CIS341
Floating point representation

Bryan S. Kim

Assistant Professor

Syracuse University

College of Engineering & Computer Science
Department of Electrical Engineering & Computer Science

Sep. 14, 2023

Slides based on contents from CS61C @ UC Berkeley & 15-213 @ CMU

Last lecture: integer representation

Crux of the problem: we can't have infinite bits

Two's complement representation

- Limited number of bit, potential overflow

Signed vs. unsigned

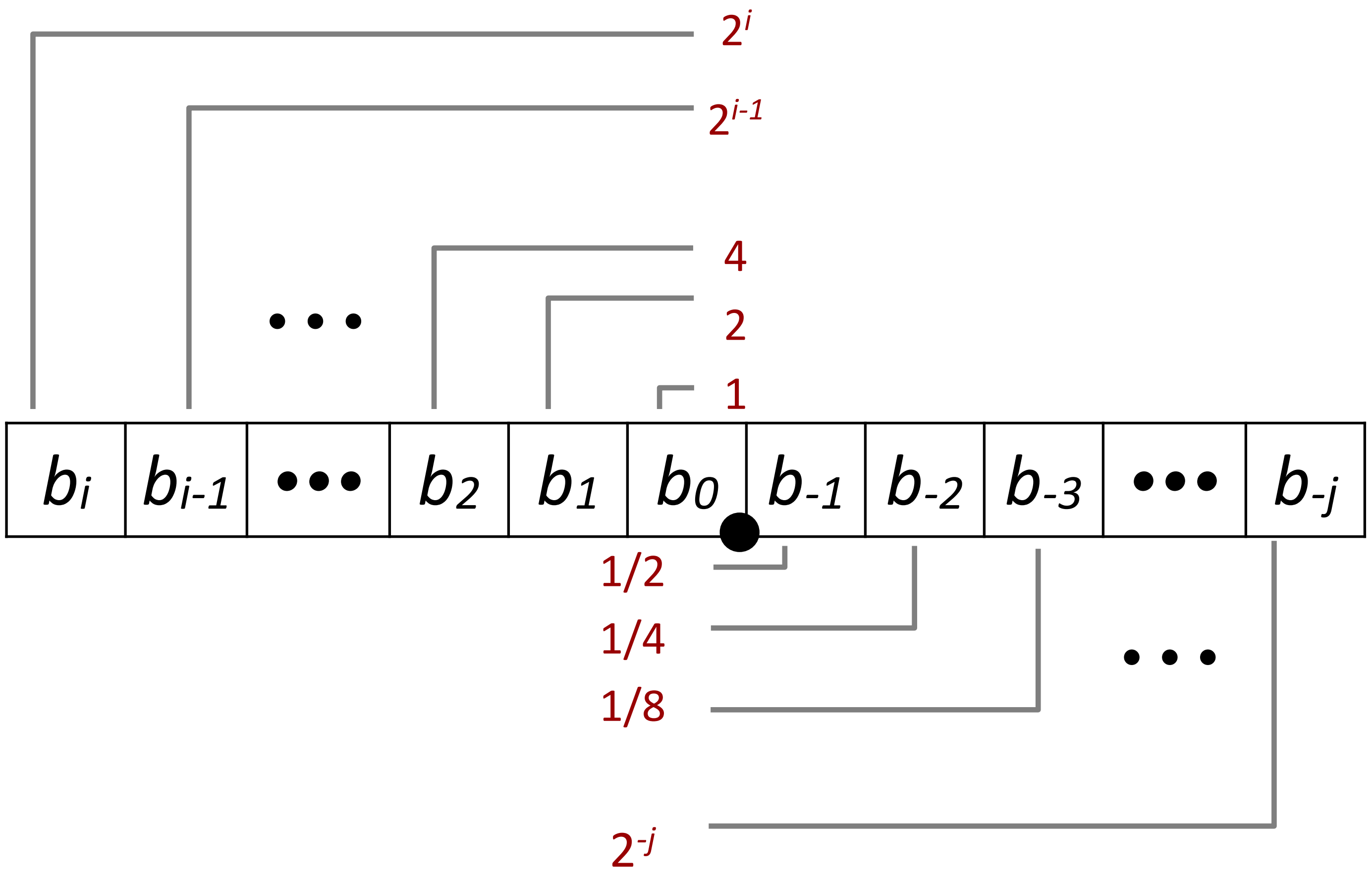
- Same bit representation
- Different interpretation
- Sometimes different operations (example: right shift)
- What happens when they are mixed

Fractional binary numbers

What is 1011.101_2?

