Messing with Floating Point

Ryan Baker

“Never mess with floating point”

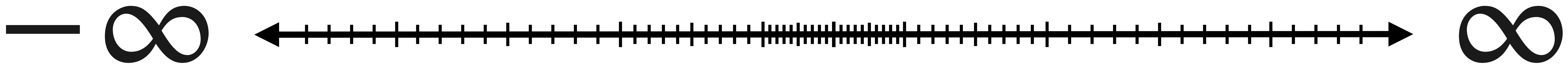
Agenda

1.f Floating Point Basics

Design Constraints

Key Considerations

- Suitable for computations in $\mathbb{R}$



- Practically implementable on hardware
- Consistent across platforms

Scientific Notation

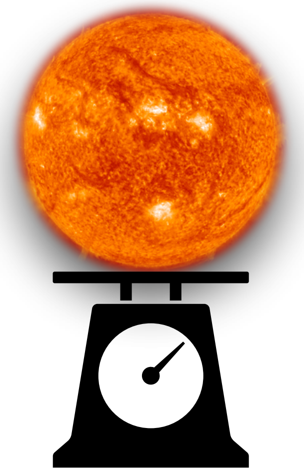
$$\pm m \times b^e \begin{cases} 1 \leq m < b \\ e \in \mathbb{Z} \end{cases}$$

$$m = \underbrace{x_0\ .\ x_1 x_2 x_3 \ldots x_{p-1}}_{p\ digits}$$

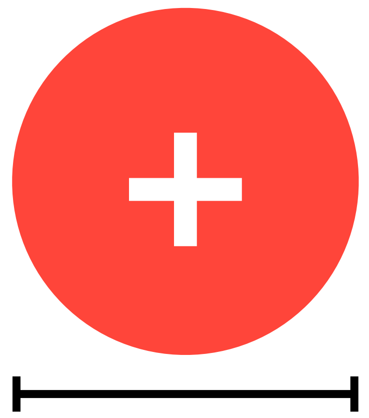
$$b = 10 \quad p = 3$$

$$3.14 \times 10^{0} \approx \pi$$

$$1.99 \times 10^{30} \approx$$



$$8.40 \times 10^{-16} \approx$$



$$3.14 \times 10^{0} \qquad 1.99 \times 10^{30} \qquad 8.40 \times 10^{-16}$$

- ✓ Suitable for computations in $\mathbb{R}$
- ✓ Practical to store and manipulate
- ✓ Consistent and reproducible

Float Representation

Binary Scientific Notation

0

- + is 0, - is 1
- Solely responsible for the sign
- `0.0` and `-0.0` are distinct

Exponent | Mantissa

`10000000`

$w\ bits$

$= 128 - 127 = 1$

- Controls the magnitude
- Stored in a *bias* format

$$e = E - k \qquad k = 127$$

- $e_{min} < e < e_{max}$

`0`

Mantissa

$$1 \leq m < 2$$

$$m = 1 + 2^{-1} + 2^{-4} + 2^{-7} + \ldots \approx 1.571$$

2^{-1} 2^{1-p}

1. `1001001 00001111 11011011`

p bits

`0` `10000000`

$$+1.571 \times 2^{1} \approx \pi$$

0	10000000	1001001 00001111 11011011

Float Format Parameters

IEEE 754 Specification

	16-bit	32-bit	64-bit	128-bit
p	11	24	53	113
w	5	8	11	15
k (bias)	15	127	1023	16383

1.f Floating Point Basics

- ✓ Design Constraints
- ✓ Scientific Notation
- ✓ Float Representation
- ✓ Float Format Parameters

Agenda

✓	Floating Point Basics
2.f	Where it Breaks Down
3.f	IEEE 754 Standard
4.f	Bonus Content

2.f Where it Breaks Down

```
float f = 0.1;
assert(f == 0.1);
```

2.1f Representation Error

Representation Error

$$\frac{1}{3} \rightarrow 3.33 \times 10^{-1}$$

$$333333\ldots$$

$$\frac{1}{10} \rightarrow 1.100110\ldots \times 2^{-4}$$

$$011001100\ldots$$

“Squeezing infinitely many real numbers into a finite number of bits requires an *approximate* representation.”

