Floating Point Numbers

Why do we lose precision?

Floating point errors bothered me a lot when I was taking my Java AP class. I wanted to get to the bottom of it but never got time.

In my junior year and in the summer of 2022, I had been reading about it and could not find a good place to start. Not much information is there on Google that can explain to high school students in simple terms why one gets non-intuitive errors when using floating point number arithmetic.

Special thanks to Mrs. Graciela Elia, my high school Computer science teacher, who has helped and guided us in our journey to understand, program, and use computer science.

Hope this helps the wider community at large to understand Floating Point numbers which I have attempted to explain in a simple way.

Real-world numerical catastrophes.

https://introcs.cs.princeton.edu/java/91float/

- *Ariane 5 rocket.* Ariane 5 rocket exploded 40 seconds after being launched by European Space Agency. Maiden voyage after a decade and 7 billion dollars of research and development. The sensor reported acceleration that so was large that it caused an overflow in the part of the program responsible for recalibrating inertial guidance. 64-bit floating point number was converted to a 16-bit signed integer, but the number was larger than 32,767 and the conversion failed. The unanticipated overflow was caught by a general systems diagnostic and dumped debugging data into an area of memory used for guiding the rocket's motors. Control was switched to a backup computer, but this had the same data. This resulted in a drastic attempt to correct the nonexistent problem, which separated the motors from their mountings, leading to the end of Ariane 5.
- *Patriot missile accident.* On February 25, 1991, an American Patriot missile failed to track and destroy an Iraqi Scud missile. Instead, it hit an Army barracks, killing 26 people. The cause was later determined to be an inaccurate calculation caused by measuring time in tenth of a second. Couldn't represent 1/10 exactly since used a 24-bit floating point. The software to fix the problem arrived in Dhahran on February 26. Here is more information.
- *Intel FDIV Bug* Error in Pentium hardwire floating point divide circuit. Discovered by Intel in July 1994, and rediscovered and publicized by math professor in September 1994. Intel recall in December 1994 cost $300 million. Another floating point bug discovered in 1997.
- *Sinking of Sleipner oil rig.* Sleipner A $700 million platform for producing oil and gas sprang a leak and sank in the North Sea in August, 1991. Error in inaccurate finite element approximation underestimate shear stress by 47% Reference.
- *Vancouver stock exchange.* Vancouver stock exchange index was undervalued by over 50% after 22 months of accumulated roundoff error. The obvious algorithm is to add up all the stock prices Instead a "clever" analyst decided it would be more efficient to recompute the index by adding the net change of stock after each trade. This computation was done using four decimal places and truncating (not rounding) the result to three. Reference

Why double math (0.1d + 0.2d -0.3d) != 0?

```java
public class DoubleDemo {


    private static void demo() {

        double problematicRealNumber =  0.1d + 0.2d - 0.3d;

        if (problematicRealNumber == 0)
            System.out.println("EXPECTED: 0.1d + 0.2d - 0.3d =" + 0);
        else
            System.out.println("UNEXPECTED: 0.1d + 0.2d - 0.3d = " + problematicRealNumber);
    }

    public static void main(String[] args) {
        demo();
    }
}
```

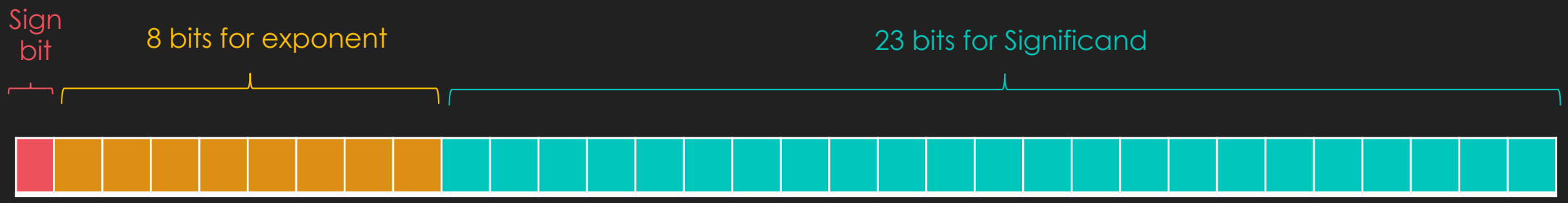
```
UNEXPECTED: 0.1d + 0.2d - 0.3d = 5.551115123125783E-17
```

Whereas float math (0.1f + 0.2f -0.3f) = 0?

```java
public class FloatDemo {


    private static void demo() {

        float realNumber =  0.1f + 0.2f - 0.3f;

        if (realNumber == 0)
            System.out.println("EXPECTED: 0.1f + 0.2f - 0.3f =" + 0);
        else
            System.out.println("UNEXPECTED: 0.1f + 0.2f - 0.3f = " + realNumber);
    }

    public static void main(String[] args) {
        demo();
    }
}
```

```
EXPECTED: 0.1f + 0.2f - 0.3f = 0
```

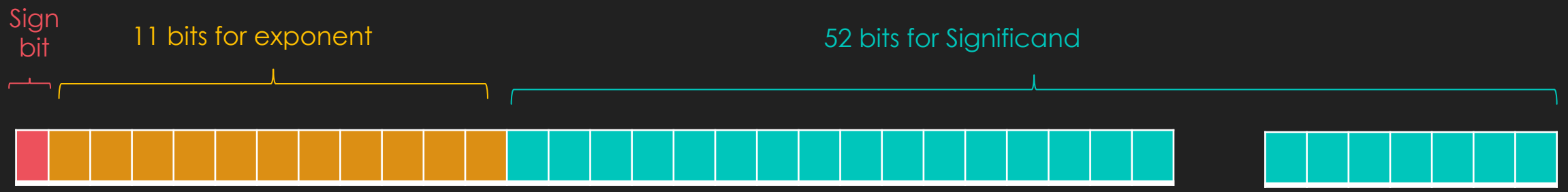
Single Precision Floating Point (32 bits)



float has 23 bits for precision.

