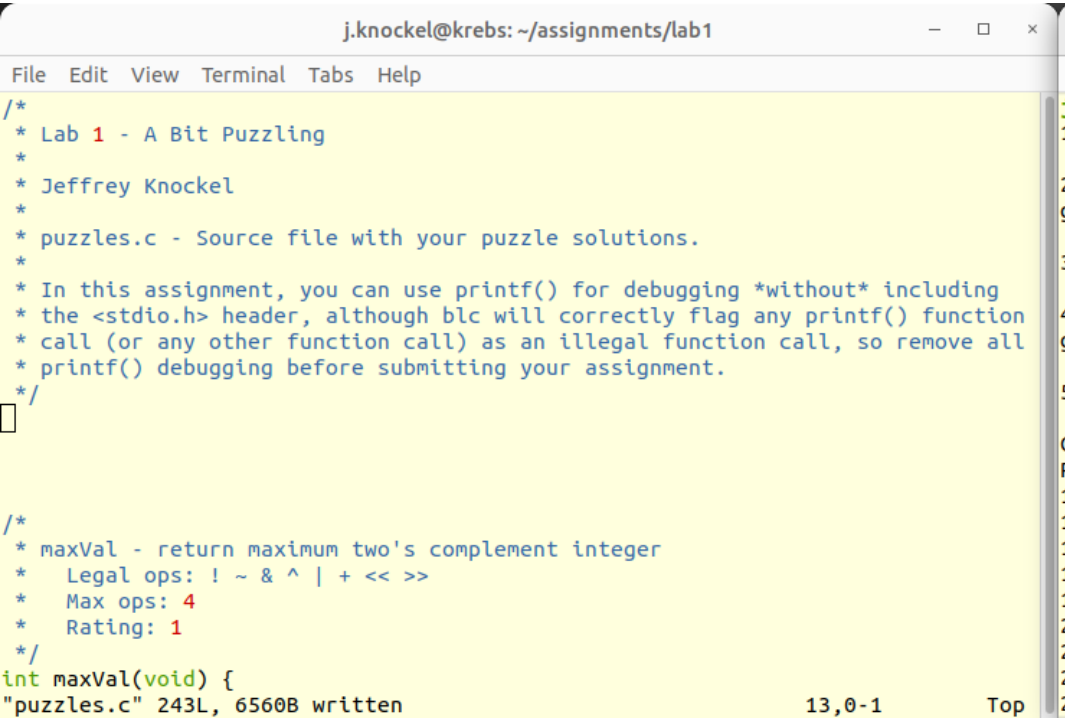
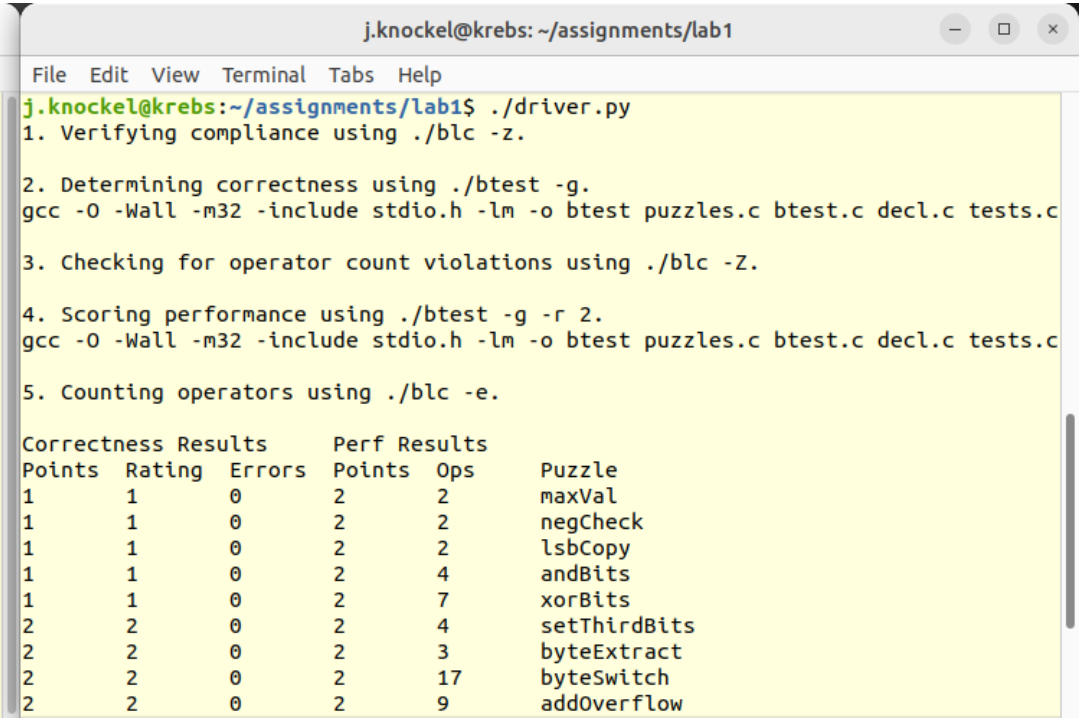