David Goldberg

```c
float f = 0.1;
assert(f == 0.1);
```

32-bit

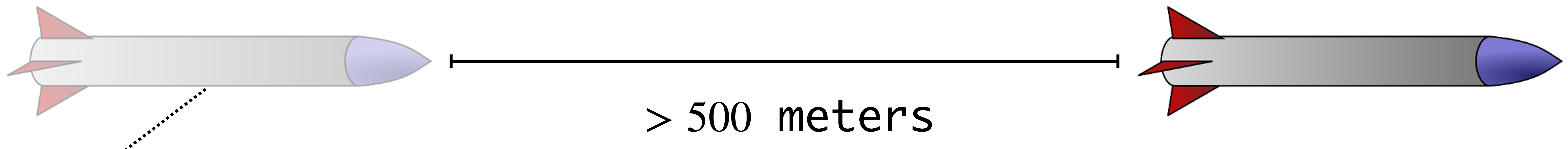
64-bit

```
float f = 0.1;
cout << setprecision(INT_MAX) << f << endl;
```

```
>> 0.100000001490116119384765625
```

```
0.10000000000000000000000000000
0.10000000149011611938476562
```

$\approx 1.5 \times 10^{-9}$



```
uint tenths;
float sec = tenths * 0.1;
```

$$96 \times 3600 \times 10 \times 9.5 \times 10^{-8} = 0.33$$

Quantifying Error

ULPs

- Units in the Last Place
- Variance in terms of least-significant digit
- Useful for representation error

Relative Error

- Same as percent error

$$E = \frac{|f - r|}{r}$$

- Useful for post-calculation analysis

Quantifying Error

$$\pi \to 3.14$$

$$\frac{\pi - 3.14}{\pi} = 0.0005$$

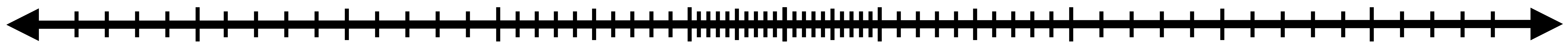
$$3.14\textcolor{red}{159265...} \to 0.159 \text{ ULPs}$$

Mitigating Representation Error

- Use higher precision types

- Prefer exact values

$\pm m \times 2^e$



All integers up to 2^p are exact

- Use stable algorithms

$$x \times 5 \iff \frac{x}{0.2}$$

$$x \times \frac{2}{3} \iff \frac{x}{1.5}$$

$$x \times 0.5 \iff \frac{x}{2}$$

```cpp
float f = numeric_limits<float>::min();
// f = 1.17549e-38
cout << (f / 2.f) << endl;
```

```
>> 5.87747e-39
```

2.2f Denormal Numbers

1.17549e-38

0	00000001	0000000 00000000 00000000

5.87747e-39

0	00000000	1000000 00000000 00000000

00000000 → Implicit bit = 0

Denormal Numbers

$\neg(1 \leq m < 2)$

- $e = e_{min} =$ `00000000`
- Help prevent underflow

- Lower precision
- Poor performance

2.3f Float Comparison

```
float f = 0.1;
assert(f == 0.1);
```

Comparison

- **Bitwise:** $a == b$ iff a and b have the same bit representation
- **IEEE 754:** Bitwise (NaN $\neq x$ $\quad -\infty < x < \infty \quad 0 = -0$)
- **Epsilon:** $a == b$ iff $|a - b| < \epsilon$
- **Relative:** $a == b$ iff $\dfrac{|a - b|}{a} < \epsilon$
- **ULPs:** $a == b$ iff $\texttt{ulp\_distance}(a,\ b) < \epsilon$

```
std::nextafter(float from, float to)
```

```
std::numeric_limits<float>::epsilon()
```

```
int ulp_distance(float a, float b)
{
    uint32_t inta = *reinterpret_cast<uint32_t*>(&a);
    uint32_t intb = *reinterpret_cast<uint32_t*>(&b);
    return intb - inta;
}
```

```cpp
float sum = 0;

for (int i = 0; i < 1'000'000'000; ++i)
    sum += 1.f / 1'000'000'000.f;

cout << sum << endl;
```

```
>> 0.03125
```