In decimal base 10, float can provide precision up to log10(2^23) digits = 6.92 digits (approx.) = 6 to 7 decimal digits of precision

Double Precision Floating Point (64 bits)

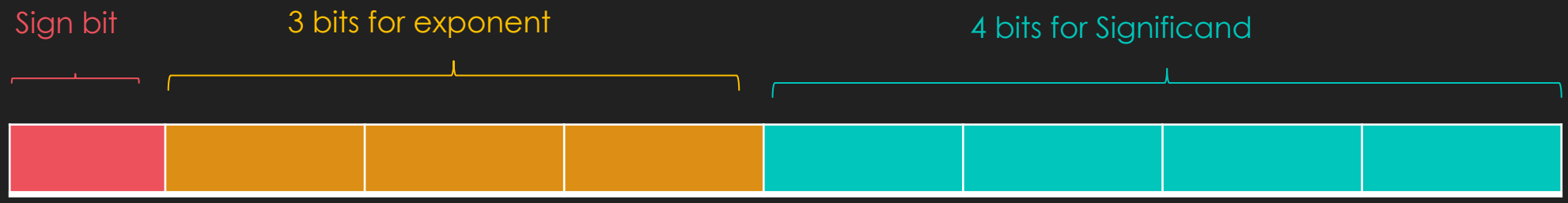


Double has 52 bits for precision.

In decimal base 10, double can provide precision up to log10(2^52) digits = 15.65355977452702 digits (approx.) = 15 to 16 decimal digits of precision

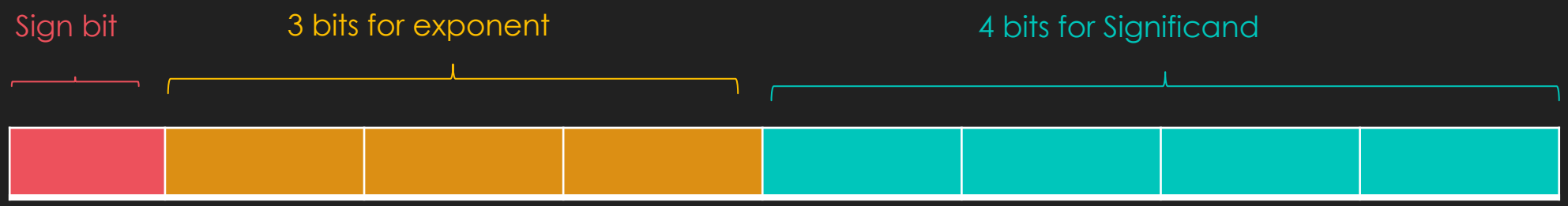
Understanding Floating Point Numbers

- 8 bits are too small to be practical to represent floating point numbers. But we will start with 8 bits, as it becomes easy to understand. Then we can extrapolate to 32 and 64 bit floating point numbers thereafter



Understanding Floating Point Numbers

- Any number that cannot be written as a sum of +ve or –ve powers of 2 CANNOT be exactly represented as a floating-point number e.g. 1/3, 1/5
- Other constraints are the number of exponent and significand bits. We will see this shortly



Why Floating-Point Numbers?

If we have integers, why do we need a new type called float or double?

Integers represent numbers like -3, -1, 0, 1, 3, 6, 2342, …..
Integer division results in an integer. Real numbers between integers cannot be represented by integers

We need a way to represent all real numbers for e.g. 3.334 or 5.0667 and so on

FLOAT and DOUBLE are the 2 floating point numbers that are used by computer systems to represent real numbers

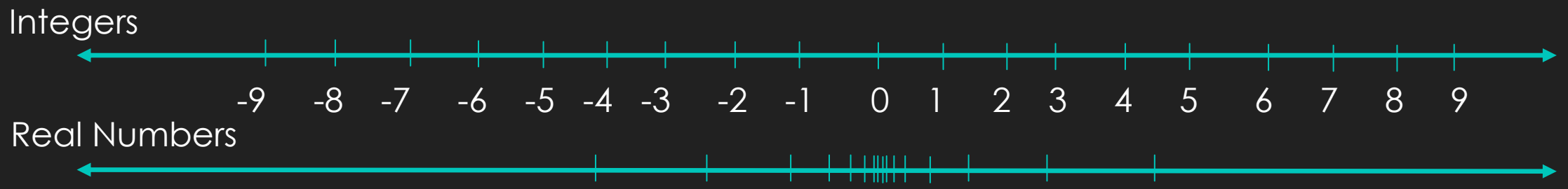
Difficulty representing all real numbers?