Fractional binary numbers

Bits right of the binary point represent fractional powers of 2



Fractional binary numbers: example

Value Representation

Value	Representation	
5 & 3/4 = 23/4	101.11_2	= 4 + 1 + 1/2 + 1/4
2 & 7/8 = 23/8	10.111_2	= 2 + 1/2 + 1/4 + 1/8
1 & 7/16 = 23/16	1.0111_2	= 1 + 1/4 + 1/8 + 1/16

Observations

- Divide by 2 by shifting right
- Multiply by 2 by shifting left
- Numbers of form $0.11111\ldots_2$ are almost 1.0
  - $1/2 + 1/4 + 1/8 + \ldots + 1/2^i$ converge to 1.0

Representable numbers

Limitation #1

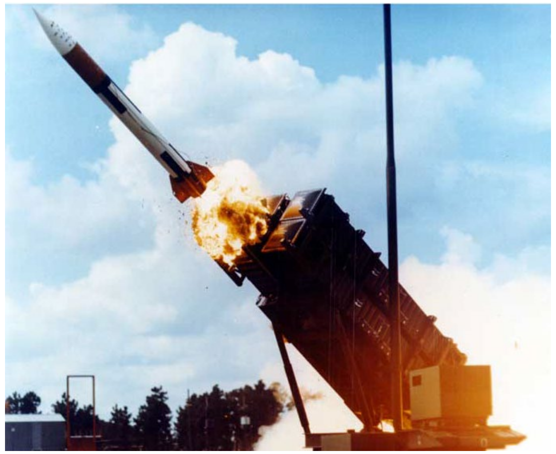
- Can only exactly represent numbers of the form $x/2^k$
  - In decimal, 1/3 is 0.3333333…
  - In binary, 1/3 is 0.010101010101…

Limitation #2

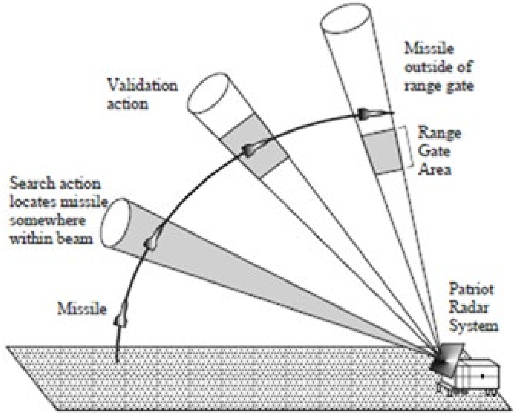
- Just one setting of binary point within the w-bits

The importance of floating point

Missile interception failure on Feb. 25, 1991



- Internal system clock incremented every 0.1 seconds, and to compute 1 second, counter was multiplied by 1/10.
- $1/10 = 0.0001100110011001100\ldots_2$, and this was stored as 24-bits.
- Introduced an error of $0.0000000000000000000000011001100_2$, or 0.000000095 seconds for every 0.1 second.
- After 100 hours of operation, the error was large enough to miss the interception.



IEEE floating point (IEEE 754)

Established in 1985 as a uniform standard for floating point

Supported by all major CPUs

- Some don’t fully implement IEEE 754 (some GPUs)

Driven by numerical concerns

- Numerical analysts predominated over hardware designers in defining the standard
- Hard to make fast in hardware

Floating point representation

$341_{10} = (-1)^0 \times 1.01010101 \times 2^8$

Numerical form

- $(-1)^s \times M \times 2^E$
- Sign bit s determines whether the number if negative or positive
- Mantissa M is normally a fractional value in range of [1.0, 2.0)
- Exponent E weighs value by power of two