0.1 + 0.2 == 0.3
0.30000000000000004
ProgrammerHumor.io

TODO

- Assignment 1 is assigned (due September 24)

Lab 1 workflow





Online book

- Link in Syllabus:
- https://diveintosystems.org/book/

Overview for today

- Continuing discussion of floating point

Scientific notation

- Written in scientific notation:
- $4.37 * 10^{14}$
- or
- $8.59 * 10^{-11}$
- (The 10 is the same every time)

Floating point

- Written in binary:
- $1.10001 * 2^{48}$
- or
- $1.01111 * 2^{-34}$
- All we need to encode in binary (plus the sign)

IEEE 754 floating point

- *Floating* point standard used across many machines
- Released in 1985
- 70 page document! 🥲
- Let’s cover the basics…

Two basic types

- float: “single precision”: 32 bits
- double: “double precision”: 64 bits

Terms

- $(-1)^s * M * 2^E = (-1)^s * 1.\text{fraction} * 2^E$
- s is sign bit
- M is mantissa
- M = 1.fraction
- E is exponent

But there's a catch

- How do we encode s, fraction, E in binary?
- s is straightforward…
  - 0 is 0, 1 is 1
- fraction is also straightforward…
  - 1011 means mantissa is $1.1011 = 2^0 + 2^{-1} + 2^{-3} + 2^{-4}$
- But E…

Interpretation of exp

- $(-1)^s * M * 2^E = (-1)^s * 1.\text{fraction} * 2^{\text{exp} - 2^{k-1} + 1}$
- s is sign bit
- M is mantissa (1.fraction)
- E = exp (as an unsigned number) - bias
- bias = $2^{k-1} - 1$, where *k* is # of bits of exp

len: 1 **k** = 8 or 11 23 or 52

Encode -2 in 32-bit float?

- $(-1)^s * M * 2^E = (-1)^s * 1.\text{fraction} * 2^{\text{exp} - 2^{k-1} + 1}$

Encode -2 in 32-bit float?

- $(-1)^s * M * 2^E = (-1)^s * 1.\text{fraction} * 2^{\text{exp} - 2^{k-1} + 1}$
- Want s = 1
- E = 1
- M = 1.00000…
- $(-1)^1 * 1.00000\ldots * 2^1 = -2$

- We want s = 1, so set s = 1
- We want M = 1.00000…
  - Since M = 1.frac, set frac = 00000…
- We want E = 1… what is exp?
- Recall:
  - E = exp - bias
  - E = exp - 2^{k-1} + 1

len: 1 **k** = 8 or 11 23 or 52

- How do we encode E = 1 in a 32-bit float?
- $E = exp - 2^{k-1} + 1$
- $\Rightarrow 1 = exp - 2^{7} + 1$
- $\Rightarrow exp = 128$
- exp is an unsigned integer, so encode 128 as:
- 10000000

len: 1 **k** = 8 or 11 23 or 52

- Combining everything, -2.0 is encoded as:
- 1 10000000 00000000000000000000000

len: 1 **k** = 8 or 11 23 or 52

- Combining everything, -2.0 is encoded as:
- 1 10000000 00000000000000000000000
- What would we have to change to encode -4.0?
- $(-1)^s * M * 2^E = (-1)^s * 1.\text{fraction} * 2^{\text{exp} - 2^{k-1} + 1}$