- Representing numbers in computers takes memory

An 8-bit memory could represent 2^8 different numbers. If the numbers were signed: we can represent -128 (-2^7) to 127 (2^7 - 1) discrete values

- We have infinite real numbers between two integers how do we represent them then? – But we have finite number of bits to represent them

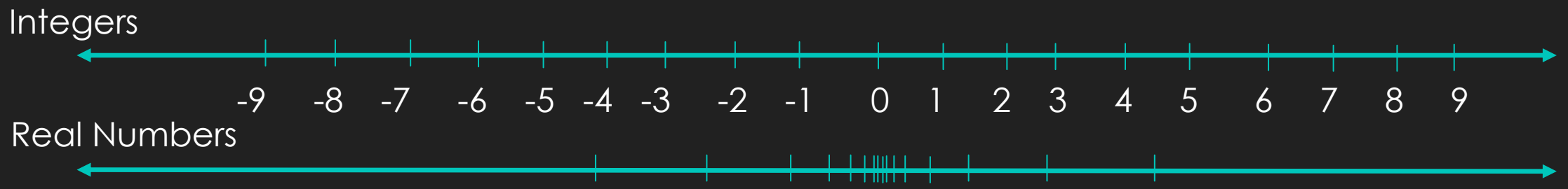
Difficulty representing all real numbers



- If the number of bits is limited and if we use the same number of bits to represent real numbers, the number of real numbers will be limited
- We need real numbers with high precision for shorter distances and not for very large numbers like distance from Earth to the sun.
- We need a way to represent very large numbers much bigger than INT.MAX_VALUE

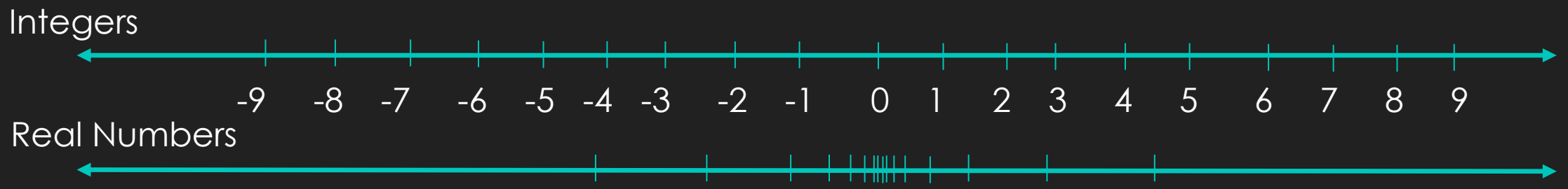
=> As the total number of numbers that a fixed set of bits can represent, we should design a way to represent real numbers very close to each other for higher precision near 0 and more gap between real numbers in computers that are farther away from 0.

Difficulty representing all real numbers



- Can you see why there is loss of precision involving real numbers?

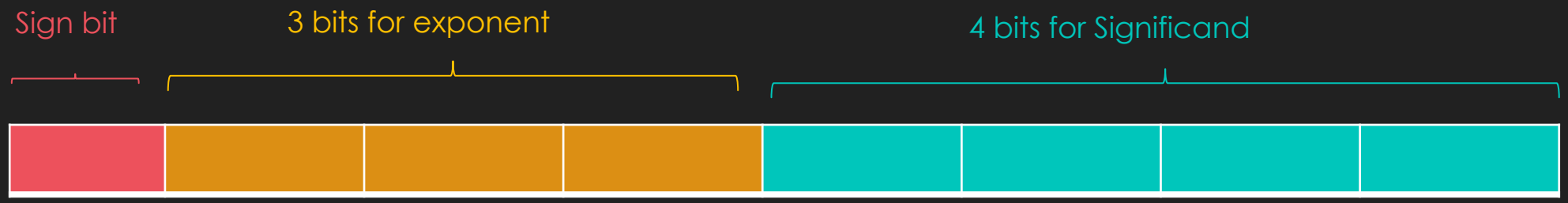
Difficulty representing all real numbers



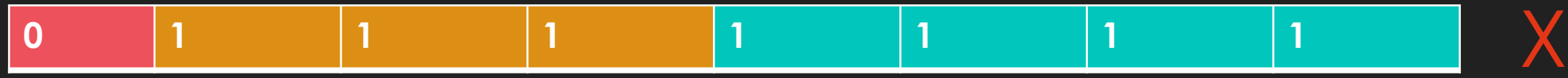
- At a high level - Can you see why there is loss of precision involving real numbers?

WE CAN NOT REPRESENT ALL DISCRETE REAL NUMBERS, AS THERE ARE INFINITE REAL NUMBERS BETWEEN 2 INTEGERS AND WE HAVE FINITE BITS. THIS IS THE REASON FOR THE LOSS OF PRECISION

Difficulty representing all real numbers



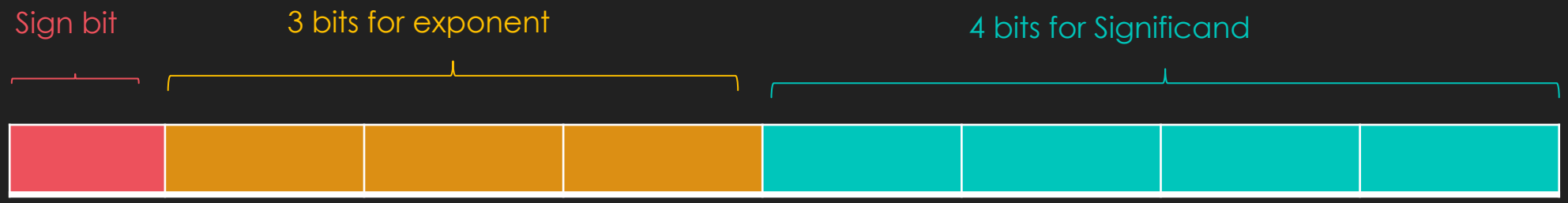
Max Value??