Encoding

- MSB **s** is sign bit s
- **exp** field encodes E (but is not equal to E)
- **frac** field encodes M (but is not equal to M)

s	exp	frac

Precision options

Single precision (32-bits): `float`

≈ 7 decimal digits, $10^{\pm 38}$

s	exp	frac
1	8-bits	23-bits

Double precision (64-bits): `double`

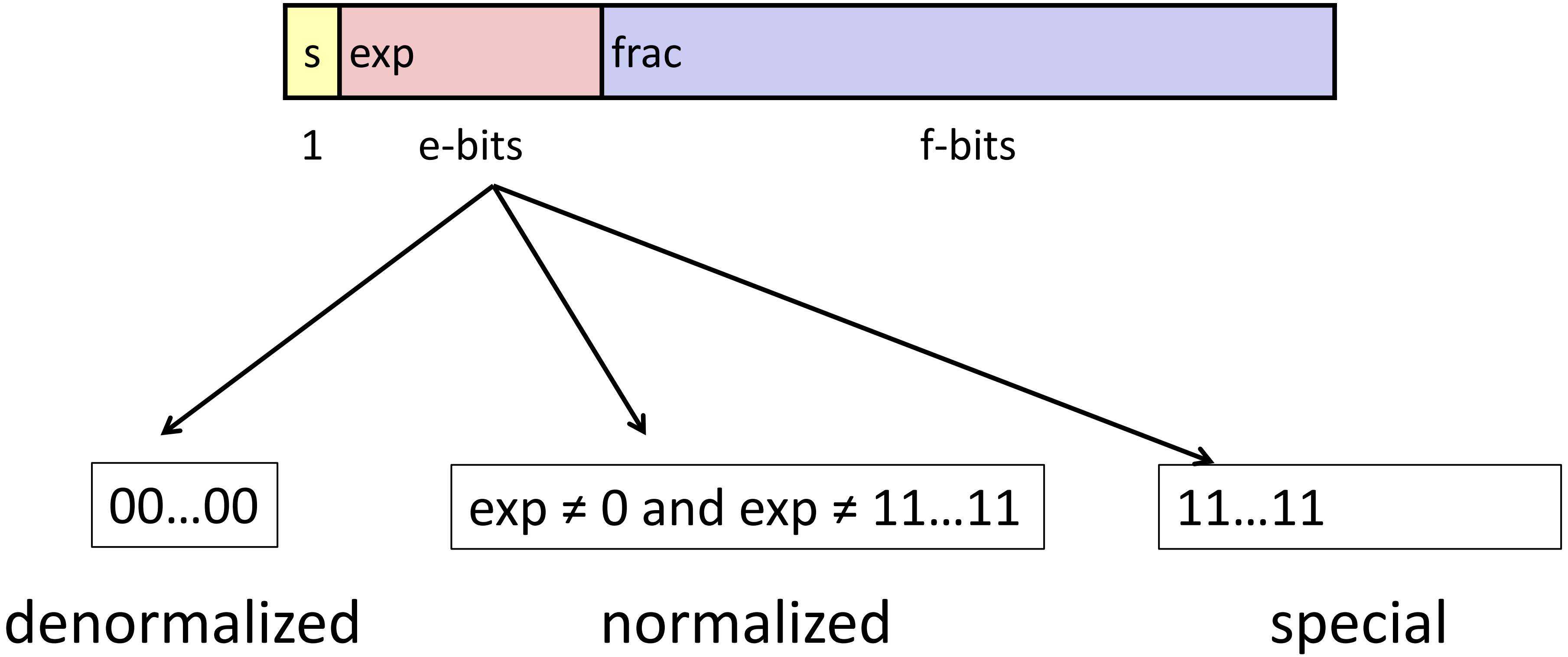
≈ 16 decimal digits, $10^{\pm 308}$

s	exp	frac
1	11-bits	52-bits

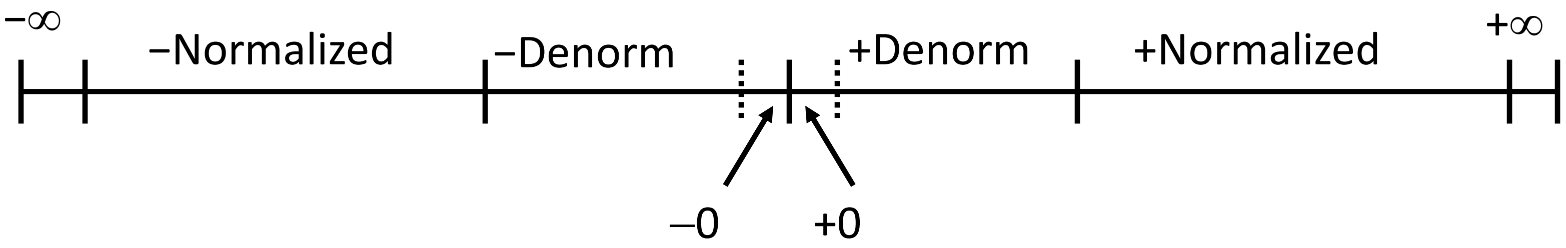
Other formats exist

- Half precision (16-bits)
- Quad precision (128-bits): `long double`

Three kinds of floating point numbers



Visualization of the floating point encodings



Normalized values

$v = (-1)^s \, M \, 2^E$

When exp ≠ 000…0 and exp ≠ 111…1

Exponent coded as a biased value: E = exp – Bias

- `exp` : unsigned value of `exp` field
- Bias: $2^{k-1}-1$, where k is the number of exponent bits
  - Single precision: 127 (`exp` : 1 to 254, E: -126 to 127)
  - Double precision: 1023 (`exp`: 1 to 2046, E: -1022 to 1023)

Mantissa coded with implied leading 1: $M = 1.xxx…x_2$

- xxx.x: bits of the `frac` field
- Minimum when `frac` is 000…0 (M = 1.0)
- Maximum when `frac` is 111…1 (M ≈ 2.0)

Normalized encoding example

$v = (-1)^s \, \boldsymbol{M} \, 2^{\boldsymbol{E}}$