- Combining everything, -2.0 is encoded as:
- 1 10000000 00000000000000000000000
- What would we have to change to encode -4.0?
- $(-1)^s * M * 2^E = (-1)^s * 1.\text{fraction} * 2^{\text{exp} - 2^{k-1} + 1}$
- Combining everything, -4.0 is encoded as:
- 1 10000001 00000000000000000000000

len: 1 **k** = 8 or 11 23 or 52

- What is this number?
- 1 10000010 00001100000000000000000
- $(-1)^s * M * 2^E = (-1)^s * 1.\text{fraction} * 2^{\text{exp} - 2^{k-1} + 1}$

len: 1 **k** = 8 or 11 23 or 52

- What is this number?
- 1 10000010 00001100000000000000000
- $(-1)^s * M * 2^E = (-1)^s * 1.\text{fraction} * 2^{\text{exp} - 2^{k-1} + 1}$
- $(-1)^1 * 0b1.000011 * 2^{0b10000010 - 2^{k-1} + 1}$

len: 1 **k** = 8 or 11 23 or 52

- What is this number?
- 1 10000010 00001100000000000000000
- $(-1)^{s} * M * 2^{E} = (-1)^{s} * 1.\text{fraction} * 2^{\text{exp} - 2^{k-1} + 1}$
- $(-1)^{1} * 0b1.000011 * 2^{0b10000010 - 2^{k-1} + 1}$
- $-1 * (1 + 1/32 + 1/64) * 2^{2 + 128 - 128 + 1}$

len: 1 **k** = 8 or 11 23 or 52

- What is this number?
- 1 10000010 00001100000000000000000
- $(-1)^s * M * 2^E = (-1)^s * 1.\text{fraction} * 2^{\text{exp} - 2^{k-1} + 1}$
- $(-1)^1 * 0b1.000011 * 2^{0b10000010 - 2^{k-1} + 1}$
- $-1 * (1 + 1/32 + 1/64) * 2^{2 + 128 - 128 + 1}$
- $-1 * (1 + 1/32 + 1/64) * 2^3$

len: 1 **k** = 8 or 11 23 or 52

- What is this number?
- 1 10000010 00001100000000000000000
- $(-1)^s * M * 2^E = (-1)^s * 1.\text{fraction} * 2^{\text{exp} - 2^{k-1} + 1}$
- $(-1)^1 * 0b1.000011 * 2^{0b10000010 - 2^{k-1} + 1}$
- $-1 * (1 + 1/32 + 1/64) * 2^{2 + 128 - 128 + 1}$
- $-1 * (1 + 1/32 + 1/64) * 2^3$
- -8.375

Degenerate cases

- What we have described is *normalized form*
- There are three forms:
  - Normalized form ✅
  - Denormalized form ?
  - Special values ?

len: 1 **k** = 8 or 11 23 or 52

- How do we encode 0?
- $(-1)^s * M * 2^E = (-1)^s * 1.\text{fraction} * 2^{\text{exp} - 2^{k-1} + 1}$
- Is there some value of s, fraction and exp?

len: 1 **k** = 8 or 11 23 or 52

- How do we encode 0?
- $(-1)^s * M * 2^E = (-1)^s * 1.\text{fraction} * 2^{\text{exp} - 2^{k-1} + 1}$
- Is there some value of s, fraction and exp?
- No.

Denormalized form

- Activated when exp bits are all 000…000
- M = 0.fraction (previously 1.fraction)
- E = 1 - bias (previously exp – bias)
- i.e., $E = 1 - 2^{k-1} + 1$
- Allows us to encode 0 and very small numbers!

Denormalized form

- Activated when exp bits are all 000…000
- M = 0.fraction (previously 1.fraction)
- E = 1 - bias (previously exp - bias)
- Allows us to encode 0 and very small numbers!
- In denormalized form, why is E always the same?

How do we encode 0 in 32-bit float?

- Use denormalized form
- $(-1)^s * M * 2^E = (-1)^s * 0.\text{fraction} * 2^{1 - 2^{k-1} + 1}$
- Set s = 0
- E is always $1 - 2^{k-1} + 1$
- M = 0.?????...

How do we encode 0 in 32-bit float?