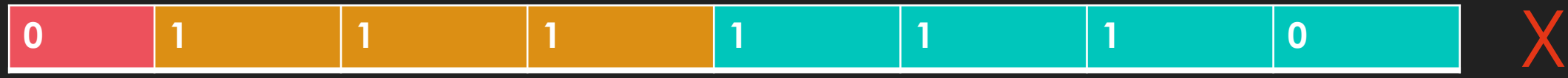
X

All 1's in exponent is reserved as Infinity by IEEE

Difficulty representing all real numbers



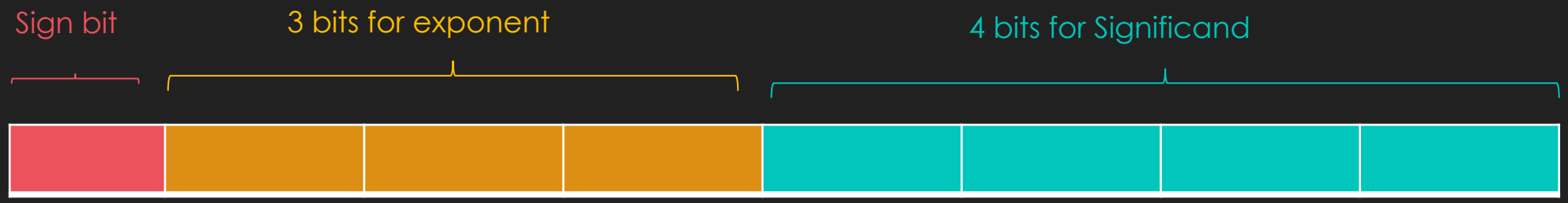
Max Value??



X

This is reserved as NaN (Not a real number) by IEEE
NaN numbers are for example: squareroot(-15), 0/0

How is exponent saved? (for normalized floating-point representation)



Saved exponent = 111 is reserved for infinity

3 bits for the exponent => 2^3 = 8 numbers that can be represented as exponents.
⇒ Given that one of the numbers = 111 (binary) is reserved, the number of exponents = 7

Middle

	000	001	010	011	100	101	110	~~111~~
Saved Exponent	0	1	2	3	4	5	6	~~7~~
Mapped Signed Exponents equally mapped around the Middle of the saved exponent	-3	-2	-1	0	1	2	3	
Bias = 2 ^ (#bits -1) – 1 = value that can be added to signed bit to get a saved exponent	3	3	3	3	3	3	3	

How is exponent saved? (for normalized floating-point representation)

Middle

	000	001	010	011	100	101	110	~~111~~
Saved Exponent	0	1	2	3	4	5	6	~~7~~
Mapped Signed Exponents equally mapped around the Middle of the saved exponent	-3	-2	-1	0	1	2	3	
Bias = 2 ^ (#bits -1) – 1 = value that can be added to signed bit to get a saved exponent	3	3	3	3	3	3	3	

How is exponent saved? (for normalized floating-point representation)

	e3	e2	e1	s4	s3	s2	s1	
0	0	0	0	0	0	0	0	1.0000 x (2 ^ (-3)) = 0.125
0	0	0	1	0	0	0	0	1.0000 x (2 ^ (-2)) = 0.25
0	0	1	0	0	0	0	0	1.0000 x (2 ^ (-1)) = 0
0	0	1	1	0	0	0	0	1.0000 x (2 ^ (0)) = 1.
0	1	0	0	0	0	0	0	1.0000 x (2 ^ (+1)) = 0.5
0	1	0	1	0	0	0	0	1.0000 x (2 ^ (+2)) = 0.5
0	1	0	1	0	0	0	0	1.0000 x (2 ^ (+3)) = 0.5

In normalized representation there is an assumed binary 1 before the decimal point which is not stored in the floating point

There are 2^4 real numbers that can be represented between any two rows by the floating-point math that has 4 bits for the significand (though there are infinite real numbers between the two rows

How is exponent saved? (for normalized floating-point representation)

In normalized representation there is an assumed binary 1 before the decimal point which is not stored in the floating point

0 0 0 0 0 0 0 0	1.0000 x (2 ^ (-3)) = 0.125
0 0 0 1 0 0 0 0	1.0000 x (2 ^ (-2)) = 0.25
	1.0000 x (2 ^ (-1)) = 0.5
	.0
	.5
	0.5

There are 2^4 real numbers that can be represented between any two rows by the floating-point math that has 4 bits for the significand (though there are infinite real numbers between the two rows

WAIT

We are unable to represent 0 or real numbers between 0 and 0.125.
Now What?

