be reproducible.
- Numerical error can be reduced using the right approach.
- The simplest approach is to use wider float.

RECAP

- Basic math operations should be reproducible.
- Numerical error can be reduced using the right approach.
- The simplest approach is to use wider float.
- Transcendentals are hard.

COMPARING FLOATS

There are 5 ways to compare 2 floats:

There are 5 ways to compare 2 floats:

- bitwise comparison

There are 5 ways to compare 2 floats:

- bitwise comparison
- direct ("exact") comparison

There are 5 ways to compare 2 floats:

- bitwise comparison
- direct ("exact") comparison
- absolute margin comparison

There are 5 ways to compare 2 floats:

- bitwise comparison
- direct ("exact") comparison
- absolute margin comparison
- relative epsilon comparison

There are 5 ways to compare 2 floats:

- bitwise comparison
- direct ("exact") comparison
- absolute margin comparison
- relative epsilon comparison
- ULP based comparison

BITWISE COMPARISON

BITWISE COMPARISON

Two floats are equal if their bit representation is same.

BITWISE COMPARISON

Two floats are equal if their bit representation is same.

This is different from writing `a == b` in your code

Under bitwise comparison:

Under bitwise comparison:

- $-0 \neq 0$

Under bitwise comparison:

- $-0 \neq 0$
- $\mathrm{NaN} = \mathrm{NaN}$

Under bitwise comparison:

- $-0 \neq 0$
- $\mathrm{NaN} = \mathrm{NaN}$
- $\mathrm{NaN} \neq \mathrm{NaN}$

DIRECT COMPARISON

DIRECT COMPARISON

```
a == b
```

Under direct comparison:

Under direct comparison:

- $-0 = 0$

Under direct comparison:

- $-0 = 0$
- $\mathrm{NaN} \neq \mathrm{NaN}$

Direct comparison is useful e.g. if two different implementations should give the same result.

Direct comparison is useful e.g. if two different implementations should give the same result.

More often it is used as a mistake.

ABSOLUTE MARGIN COMPARISON

ABSOLUTE MARGIN COMPARISON

$$|\mathrm{lhs} - \mathrm{rhs}| \leq \mathrm{margin}$$

Two numbers are equal if their difference is less than some fixed margin.

PROS/CONS

PROS/CONS

- easy to reason about decimally

PROS/CONS

- easy to reason about decimally
- does not break down around 0

PROS/CONS

- easy to reason about decimally
- does not break down around 0
- same as direct comparison for large numbers

Fun fact, these two are not equivalent for floats:

Fun fact, these two are not equivalent for floats:

$$|\mathrm{lhs} - \mathrm{rhs}| \le \mathrm{margin}$$

Fun fact, these two are not equivalent for floats:

$$|\mathrm{lhs} - \mathrm{rhs}| \leq \mathrm{margin}$$

$$\mathrm{lhs} + \mathrm{margin} \geq \mathrm{rhs} \ \wedge \ \mathrm{rhs} + \mathrm{margin} \geq \mathrm{lhs}$$

RELATIVE MARGIN COMPARISON

RELATIVE MARGIN COMPARISON

$$|\mathrm{lhs} - \mathrm{rhs}| \leq \epsilon * \max(|\mathrm{lhs}|, |\mathrm{rhs}|)$$

RELATIVE MARGIN COMPARISON

$$|\mathrm{lhs} - \mathrm{rhs}| \leq \epsilon * \max(|\mathrm{lhs}|, |\mathrm{rhs}|)$$

$$|\mathrm{lhs} - \mathrm{rhs}| \leq \epsilon * \min(|\mathrm{lhs}|, |\mathrm{rhs}|)$$

Two numbers are equal if their difference is within some factor of each other.

PROS/CONS

PROS/CONS

- easy to reason about decimally

PROS/CONS

- easy to reason about decimally
- does not break down for large numbers

PROS/CONS

- easy to reason about decimally
- does not break down for large numbers
- breaks down around 0

"UNITS IN LAST PLACE" (ULP) BASED COMPARISON

ULP distance of two numbers is how many "steps" need to be taken from one to another.

ULP distance of two numbers is how many "steps" need to be taken from one to another.

A step is going between two representable numbers.

$$\text{ulpDistance}(2.400 * 10^4, 2.400 * 10^4) = 0$$

$$\mathrm{ulpDistance}(2.400 * 10^4, 2.400 * 10^4) = 0$$

$$\mathrm{ulpDistance}(2.400 * 10^4, 2.401 * 10^4) = 1$$

$$\text{ulpDistance}(2.400 * 10^4, 2.400 * 10^4) = 0$$

$$\text{ulpDistance}(2.400 * 10^4, 2.401 * 10^4) = 1$$

$$\text{ulpDistance}(2.400 * 10^4, 2.500 * 10^4) = 100$$

$$\text{ulpDistance}(-0.000 * 10^0, 0.000 * 10^0) = \textbf{?}$$

Two numbers are equal if their ULP distance is less than some fixed number.

PROS/CONS

PROS/CONS

- handles both 0 and large numbers gracefully

PROS/CONS

- handles both 0 and large numbers gracefully
- easy to reason about *numerically*

PROS/CONS

- handles both 0 and large numbers gracefully
- easy to reason about *numerically*
- very hard to reason about *decimally*

RECAP

RECAP

- There is no universal approach to comparing floats.