$\boldsymbol{E} = \text{exp} - \boldsymbol{Bias}$

```
float f = 341.0;
```

$341 = 101010101_2 = 1.01010101 \times 2^8$

Mantissa

- M = 1.01010101
- frac = $\underline{01010101}000000000000000_2$

Exponent

- E = 8
- Bias = 127
- Exp = 135 = 10000111_2

Result

0	10000111	01010101000000000000000
s	exp	frac

Denormalized values

$v = (-1)^s \, \boldsymbol{M} \, 2^{\boldsymbol{E}}$
$\boldsymbol{E} = 1 - \boldsymbol{Bias}$

When exp = 000…0

Exponent value: E = 1- Bias

- Bias is same as the case for normalized ($2^{k-1}-1$)

Mantissa coded with implied leading 0: $M = 0.xxx…x_2$

- xxx.x: bits of the `frac` field

Examples

- exp = 000…0, frac = 000…0
  - Represents zero value (+0 when s = 0, -0 when s=1)
- exp = 000…0, frac ≠ 000…0
  - Equi-spaced numbers close to zero

Special values

When exp = 111…1

When exp = 111…1, frac = 000…0

- Represents infinity (∞)
- + ∞ when s = 0, - ∞ when s = 1

When exp = 111…1, frac ≠ 000…0

- Not-a-number (NAN)
- Example: sqrt(-1), ∞- ∞, ∞×0

C float decoding example

`Float: 0xC0A00000`

C float decoding example

```
Float: 0xC0A00000

Binary: 1100 0000 1010 0000 0000 0000 0000 0000
```

C float decoding example

```
Float: 0xC0A00000

Binary: 1100 0000 1010 0000 0000 0000 0000 0000
```