We solve this by storing the floating point in the de-normalized form if the saved exponents are all 000

⇒ We do not assume there is an assumed binary 1 before the decimal point and the mapped exponent for e3e2e1=000 and e3e2e1=001 are both (1-Bias) = -2

⇒ if e3e2e1 = 000, then the floating-point value = 0.s4s3s2s1 x 2 ^ (-2)
And
If e3e2e1 = 001, then the floating point value = 1.s4s3s2s1 x 2 ^ (-2)

How is exponent saved?

	e3	e2	e1	s4	s3	s2	s1	
0	0	0	0	0	0	0	0	0.0000 x (2 ^ (-2)) = 0
0	0	0	1	0	0	0	0	1.0000 x (2 ^ (-2)) = 0.25
0	0	1	0	0	0	0	0	1.0000 x (2 ^ (-1)) = 0
0	0	1	1	0	0	0	0	1.0000 x (2 ^ (0)) = 1.
0	1	0	0	0	0	0	0	1.0000 x (2 ^ (+1)) = 0.5
0	1							^ (+2)) = 0.5
0								(+3)) = 0.5

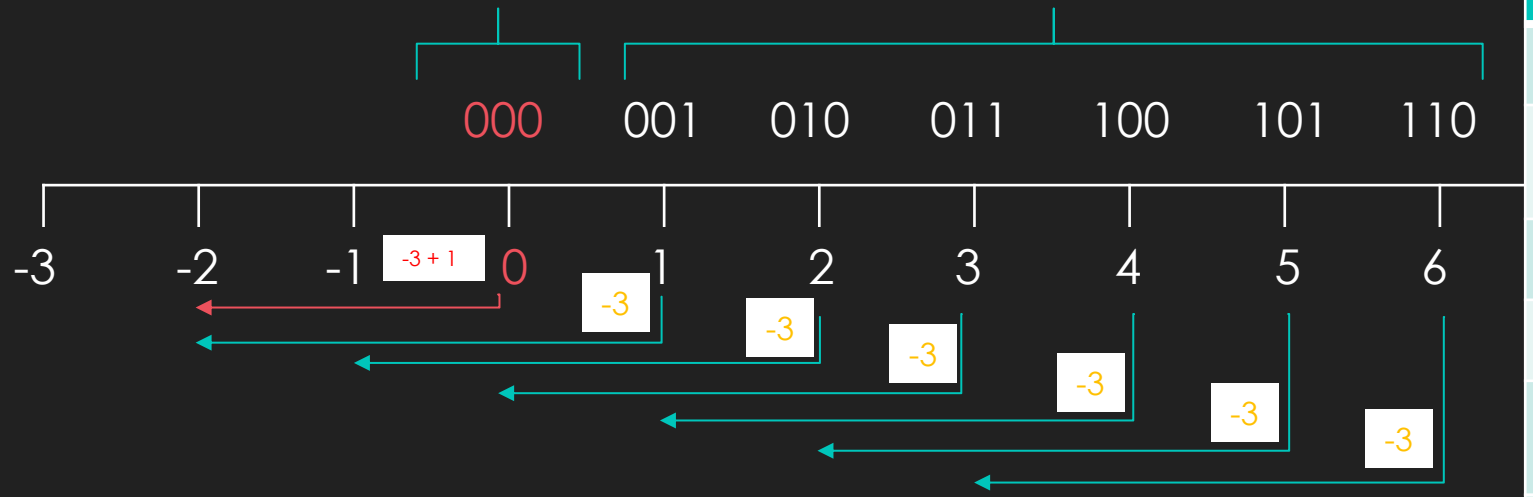
In normalized representation there is an assumed binary 1 before the decimal point which is not stored in the floating point

There are 2^4 real numbers that can be represented between any two rows by the floating-point math that has 4 bits for the significand (though there are infinite real numbers between the two rows

In de-normalized representation when saved exponent e3e2e1 = 000, there is NO assumed binary 1 before the decimal and the mapped exponent = (1 – bias) = -2

How are 3 bits of the exponent in 8 bits real number representation saved?

- The exponent is not stored in 1s or 2s complement
- Bias = 2^(n-1) – 1. (where n = number of exponent bits)

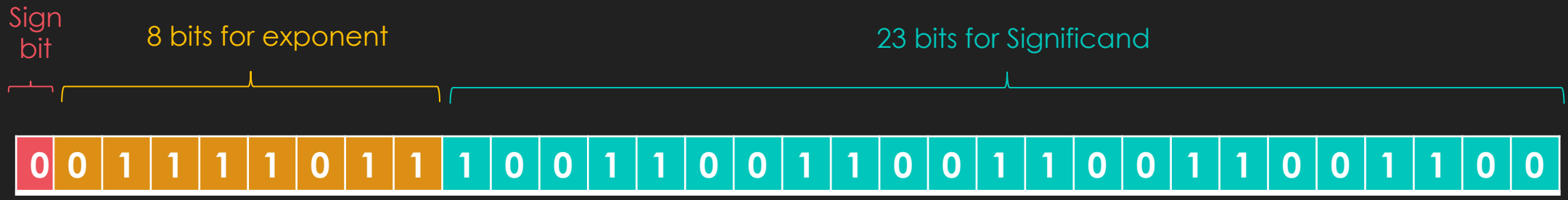


Saved Exponent	Bias	Exponent	Real #
7 (111)	Reserved for infinity		
6 (110)	3	6 – 3 = 3	1.SSSS * (2 ^ 3)
5 (101)	3	5 – 2 = 2	1.SSSS * (2 ^ 2)
4 (100)	3	4 – 3 = 1	1.SSSS * (2 ^ 1)
3 (011)	3	3 – 3 = 0	1.SSSS * (2 ^ 0)
2 (010)	3	2 – 3 = -1	1.SSSS * (2 ^ -1)
1 (001)	3	1 – 3 = -2	1.SSSS* (2 ^ -2)
0 (000)	3	1 – 3 = -2	0.SSSS * (2 ^ -2)

If saved exponent is NOT 000, "mapped exponent" = ("saved exponent" – Bias)
+ floating-point representation is "normalized" => it starts with bit value "1" before the decimal

If saved exponent is 000, "mapped exponent" = (1 – Bias)
+ floating-point representation is "de-normalized" => it starts with bit value "0" before the decimal
This helps in the representation of real numbers between 0 and 1.SSSS x (2 ^ -2)

How is 0.1 decimal represented in 32-bit single precision Java float floating point?