RECAP

- There is no universal approach to comparing floats.
- Direct comparison is most often a mistake, but not always.

RECAP

- There is no universal approach to comparing floats.
- Direct comparison is most often a mistake, but not always.
- You need to understand how your tools compute ULP distance.

WHAT ABOUT CATCH2?

APPROX

APPROX

```
REQUIRE( Pi == Approx(3.14) );
```

APPROX

```
REQUIRE( Pi == Approx(3.14) );
```

- Supports absolute margin comparison

APPROX

```
REQUIRE( Pi == Approx(3.14) );
```

- Supports absolute margin comparison
- Supports relative margin comparison*

APPROX

```
REQUIRE( Pi == Approx(3.14) );
```

- Supports absolute margin comparison
- Supports relative margin comparison*
- LEGACY

MATCHERS

MATCHERS

```
REQUIRE_THAT( Pi, WithinRel(3.14) );
```

MATCHERS

```
REQUIRE_THAT( Pi, WithinRel(3.14) );
```

- `WithinAbs` — absolute margin comparison

MATCHERS

```
REQUIRE_THAT( Pi, WithinRel(3.14) );
```

- `WithinAbs` — absolute margin comparison
- `WithinRel` — relative margin comparison

MATCHERS

```
REQUIRE_THAT( Pi, WithinRel(3.14) );
```

- `WithinAbs` — absolute margin comparison
- `WithinRel` — relative margin comparison
- `WithinULP` — ULP based comparison

CATCH2 ULPS

CATCH2 ULPS

$$\begin{aligned}
\text{ulpDistance}(-0, 0) &= 0 \\
\text{ulpDistance}(\text{DBL\_MAX}, \infty) &= 1 \\
\text{ulpDistance}(\text{NaN}, X) &= \infty
\end{aligned}$$

LIMITATIONS

Non IEEE-754 platforms are an obvious issue, Non IEEE-754 platforms are an obvious issue,

e.g. IBM defaults to a different model for floats

But some IEEE-754 platforms can cause issues too, But some IEEE-754 platforms can cause issues too, e.g. Intel's old FPU defaults to 80-bit computations.

Some languages treat floats differently at compile time and runtime.

Some languages treat floats differently at compile time and runtime.

e.g. Go or C++

Go does its own thing for constants

Go does its own thing for constants

```go
package main

import "fmt"

func main() {
	a := -0.0
	fmt.Println(a, 1/a)
}
```

Go does its own thing for constants

```go
package main

import "fmt"

func main() {
    a := -0.0
    fmt.Println(a, 1/a)
}
```

```
0 +Inf
```

C++ does not handle some edge cases well

C++ does not handle some edge cases well

```
#include <fmt/core.h>

int main() {
    double neg_zero = -0.0;
    double neg_infinity = 1 / neg_zero;

    fmt::print("{} {}\n", neg_zero, neg_infinity);
}
```

C++ does not handle some edge cases well

```cpp
#include <fmt/core.h>

int main() {
    double neg_zero = -0.0;
    double neg_infinity = 1 / neg_zero;

    fmt::print("{} {}\n", neg_zero, neg_infinity);
}
```

```
-0 -inf
```

C++ does not handle some edge cases well

C++ does not handle some edge cases well

```
#include <fmt/core.h>

int main() {
    constexpr double neg_zero = -0.0;
    constexpr double neg_infinity = 1 / neg_zero;

    fmt::print("{} {}\n", neg_zero, neg_infinity);
}
```

C++ does not handle some edge cases well

```cpp
#include <fmt/core.h>

int main() {
    constexpr double neg_zero = -0.0;
    constexpr double neg_infinity = 1 / neg_zero;

    fmt::print("{} {}\n", neg_zero, neg_infinity);
}
```

```
error: '(1.0e+0 / -0.0)' is not a constant expression
   15 |     constexpr double neg_infinity = 1 / neg_zero;
      |
```

There are compiler flags that break IEEE standard

There are compiler flags that break IEEE standard

- `-ffast-math` (`/fp:fast`)

There are compiler flags that break IEEE standard

- `-ffast-math` (`/fp:fast`)
- `-Ofast`

There are compiler flags that break IEEE standard

- `-ffast-math` (`/fp:fast`)
- `-Ofast`
- `-funsafe-math-optimizations`
- `-ffinite-math-only`
- …

External code can also break reproducibility

External code can also break reproducibility

```
int evil() {
    fesetround(FE_DOWNWARD);

    // never resets original rounding mode
}
```

RECAP

RECAP

- You have to know what your language guarantees

RECAP

- You have to know what your language guarantees
- Compiler flags can break IEEE 754 guarantees

RECAP

- You have to know what your language guarantees
- Compiler flags can break IEEE 754 guarantees
- External code can break reproducibility as well

THE END

- You can't constant fold $x + 0.0$ into x

- You can't constant fold $x + 0.0$ into x
- You can't constant fold $x * 0.0$ into 0.0

- You can't constant fold $x + 0.0$ into x
- You can't constant fold $x * 0.0$ into 0.0

QUESTIONS?