s	exp	frac
1	1000 0001	010 0000 0000 0000 0000 0000
1	8-bits	23-bits

exp is neither 000…0 or 111…1

Normalized value

E = exp – Bias

M = 1.xxx

C float decoding example

`Float: 0xC0A00000`

`Binary: 1100 0000 1010 0000 0000 0000 0000 0000`

s	exp	frac
1	1000 0001	010 0000 0000 0000 0000 0000
1	8-bits	23-bits

E = exp - Bias = 129 - 127 = 2

S = 1

M = 1.010 0000 0000 0000 0000 0000

= 1 + 1/4 = 1.25

exp is neither 000…0 or 111…1

Normalized value

E = exp – Bias

M = 1.xxx

C float decoding example

`Float: 0xC0A00000`

`Binary: 1100 0000 1010 0000 0000 0000 0000 0000`

s	exp	frac
1	1000 0001	010 0000 0000 0000 0000 0000
1	8-bits	23-bits

E = exp - Bias = 129 - 127 = 2

S = 1

M = 1.010 0000 0000 0000 0000 0000

= 1 + 1/4 = 1.25

v = $(-1)^S \times M \times 2^E = (-1)^1 \times 1.25 \times 2^2 = -5$

exp is neither 000…0 or 111…1

Normalized value

E = exp – Bias

M = 1.xxx

C float decoding example

`Float: 0x001C0000`

C float decoding example

```
Float: 0x001C0000
Binary: 0000 0000 0001 1100 0000 0000 0000 0000
```

C float decoding example

Float: 0x001C0000

Binary: 0000 0000 0001 1100 0000 0000 0000 0000

s	exp	frac
0	0000 0000	001 1100 0000 0000 0000 0000
1	8-bits	23-bits

exp is 000…0

Denormalized value

E = 1 – Bias

M = 0.xxx

C float decoding example

`Float: 0x001C0000`

`Binary: 0000 0000 0001 1100 0000 0000 0000 0000`

s	exp	frac
0	0000 0000	001 1100 0000 0000 0000 0000
1	8-bits	23-bits

E = 1 - Bias = 1 - 127 = -126

S = 0

M = 0.001 1100 0000 0000 0000 0000

= 1/8 + 1/16 + 1/32 = 7/32

exp is 000…0

Denormalized value

E = 1 – Bias

M = 0.xxx

C float decoding example

`Float: 0x001C0000`

`Binary: 0000 0000 0001 1100 0000 0000 0000 0000`

s	exp	frac
0	0000 0000	001 1100 0000 0000 0000 0000
1	8-bits	23-bits

$E = 1 - \text{Bias} = 1 - 127 = -126$

$S = 0$

$M = 0.001\ 1100\ 0000\ 0000\ 0000\ 0000$

$= 1/8 + 1/16 + 1/32 = 7/32$

$v = (-1)^S \times M \times 2^E = (-1)^0 \times 7/32 \times 2^{-126} \approx 2.571393892 \times 10^{-39}$

exp is 000…0
Denormalized value
E = 1 – Bias
M = 0.xxx

Group activity (1 of 2)

Floating point arithmetic

Floating point operations: basic idea

$$x +_f y = \mathrm{Round}(x + y)$$

$$x \times_f y = \mathrm{Round}(x \times y)$$

First **compute the exact result**

Make it fit into desired precision

- Possibly ends up being infinity if exponent too large
- Possibly round to fit into `frac`

Rounding

Should 0.5 round to 0 or 1?

	1.4	1.6	1.5	2.5	-1.5
Towards zero	1 ↓	1 ↓	1 ↓	2 ↓	-1 ↑
Round down	1 ↓	1 ↓	1 ↓	2 ↓	-2 ↓
Round up	2 ↑	2 ↑	2 ↑	3 ↑	-1 ↑
Nearest even*	1 ↓	2 ↑	2 ↑	2 ↓	-2 ↓

*Round to nearest, but if half-way in-between then round to nearest even

Round-to-even

Default rounding mode (other roundings are biased)

If exactly halfway between two possible values

- Round so that the least significant digit is even

Otherwise, round to nearest

Decimal examples (round to nearest hundredth)

7.8749999	7.87	(less than halfway)
7.8750000	7.88	(half way: round up to even)
7.8750001	7.88	(greater than halfway)
7.8800000	7.88	
7.8849999	7.88	(less than halfway)
7.8850000	7.88	(half way: round down to even)

Rounding binary

"Even" when least significant bit is 0

"Half way" when bits right of the rounding position = 100…0

Binary example (round to nearest 1/4)

10.00011	10.00	(less than halfway)
10.00100	10.00	(halfway: round down)
10.00101	10.01	(more than halfway)
10.01000	10.01	
10.01011	10.01	(less than halfway)
10.01100	10.10	(halfway: round up)