0.1 decimal = 0.0001100110011001100110011001100 in binary ---------→

0.1 * 2 = 0.2 => 0 = 1st digit in binary after the decimal
0.2 * 2 = 0.4 => 0 = 2nd digit in binary after the decimal
0.4 * 2 = 0.8 => 0 = 3rd digit in binary after the decimal
0.8 * 2 = 1.6 => 1 = 4th digit in binary after the decimal
0.6 * 2 = 1.2 => 1 = 5th digit in binary after the decimal
0.2 * 2 = 0.4 => 0 = 6th digit in binary after the decimal
0.4 * 2 = 0.8 => 0 = 7th digit in binary after the decimal
0.8 * 2 = 1.6 => 1 = 8th digit in binary after the decimal
0.6 * 2 = 1.2 => 1 = 9th digit in binary after the decimal
0.2 * 2 = 0.4 => 0 = 10th digit in binary after the decimal
0.4 * 2 = 0.8 => 0 = 11th digit in binary after the decimal
0.8 * 2 = 1.6 => 1 = 12th digit in binary after the decimal

In Normalized form:

1.10011001100110011001100 x 2 ^ (-4)

For 32 bit floating point:

Significand (23 bits) = 10011001100110011001100
Saved exponent – Bias = Exponent
⇒ Saved Exponent = Exponent + Bias = -4 + (2^(8-1) - 1) = 123 decimal = 1111011

Difficulty representing all real numbers

What is the Max real value we can represent using 8 bits?

Exponent + Bias = (110)binary
=> Exponent = (110)binary – Bias = 6 – Bias
=> Bias = 2 ^ (3 – 1) – 1 = 3
⇒ Max Value is (1.1111)binary x 2 ^ (6 – 3)
⇒ ((1*2^0) + (1* 2^-1) + (1* 2^-2) + (1* 2^-3) + (1* 2^-4)) * 2 ^ 4
⇒ (1+0.5 + 0.25 + 0.125 + 0.0625) * 8 = 1.9375 * 8 = 15.5

15.5 is not that big a real value

This is the reason why
- float has 7 bits for the exponent to have bigger MAX values
- double has 11 bits for the exponent to have even bigger MAX values

Similar reasoning can be applied why
- float has 23 bits for the significand to have higher precision (5 or 6 decimal digits of precision)
- double has 52 bits for the significand to have even higher precision (15 or 16 decimal digits of precision)

Difficulty representing all real numbers

What is the Min Positive Real Value after zero?

Bias = 2 ^ (3 – 1) – 1 = 3 (we have 3 bits for the exponent)
Exponent + Bias = (1)binary if above 3 bits are 0 (exponent is NOT stored in Scientific form when it is 0)
=> Exponent = (1)binary – Bias = 1 – 3 = -2

The above number = (0)binary * 2 ^(-2) = 0

Next Positive value:

= (0.0001)binary * 2 ^(-2) = (0.0625) * (1/4) = 0.015625

Next Positive value:

= 0.125 * (1/4) = 0.03125

Difficulty representing all real numbers

Next Positive value:

= (0.0011)binary * (1/4) = (0.125 + 0.0625) * 1/4 = 0.046875

Next Positive value:

= (0.01)binary * (1/4) = (0.25) * 1/4 = 0.0625

Next Positive value:

= (0.0101)binary * (1/4) = (0.25 + 0.0625) * 1/4 = 0.078125

Max Positive value when saved exponent = 000:

= (0.1111)binary * (1/4) = (0.5+ 0.25 + 0.125 + 0.0625) * 1/4 = 0.234375

Difficulty representing all real numbers

What is the Min Positive Real Value when saved exponent = 1?

Bias = 2 ^ (3 – 1) – 1 = 3 (we have 3 bits for the exponent)
Exponent + Bias = saved exponent = 1(exponent is stored in Scientific form => the binary number before decimal point = 1)
=> Exponent = (1)binary – Bias = 1 – 3 = -2
The above number = (1.0000)binary * 2 ^(-2) = 0.25

Next Positive value:

= (1.0001)binary * 2 ^(-2) = (1.0 + 0.0625) * (1/4) = 0.265625

Next Positive value:

= 1.0 + 0.125 * (1/4) = 0.28125

Difficulty representing all real numbers

Next Positive value:

= (1.0011)binary * (1/4) = (1.0 + 0.125 + 0.0625) * 1/4 = 0.296875

Next Positive value:

= (1.01)binary * (1/4) = (1.0 + 0.25) * 1/4 = 0.3125

Next Positive value:

= (1.0101)binary * (1/4) = (1.0 + 0.25 + 0.0625) * 1/4 = 0.328125

Max Positive value when saved exponent = 001:

= (1.1111)binary * (1/4) = (10 + 0.5+ 0.25 + 0.125 + 0.0625) * 1/4 = 0.484375

Difficulty representing all real numbers

What is the Min Positive Real Value when saved exponent = 1?