2.4f Resolution Error

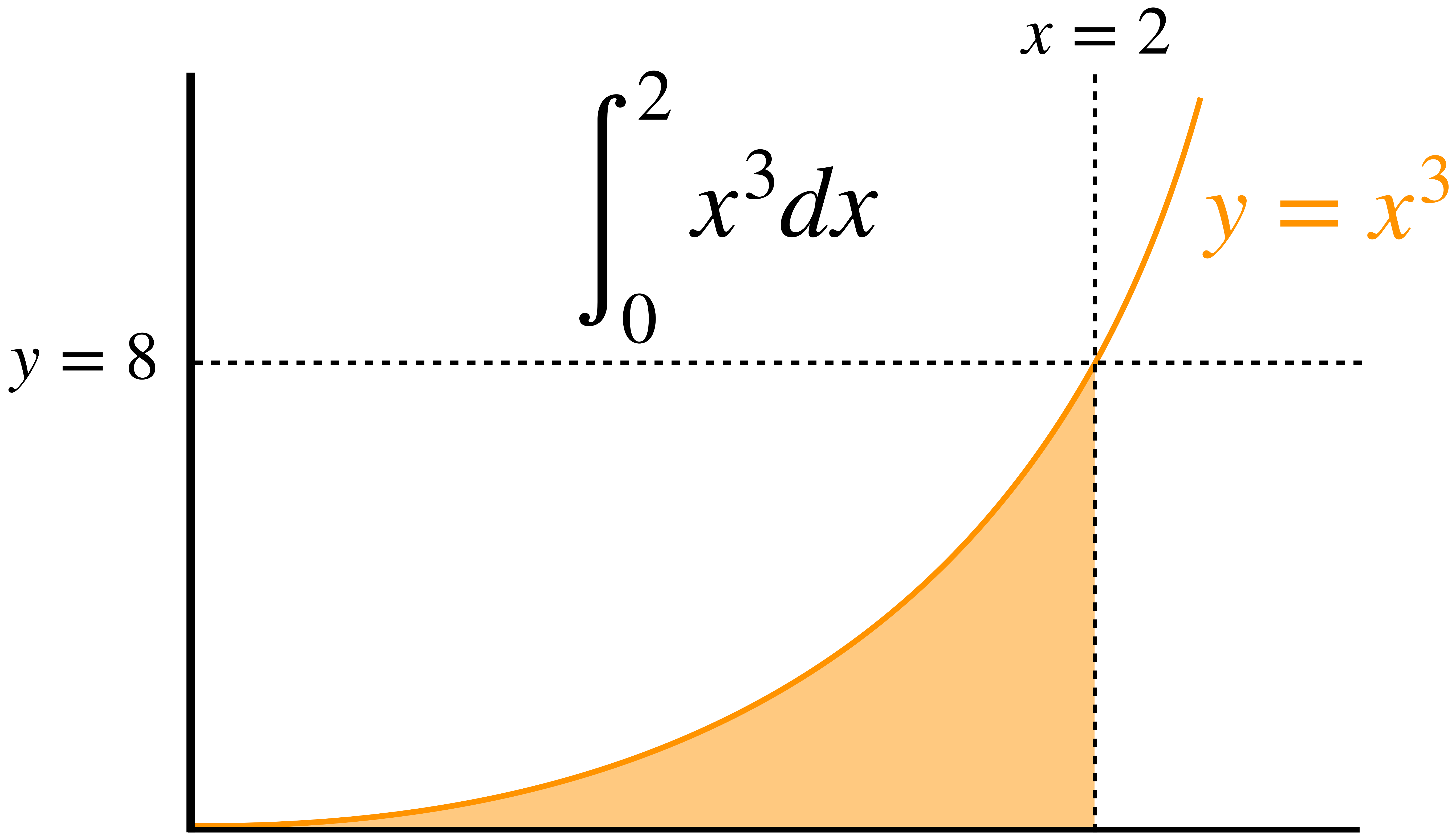
Resolution Error

- Occurs when adding two very mismatched numbers

$$100 + 0.01 \rightarrow \begin{array}{r} 1.00 \times 10^{2} \\ + \ 1.00 \times 10^{-2} \\ \hline \end{array} \rightarrow \begin{array}{r} 1.00 \times 10^{2} \\ + \ 0.00 \times 10^{2} \\ \hline 1.00 \times 10^{2} \end{array} \rightarrow 100$$

- Group computations by magnitude
- Use more sophisticated algorithms

$$= 4$$



```cpp
// integrate x**3 from 0 -> 2
float sum = 0;
const float dx = 0x1.0p-22;

for (float x = 0; x <= 2; x += dx)
    sum += x * x * x * dx;

cout << sum << endl;
```

>> 4.00028

```cpp
// integrate x**3 from 0 -> 2
float sum = 0;
const float dx = 0x1.0p-22;

for (float x = 2; x >= 0; x -= dx)
    sum += x * x * x * dx;

cout << sum << endl;
```

>> 3.94732

```
float kahan_sum(const vector<float>& addends)
{
    float sum = 0;  // accumulator
    float comp = 0; // running compensation for lost bits
    for (float f: addends)
    {
        float y = f - comp;    // comp = 0 on first iter
        float t = sum + y;     // sum > y… y gets truncated
        comp = (t - sum) - y; // comp recovers lost part of y
        sum = t;               // error gets recovered next iter
    }
    return sum;
}
```