Floating point multiplication

$(-1)^{S1}\, M1\, 2^{E1} \times (-1)^{S2}\, M2\, 2^{E2}$

Result format: $(-1)^S\, M\, 2^E$

- Sign s = s1 ^ s2 (note: $(-1)^{s1 \wedge s2}$ is same as $(-1)^{s1+s2}$)
- Mantissa M = $M1 \times M2$
- Exponent E = E1 + E2

Adjustments

- If M ≥ 2, shift M right, increment E
- If E is out of range, overflow into infinity
- Round M to fit `frac` precision

Example (given 4-bit mantissa)

$$1.010 \times 2^2 \times 1.110 \times 2^3 = 10.0011 \times 2^5$$
$$= 1.00011 \times 2^6 = 1.001 \times 2^6$$

Floating point addition

$(-1)^{S1}\, M1\, 2^{E1} + (-1)^{S2}\, M2\, 2^{E2}$ (assume E1>E2)

Result format: $(-1)^S\, M\, 2^E$

- Align and add to get sign s and mantissa M
- Exponent E is E1 (the larger of the two)

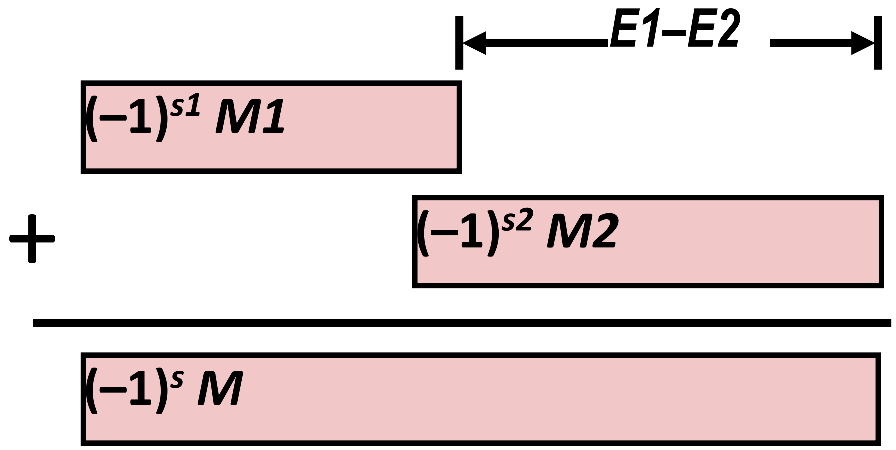
Adjustments

- If M ≥ 2, shift M right, increment E
- If M < 1, shift M left, decrement E
- If E is out of range, overflow into infinity
- Round M to fit `frac` precision

Example (given 4-bit mantissa)

$$1.010\times 2^2 + 1.110\times 2^3 = (0.101+1.110)\times 2^3 = 10.011\times 2^3 = 1.010\times 2^4$$

Get binary points lined up



Floating point arithmetic quirks

fp.c

(3.14 + 1E10) – 1E10 = ?

3.14 + (1E10 - 1E10) = ?

1E20 * (1E20 – 1E20) = ?

1E20 * 1E20 – 1E20 * 1E20 = ?

Floating point in C

`float` *single precision (32-bit)*

`double` *double precision (64-bit)*

Casting between `int`, `float`, and `double` changes bit representation

`double` or `float` into `int`

- Truncates fractional part

`int` into `double`

- Exact conversion (`int` is 32-bits, `double`'s mantissa is 52-bits)

`int` into `float`

- Will round-to-even

Floating point casting quirks

Given `int x; float f; double d;`

False	x == (int) (float) x	
True	x == (int) (double) x	
True	f == (float) (double) f	
False	d == (double) (float) d	
True	f == -(-f)	
False	2/3 == 2/3.0	
True	d < 0.0	➔ ((d*2) < 0.0)
True	d > f	➔ -f > -d
True	d * d >= 0.0	
False	(d+f)-d == f	

Group activity (2 of 2)

To-dos

proj0 (due Yesterday!)

hw0 (due next Wednesday)