Bias = 2 ^ (3 – 1) – 1 = 3 (we have 3 bits for the exponent)
Exponent + Bias = saved exponent = 1(exponent is stored in Scientific form => the binary number before decimal point = 1)
=> Exponent = (10)binary – Bias = 2 – 3 = -1
The above number = (1.0000)binary * 2 ^(-1) = 0.5

Next Positive value:

= (1.0001)binary * 2 ^(-1) = (1.0 + 0.0625) * (1/2) = 0.53125

Next Positive value:

= 1.0 + 0.125 * (1/2) = 0.5625

Difficulty representing all real numbers

Next Positive value:

= (1.0011)binary * (1/2) = (1.0 + 0.125 + 0.0625) * 1/2 = 0.59375

Next Positive value:

= (1.01)binary * (1/2) = (1.0 + 0.25) * 1/2 = 0.625

Next Positive value:

= (1.0101)binary * (1/2) = (1.0 + 0.25 + 0.0625) * 1/2 = 0.65625

Max Positive value when saved exponent = 010:

= (1.1111)binary * (1/2) = (10 + 0.5+ 0.25 + 0.125 + 0.0625) * 1/2 = 0.96875

Difficulty representing all real numbers

What is the Min Positive Real Value when saved exponent = 1?

Bias = 2 ^ (3 – 1) – 1 = 3 (we have 3 bits for the exponent)
Exponent + Bias = saved exponent = 1(exponent is stored in Scientific form => the binary number before decimal point = 1)
=> Exponent = (10)binary – Bias = 3 – 3 = 0
The above number = (1.0000)binary * 2 ^(0) = 1.0

Next Positive value:

= (1.0001)binary * 2 ^(0) = (1.0 + 0.0625) = 1.0625

Next Positive value:

= 1.0 + 0.125 = 1.125

Difficulty representing all real numbers

Next Positive value:

= (1.0011)binary * (1) = (1.0 + 0.125 + 0.0625) = 1.1875

Next Positive value:

= (1.01)binary * (1) = (1.0 + 0.25) = 1.25

Next Positive value:

= (1.0101)binary * (1) = (1.0 + 0.25 + 0.0625) = 1.3125

Max Positive value when saved exponent = 010:

= (1.1111)binary * (1) = (1.0 + 0.5+ 0.25 + 0.125 + 0.0625) = 1.9375

Difficulty representing all real numbers:

Difference between two consecutive real numbers doubles when the exponent is incremented by 1

	Real Number	Difference between 2 consequitive real numbers	
when saved exponent = 000	0		difference bertween 2 consecutive real numbers = 0.015625
	0.015625	0.015625	
	0.03125	0.015625	
	0.046875	0.015625	
	0.0625	0.015625	
	0.078125	0.015625	
	0.234375		
when saved exponent = 001	0.25	0.015625	difference bertween 2 consecutive real numbers = 0.015625
	0.265625	0.015625	
	0.28125	0.015625	
	0.296875	0.015625	
	0.3125	0.015625	
	0.328125	0.015625	
	0.484375	0.015625	
when saved exponent = 010	0.5	0.015625	difference bertween 2 consecutive real numbers = 0.015625 x 2 Note: Difference between 2 consecutive real number doubles, when the exponent is incremented by 1
	0.53125	0.03125	
	0.5625	0.03125	
	0.59375	0.03125	
	0.625	0.03125	
	0.65625	0.03125	
	0.96875	0.03125	
when saved exponent = 011	1	0.03125	difference bertween 2 consequitive real numbers = 0.015625 x 2 X 2 Note: Difference between 2 consequitive real number is now 4 times, when the exponent is incremented by 1
	1.0625	0.0625	
	1.125	0.0625	
	1.1875	0.0625	
	1.25	0.0625	
	1.3125	0.0625	
	1.9375	1.9375	

Difficulty representing all real numbers:

- If 8 bits are used to represent real numbers, the total number of real numbers = 2^8
- The numbers near to zero are closer and the distance between the two numbers doubles with every increment of the exponent. Numbers farther away from 0 are further away. Max number as shown before using 8-bit for real numbers cannot go beyond 15.5 though

15.5 is not that big a real value and is the MAX one can represent using 8 bits

This is the reason why
- float has 7 bits for the exponent to have bigger MAX values
- double has 11 bits for the exponent to have even bigger MAX values

Similar reasoning can be applied why
- float has 23 bits for the significand to have higher precision
- double has 52 bits for the significand to have even higher precision

Why double math (0.1d + 0.2d -0.3d) != 0?

```java
public class DoubleDemo {


    private static void demo() {

        double problematicRealNumber =  0.1d + 0.2d - 0.3d;

        if (problematicRealNumber == 0)
            System.out.println("EXPECTED: 0.1d + 0.2d - 0.3d =" + 0);
        else
            System.out.println("UNEXPECTED: 0.1d + 0.2d - 0.3d = " + problematicRealNumber);
    }

    public static void main(String[] args) {
        demo();
    }
}
```

```
UNEXPECTED: 0.1d + 0.2d - 0.3d = 5.551115123125783E-17
```

Why double math (0.1d + 0.2d -0.3d) != 0?

double 0.1= 1111111011100110011001100110011001100110011001100110011001100110011010
[+] [Saved Exponent = 1111111011] [Significand = 1001100110011001100110011001100110011001100110011001100110011010]

double 0.2= 1111111100100110011001100110011001100110011001100110011001100110011010
[+] [Saved Exponent = 1111111100] [Significand = 1001100110011001100110011001100110011001100110011001100110011010]

0.1d + 0.2d = 0.30000000000000004d = 1111111101001100110011001100110011001100110011001100110011001100110100
[+] [Saved Exponent = 1111111101] [Significand = 0011001100110011001100110011001100110011001100110011001100110100]

Why 0.1d + 0.2d = 0.30000000000000004d ?