- Use denormalized form
- $(-1)^s * M * 2^E = (-1)^s * 0.\text{fraction} * 2^{1 - 2^{k-1} + 1}$
- Set s = 0
- E is always $1 - 2^{k-1} + 1$
- M = 0.00000…
- $(-1)^0 * 0.00000\ldots * 2^{-126} = 0$

How do we encode 0 in 32-bit float?

- Use denormalized form
- $(-1)^s * M * 2^E = (-1)^s * 0.\text{fraction} * 2^{1 - 2^{k-1} + 1}$
- Set s = 0 (what happens if s = 1?)
- E is always $1 - 2^{k-1} + 1$
- M = 0.00000…
- $(-1)^0 * 0.00000\ldots * 2^{-126} = 0$

How do we encode 0 in 32-bit float?

- Use denormalized form
- $(-1)^s * M * 2^E = (-1)^s * 0.\text{fraction} * 2^{1 - 2^{k-1} + 1}$
- Set s = 0 (what happens if s = 1?)
- E is always $1 - 2^{k-1} + 1$
- M = 0.00000…
- $(-1)^1 * 0.00000\ldots * 2^{-126} = 0$

We have two zeros…

- Positive zero
- Negative zero
- 

Degenerate cases

- Now we have described *denormalized form*
- There are three forms:
  - Normalized form ✅
  - Denormalized form ✅
  - Special values ?

Special values

- Activated when exp bits are all 111…111
- If frac is all 000...000:
  - Infinity
- Otherwise:
  - NaN (not a number)

Infinity

- Can reach if values get too high
  - Floating point does not overflow
  - “Saturates” to infinity (or negative infinity)
- 123.0 / 0.0

NaN

- Result from weird operations that can’t even be described as infinity
- sqrt(-1)
- infinity + negative infinity

NaN

- Has some bizarre properties too!
- NaN plus any other math operation → NaN
  - NaN + 2.0 → NaN
  - 3.0 * NaN → NaN
- NaN is not equal to anything, including itself
  - NaN == NaN → False

Floating point exercises

value = $(-1)^s \times M \times 2^E$

sign (s), significand (M), exponent (E)

len: 1 **k** = 8 or 11 23 or 52

Normalized

E = exp (unsigned) **- bias**

bias = $2^{k-1} - 1$

M = 0b1.frac (binary)

Denormalized

when exp = 00...00

E = 1 - bias

M = 0b0.frac (binary)