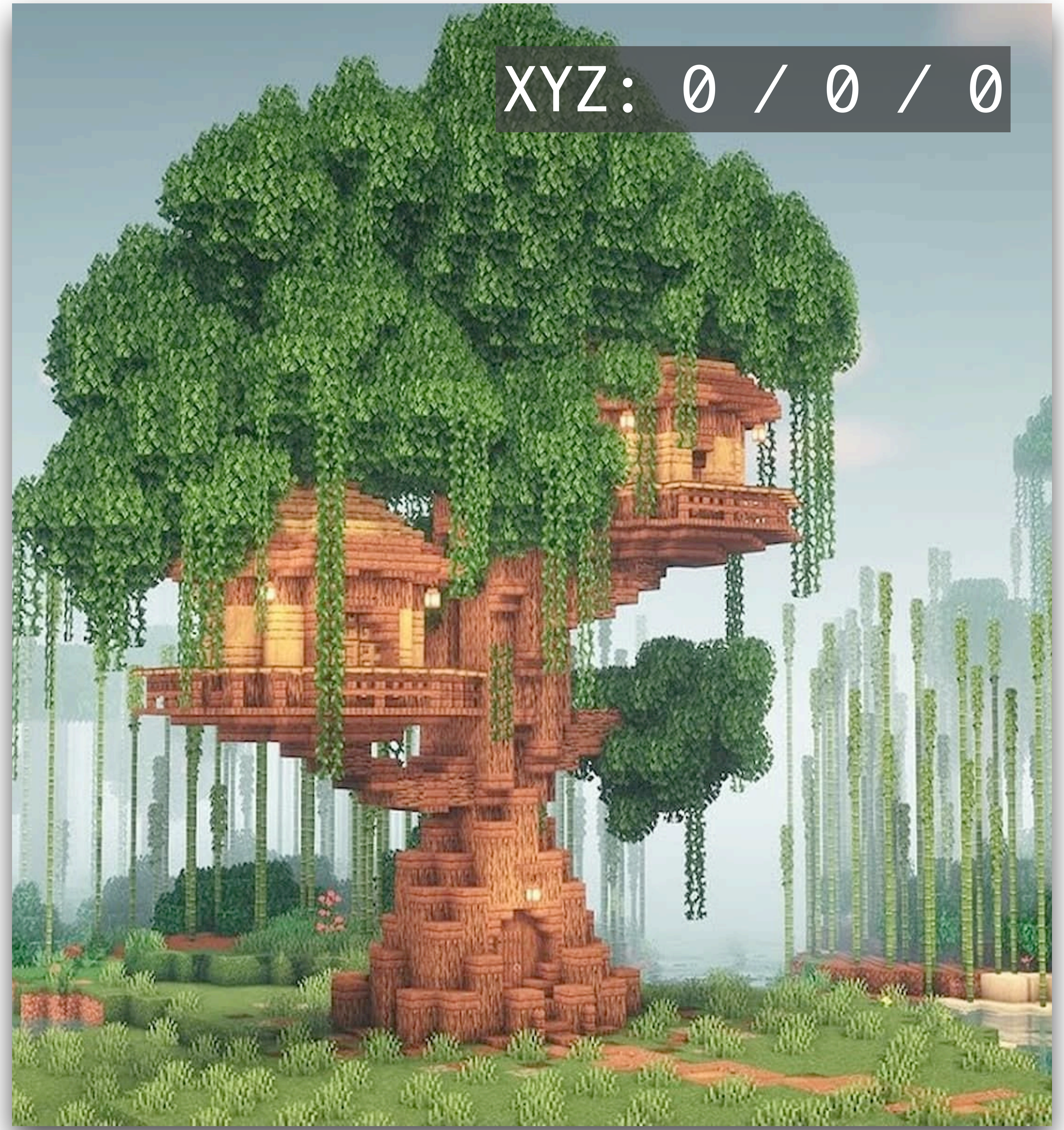
```
vector<float> nums(100'000'000);

random_device rd;
mt19937 gen(rd());
uniform_real_distribution<float> dist(0.f, 1.f);

generate(nums.begin(), nums.end(), [&]() { return dist(gen); });

cout << int(kahan_sum(nums)) << endl;
cout << int(naive_sum(nums)) << endl;
```

```
>> 49997528
>> 16777216
```



- `131072:` diagonal walking is bumpy
- `262144:` strings become invisible
- `2097152:` blocks render in 2D
- `4194304:` walking is impossible
- `8388608:` fall through the floor
- `16777216:` every 2nd block renders

2.f Where it Breaks Down

- ✓ Representation Error
- ✓ Denormalization
- ✓ Float Comparison
- ✓ Resolution Error

Agenda

✓	Floating Point Basics
✓	Where it Breaks Down
3.f	IEEE 754 Standard
4.f	Bonus Content

3.f IEEE 754 Standard

IEEE 754 Standard

Introduction

- Developed in **1985**, updated in **2019**
- IEEE 754:2019 == ISO/IEC 60559:2020
- Representation, exceptional values, exceptions, and operations

- **Independent from C++**

C++ Standard

- Does not mandate IEEE 754 compliance
- C++23 has very little to say about floating point
- C++26 / C++29 might have more

C++26 / C++29

P3375R0

- **Compiler flags:** `-fno-fast-math`
- **Specification overhaul:** define behavior for `float`
- **New type:** `std::IEEE_float32_t`
- **Qualification:**

```
float func(float a, float b) reproducible
```

Floating Point Environment

- To check IEEE 754 compliance:

```
std::numeric_limits<float>::is_iec559
```

- To access floating point environment:

```
#include <cfenv>

#pragma STDC FENV_ACCESS ON
```