0.1d Significand = 1001100110011001100110011001100110011001100110011001100110011010
0.2d Significand = 1001100110011001100110011001100110011001100110011001100110011010
+
0011001100110011001100110011001100110011001100110011001100110100

Carry 1 lost

Since the saved exponent is NOT Zero, the number is in normalized form => Exponent = (save exponent - Bias)
Bias = 2^(#of bits for exponent in double – 1) -1 = 2^(11 – 1) -1 = (2^10 -1) => Exponent = (1111111101 – (2^10 -1))= 1021 – 1023 = -2
1.0011001100110011001100110011001100110011001100110011001100110100 * (2 ^ (-2)) = 1.2000000000000017764 / 4 = 0.30000000000000004d
Since double provides 16 digits of decimal precision, the sum is 0.30000000000000004d

Whereas float math (0.1f + 0.2f -0.3f) = 0?

```java
public class FloatDemo {


    private static void demo() {

        float realNumber =  0.1f + 0.2f - 0.3f;

        if (realNumber == 0)
            System.out.println("EXPECTED: 0.1f + 0.2f - 0.3f =" + 0);
        else
            System.out.println("UNEXPECTED: 0.1f + 0.2f - 0.3f = " + realNumber);
    }

    public static void main(String[] args) {
        demo();
    }
}
```

```
EXPECTED: 0.1f + 0.2f - 0.3f = 0
```

Why float math (0.1d + 0.2d -0.3d) = 0?

float 0.1= 11110111001100110011001100111001101

[+] [Saved Exponent = 1111011] [Significand = 10011001100110011001101]

float 0.2= 11111001001100110011001100111001101

[+] [Saved Exponent = 1111100] [Significand = 10011001100110011001101]

0.1f + 0.2f = 0.3d = 11111010011001100110011001110011010

[+] [Saved Exponent = 1111101] [Significand = 00110011001100110011010]

Why 0.1d + 0.2d = 0.3d ?

0.1d Significand = 10011001100110011001101

0.2d Significand = 10011001100110011001101

\+

01100110011001100110100

Carry 1 lost

Since the saved exponent is NOT Zero, the number is in normalized form => Exponent = (save exponent - Bias)

Bias = 2^(#of bits for exponent in float – 1) -1 = 2^(8 – 1) -1 = (2^7 - 1) => Exponent = (1111101 – (2^7 -1))= 125 – 127 = -2

1.00110011001100110011010 * (2 ^ (-2)) = 1.20000004768371582031 / 4 = 0.3000000000000000 4d

Since float provides 6 digits of decimal precision, the sum is 0.3 and not 0.300000000000000004

Real-world numerical catastrophes.

https://introcs.cs.princeton.edu/java/91float/

- *Ariane 5 rocket.* Ariane 5 rocket exploded 40 seconds after being launched by European Space Agency. Maiden voyage after a decade and 7 billion dollars of research and development. The sensor reported acceleration that so was large that it caused an overflow in the part of the program responsible for recalibrating inertial guidance. 64-bit floating point number was converted to a 16-bit signed integer, but the number was larger than 32,767 and the conversion failed. The unanticipated overflow was caught by a general systems diagnostic and dumped debugging data into an area of memory used for guiding the rocket's motors. Control was switched to a backup computer, but this had the same data. This resulted in a drastic attempt to correct the nonexistent problem, which separated the motors from their mountings, leading to the end of Ariane 5.
- *Patriot missile accident.* On February 25, 1991 an American Patriot missile failed to track and destroy an Iraqi Scud missile. Instead it hit an Army barracks, killing 26 people. The cause was later determined to be an inaccurate calculation caused by measuring time in tenth of a second. Couldn't represent 1/10 exactly since used a 24-bit floating point. The software to fix the problem arrived in Dhahran on February 26. Here is more information.
- *Intel FDIV Bug* Error in Pentium hardwire floating point divide circuit. Discovered by Intel in July 1994, and rediscovered and publicized by math professor in September 1994. Intel recall in December 1994 cost $300 million. Another floating point bug discovered in 1997.
- *Sinking of Sleipner oil rig.* Sleipner A $700 million platform for producing oil and gas sprang a leak and sank in the North Sea in August, 1991. Error in inaccurate finite element approximation underestimate shear stress by 47% Reference.
- *Vancouver stock exchange.* Vancouver stock exchange index was undervalued by over 50% after 22 months of accumulated roundoff error. The obvious algorithm is to add up all the stock prices Instead a "clever" analyst decided it would be more efficient to recompute the index by adding the net change of stock after each trade. This computation was done using four decimal places and truncating (not rounding) the result to three. Reference

Lessons

https://introcs.cs.princeton.edu/java/91float/

- Use double instead of float for accuracy.
- Use float only if you really need to conserve memory and are aware of the associated risks with accuracy. Usually, it doesn't make things faster, and occasionally makes things slower.
- Be careful of calculating the difference between two very similar values and using the result in a subsequent calculation.
- Be careful about adding two quantities of very different magnitudes.
- Be careful about repeating a slightly inaccurate computation many many times. For example, calculating the change in the position of planets over time.
- Designing stable floating point algorithms is highly nontrivial. Use libraries when available.