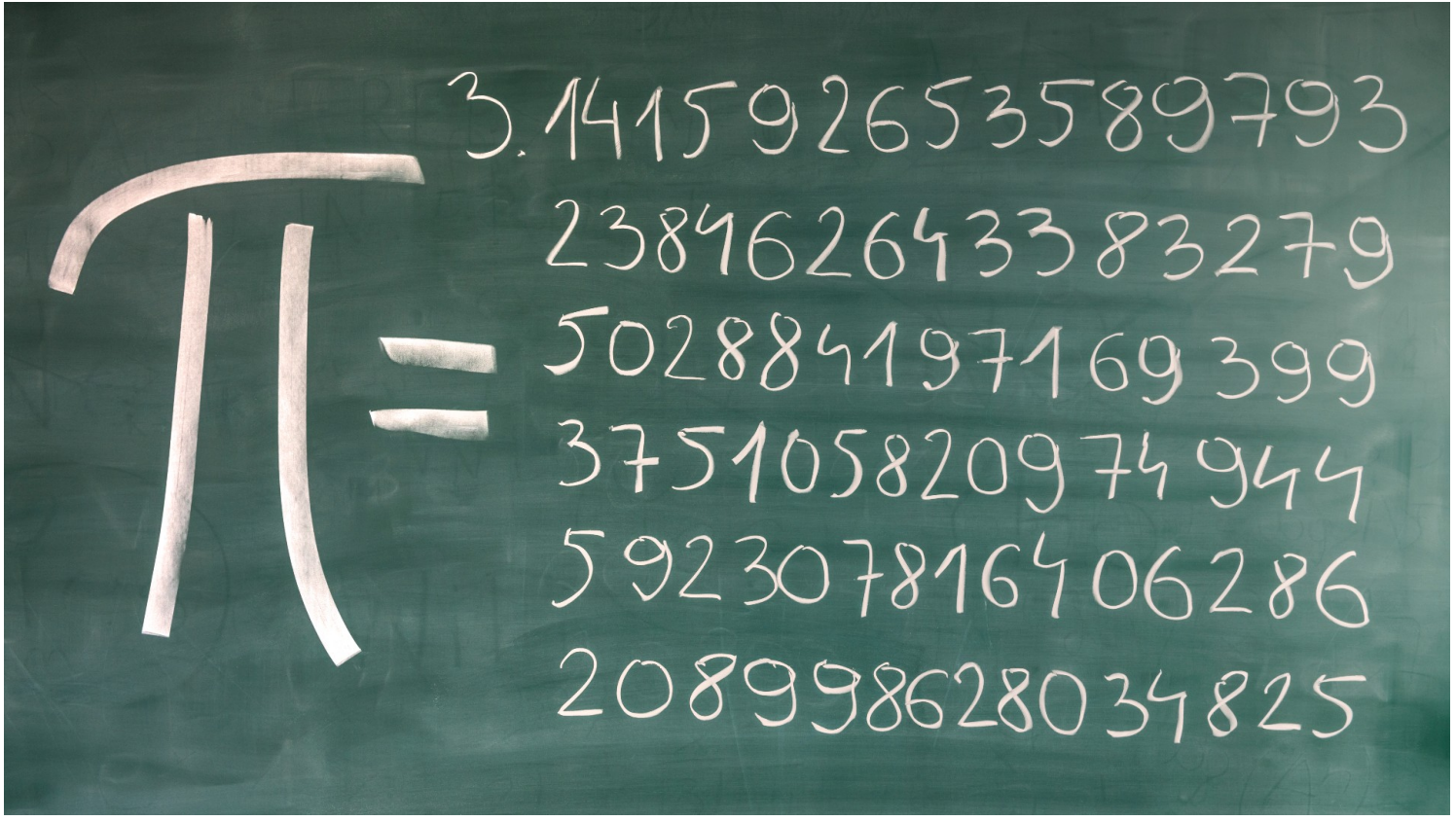
Special

when exp = 11...11

(a) frac = 00...00: **Infinity**

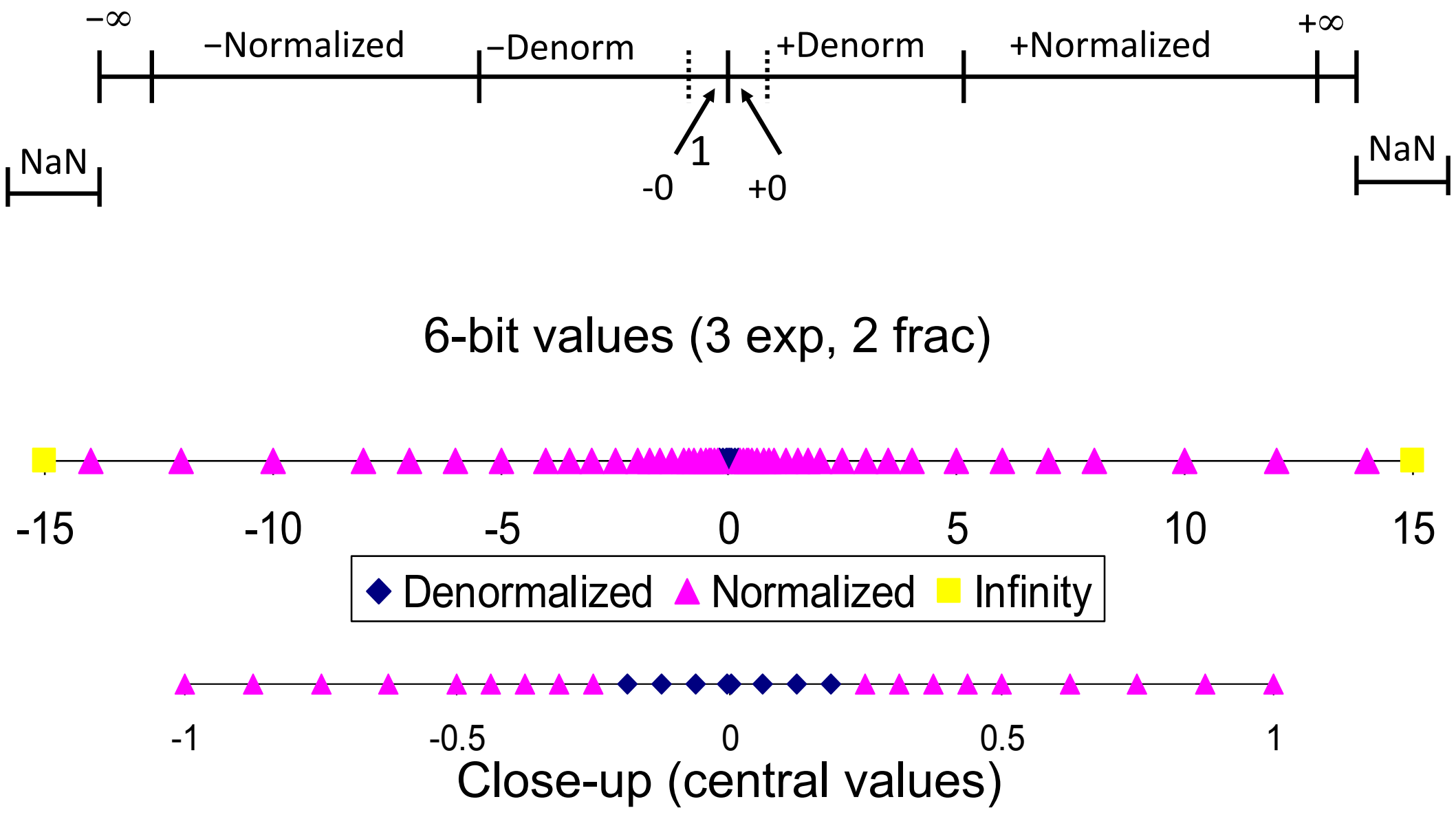
(b) frac ≠ 00...00: **NaN**

Why can’t we be exact?



Some common values cannot be exactly represented

- Some same as decimal (1/3):
  - Dec: 0.333…
  - Bin: 0.010101…
- Some new (1/10):
  - Dec: 0.1
  - Bin: 0.0001100110011…


−∞
−Normalized
−Denorm
+Denorm
+Normalized
+∞
NaN
1
-0
+0
NaN
6-bit values (3 exp, 2 frac)
-15
-10
-5
0
5
10
15
Denormalized
Normalized
Infinity
-1
-0.5
0
0.5
1
Close-up (central values)

Floating point number line

- How many floating point values can a 6-bit float represent?

Floating point number line

- How many floating point values can a 6-bit float represent? $2^6 = 64$

Floating point number line

- How many floating point values can a 6-bit float represent? $2^6 = 64$
- Numbers are denser closer to zero
- More bits means larger floats but also denser floats

Floating point rounding

- When we can’t represent a value exactly, we have to round
- Floating point as many rounding modes
- The specifics are part of that 70 page document 🥲
- We won’t learn how to do floating point arithmetic
- Done by specialized hardware

Not even Python will save us

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.3
0.3
```

float.py

```
print(10.0**20      - 10.0**20)     # 10^20      - 10^20
print(10.0**20      - 10.0**20 + 1) # 10^20      - 10^20 + 1
print(10.0**20 + 1 - 10.0**20)      # 10^20 + 1 - 10^20
print(10.0**20 + 10000 - 10.0**20) # 10^20 + 10000 - 10^20
```

Associativity does not hold

- Associativity: (a + b) + c = a + (b + c)
- Not true for floating point!

Positive vs. negative zero

```c
#include <stdio.h>

int main(void) {
    printf("0.0 * 5.0 = %f\n", 0.0 * 5.0);
    printf("-0.0 * 5.0 = %f\n", -0.0 * 5.0);
    return 0;
}
```

Positive vs. negative zero

```c
#include <stdio.h>

int main(void) {
    printf("0.0 * 5.0 = %f\n", 0.0 * 5.0);
    printf("-0.0 * 5.0 = %f\n", -0.0 * 5.0);
    return 0;
}
```