3.1f Exceptional Values

Exceptional Values

Zero

	Sign	Exponent	Mantissa
$0 =$	0	00000000	0000000 00000000 00000000
$-0 =$	1	00000000	0000000 00000000 00000000

"Comparisons shall ignore the sign of zero (so +0 = -0)"

$$x \times 0 = 0 \qquad x \times -0 = -0 \qquad 0 \times -0 = -0$$

Exceptional Values

Infinity

$\infty =$ `0` `11111111` `0000000 00000000 00000000`

$-\infty =$ `1` `11111111` `0000000 00000000 00000000`

$$-\infty < x < \infty$$

$$\frac{x}{\infty} = 0$$

$$\frac{x}{0} = \infty$$

$$\infty + x = \infty$$

Exceptional Values

NaN

	Sign	Exponent	Mantissa
qNaN =	0	11111111	1000000 00000000 00000000
sNaN =	0	11111111	0??????? ???????? ????????

$$\frac{0}{0} = \text{NaN}$$

$$\text{NaN} + x = \text{NaN}$$

$$\infty - \infty = \text{NaN}$$

$$\text{NaN} \neq x \neq \text{NaN}$$

```
bool std::isnan(float f)

bool std::isinf(float f)

float std::numeric_limits<float>::quiet_NaN()

float std::numeric_limits<float>::signaling_NaN()

float std::numeric_limits<float>::infinity()
```

3.2f Exceptions

Exceptions

Invalid Operation

`FE_INVALID`

"The *invalid operation* exception is signaled if and only if there is no usefully definable result."

$$\frac{0}{0} \qquad \frac{\infty}{\infty} \qquad 0 \times \infty \qquad \mathrm{sqrt}(-x)$$

Exceptions

Division by Zero

`FE_DIVBYZERO`

"The *divideByZero* exception shall be signaled if and only if an exact infinite result is defined for an operation on finite operands."

$$\frac{x}{0} \qquad \log(0)$$

Exceptions

Overflow

`FE_OVERFLOW`

> "The *overflow* exception shall be signaled if the destination format's largest finite number is exceeded in magnitude…"

$$\texttt{FLT\_MAX} \times 2 = \begin{cases} \texttt{FLT\_MAX} \\ \infty \end{cases}$$

Exceptions

Underflow

`FE_UNDERFLOW`

“The *underflow* exception shall be signaled when a tiny non-zero result is detected”

$$|x| < 2^{e_{min}}$$

Exceptions

Inexact

`FE_INEXACT`

"Except as specified otherwise, an operation delivering a numerical result that signals no other exception shall signal inexact if its rounded result differs from what would have been computed were both the exponent range and precision unbounded."

$$0.1 \qquad \frac{1}{3} \qquad \mathrm{sqrt}(2)$$

```
int std::fetestexcept(int e)

int std::feclearexcept(int e)

int std::feraiseexcept(int e)
```

```
void show_fe_exceptions()
{
    cout << "exceptions raised:";

    if (fetestexcept(FE_INVALID))   cout << " invalid";
    if (fetestexcept(FE_DIVBYZERO)) cout << " divbyzero";
    if (fetestexcept(FE_OVERFLOW))  cout << " overflow";
    if (fetestexcept(FE_UNDERFLOW)) cout << " underflow";
    if (fetestexcept(FE_INEXACT))   cout << " inexact";

    feclearexcept(FE_ALL_EXCEPT);
    cout << endl;
}
```

```
show_fe_exceptions();

float f = 0.1; // 0.1 is inexact
show_fe_exceptions();

f /= 0; divide 0 = infinity
show_fe_exceptions();

f *= 0; // infinity * 0
show_fe_exceptions();
```

```
>> exceptions raised:
>> exceptions raised: inexact
>> exceptions raised: divbyzero
>> exceptions raised: invalid
```

3.3f Rounding Modes

Rounding Modes

- Round to 0: $x \to 0$ `FE_TOWARDZERO`
- Round up: $x \to \infty$ `FE_UPWARD`
- Round down: $x \to -\infty$ `FE_DOWNWARD`
- Round to nearest: `FE_TONEAREST`

$$3.142 \to 3.14 \qquad 2.718 \to 2.72$$

$$1.005 \to 1.00 \qquad 1.015 \to 1.02$$

```cpp
int std::fegetround();

int std::fesetround(int round);
```

3.4f Arithmetic Guarantees

Basic Operations

`+ - * / sqrt fma`

- Deterministic and reproducible
- **Exactly rounded (correct)**
- $a + b = b + a \quad a \times b = b \times a$
- $(a + b) + c \neq a + (b + c) \quad (a \times b) \times c \neq a \times (b \times c)$
- Handling of special values

```
float std::fma(float x, float y, float z)
```

Conversion Operations

`float->int   int->float   float->float   float->string`

- Exactly rounded
- For conversion to integers:
  - `int(float x)` always rounds to zero
  - `std::rint(float x)` obeys the current rounding mode
- Parse decimal strings
- Produce a unique string representation *(9 digits, 17 digits)*

3.f IEEE 754 Standard

- ✓ Exceptional Values
- ✓ Exceptions
- ✓ Rounding Modes
- ✓ Arithmetic Guarantees

Agenda

✓	Floating Point Basics
✓	Where it Breaks Down
✓	IEEE 754 Standard
4.f	Bonus Content

4.f Bonus Content

4.1f Ryu Float-to-String

```cpp
float f = 0.1;
cout << setprecision(INT_MAX) << f << endl;
```

```
>> 0.100000001490116119384765625
```

0 01111011 1001100 11001100 11001101

$$e := \boxed{01111011} = 123 - 127 = -4$$

$$m := \boxed{1001100\ 11001100\ 11001101} = 1.6$$

```
e = -4
```

```
m = 1.6
```

$$1.6 \times 2^{-4} = 0.1$$

```
e = -4

m = 1.6 << 23
```

$$1.6 \times 2^{-4} = 0.1$$

```
e = -4

m = 1.6 << 23 = 13,421,773
```

$$1.6 \times 2^{-4} = 0.1$$

```
e = -4 - 23

m = 1.6 << 23 = 13,421,773
```

$$1.6 \times 2^{-4} = 0.1$$

```
e = -4 - 23 = -27

m = 1.6 << 23 = 13,421,773
```

$$1.6 \times 2^{-4} = 0.1$$

```
e = -4 - 23 = -27

m = 1.6 << 23 = 13,421,773
```

$$13{,}421{,}773 \times 2^{-27} = 0.1$$

$e = -27$

$m = 13{,}421{,}773$

```
e = -27

m = 13,421,773 * 5**27 * 5**-27
```

```
e = -27

m = 13,421,773 * 5**27 * 5**-27

13,421,773 * 2**-27
```

```
e = -27

m = 13,421,773 * 5**27 * 5**-27

13,421,773 * 5**27 * 5**-27 * 2**-27
```

```
e = -27

m = 13,421,773 * 5**27 * 5**-27

13,421,773 * 5**27 * 10**-27
```

```
e = -27

m = 13,421,773 * 5**27 * 5**-27

100000001490116119384765625 * 10**-27
```

0.100000001490116119384765625

```cpp
float f = 0.1;
cout << setprecision(INT_MAX) << f << endl;
```

```
>> 0.100000001490116119384765625
```

4.2f Fast Inverse Square Root

```c
float Q_rsqrt(float number)
{
    float x = number * 0.5f;
    float y = number;

    int i = * (int*) &y;          // evil bit hack
    i = 0x5f3759df - (i >> 1); // what the f?
    y = * (float*) &i;
    y *= (1.5f - (x * y * y)); // 1st iteration
    y *= (1.5f - (x * y * y)); // 2nd iteration

    return y;
}
```

4.3f Hexadecimal Float Literals

`0x<mantissa>p<exponent>`

Thank you!

rbaker@xtier.com