```
$ ./zeros
0.0 * 5.0 = 0.000000
-0.0 * 5.0 = -0.000000
```

Positive vs. negative zero

```c
#include <stdio.h>

int main(void) {
    printf("1.0 / 0.0 = %f\n", 1.0 / 0.0);
    printf("1.0 / -0.0 = %f\n", 1.0 / -0.0);
    return 0;
}
```

Positive vs. negative zero

```c
#include <stdio.h>

int main(void) {
    printf("1.0 / 0.0 = %f\n", 1.0 / 0.0);
    printf("1.0 / -0.0 = %f\n", 1.0 / -0.0);
    return 0;
}
```

```
$ ./zeros2
1.0 / 0.0 = inf
1.0 / -0.0 = -inf
```

Use in C code

```
int a                = 3;
long b               = 3L;
unsigned int c       = 3u;
unsigned long d      = 3uL;
float x              = 3.0f;
double y             = 3.0;
```

Casting in C

```
// signed <-> unsigned does not change bit representation:
unsigned int a = (unsigned int) -1;
// int <-> double changes bit representation:
unsigned int b = (unsigned int) 3.0;
double x = (double) 3;
```

Wider than double?!

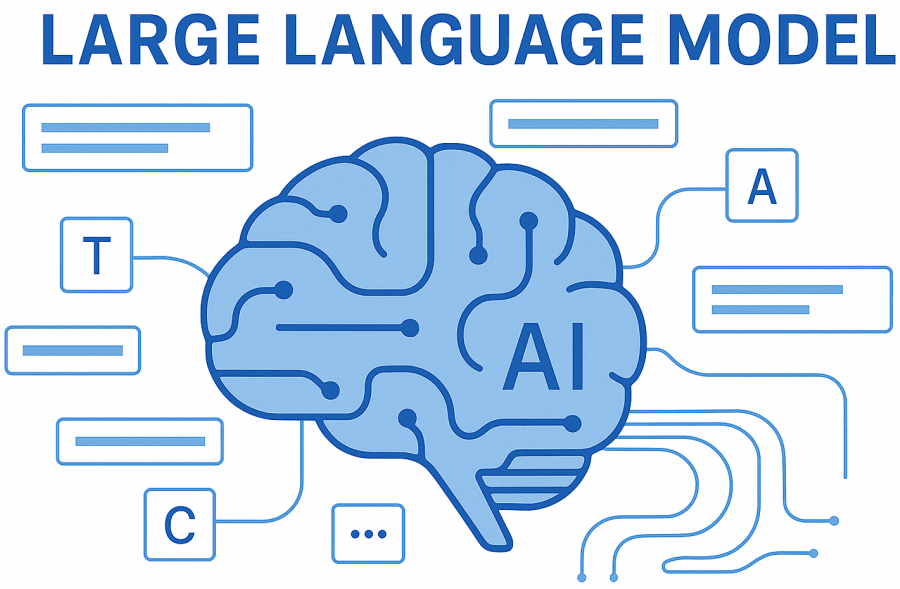
- float: “single precision”: 32 bits
- double: “double precision”: 64 bits
- x86 machines such as krebs have a *long double*
- 80 bits: 1 sign bit, 15-bit exponent, 64-bit fraction

Should I use float or double?

- Usually double
- Even if you don’t need more size, more precision is nice!
- Floats take less memory, computation is faster

Sometimes precision isn’t necessary

- Large language models (LLMs) typically use 16-bit “half precision” floats
- Quantized models often use 8-bit or 4-bit floats!



Floating point is dangerous

Not even Python will save us

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.3
0.3
```

Alternatives

- Fixed point
  - $1343.77 → 134377 cents
- Rational numbers
  - Keep track of numerator and denominator

Rationals in Python

```
>>> from fractions import Fraction
>>> Fraction(1, 10) + Fraction(1, 10) + Fraction(1, 10)
Fraction(3, 10)
```

Rationals downside

- 47985420975943853245439857324095742390875942307508943750984237509874320859742309857340289573241049823108947230917483902147329123498231904723917483902147329123498231904
- Some numbers cannot be represented as rationals
  - Transcendental: π
  - Irrational: √2

TODO

- Assignment 1 is assigned (due September 24)