CSC207

Week 11: Floating Point Numbers

UNIVERSITY OF TORONTO MISSISSAUGA

Learning Objective for Today

Learn the basics of floating point representations!

Demo

Motivating Examples …

What just happened?

To understand we need to know how numbers are really stored in a computer.

Integers like 207 or 42 are easy.

Numbers with fractional parts are trickier.

What are integers?

These are stored as binary representations:

byte (8 bit binary)
short, char (16 bit binary)
int (32 bit binary)
long (64 bit binary)

The range of values depends on the number of bits

What's the binary representation of short x = 207?

What are integers?

These are stored as binary representations:

byte (8 bit binary)
short, char (16 bit binary)
int (32 bit binary)
long (64 bit binary)

The range of values depends on the number of bits

What's the binary representation of short x = 207?

0000 0000 1100 1111

$= 1*2^7 + 1*2^6 + 0*2^5 +0* 2^4 +1*2^3 +1* 2^2 +1* 2^1 + 1*2^0$

What are fractions?

These are also stored as binary numbers.

What's the binary representation of 0.0111?

What are fractions?

These are also stored as binary numbers.

What's the binary representation of 0.0111?

$= 0*2^{-1} + 1*2^{-2} + 1*2^{-3} +1*2^{-4}$

$= 0/2 + 1/4 + 1/8 + 1/16$

$= 0.25 + 0.125 + 0.0625$

$= 0.4375$

Finite representation or not: decimals

Real that in decimal, not all fractions can be represented as a finite number of decimal digits.

Examples?

Finite representation or not: decimals

Real that in decimal, not all fractions can be represented as a finite number of decimal digits.

1/3 = 0.333333…

sqrt(2) = 1.4142135 …

What's the necessary condition for a value to be representable with a finite number of decimal digits?

$$v = \sum_{i=1}^{n} d_i \cdot 10^{-i}$$

Finite representation or not: decimals

What’s the necessary condition for a value to be representable with a finite number of binary digits?

Finite representation or not: decimals

What's the necessary condition for a value to be representable with a finite number of binary digits?

$$v = \sum_{i=1}^{n} d_i \cdot 2^{-i}$$

0.5 … finitely expressible in binary or no?

0.25 … finitely expressible in binary or no?

0.75 … finitely expressible in binary or no?

0.825 … finitely expressible in binary or no?

0.1 …. finitely expressible in binary or no?

Finite representation or not: decimals

What's the necessary condition for a value to be representable with a finite number of binary digits?

$$v = \sum_{i=1}^{n} d_i \cdot 2^{-i}$$

0.5 … finitely expressible in binary or no? 0.1

0.25 … finitely expressible in binary or no? 0.01

0.75 … finitely expressible in binary or no? 0.11

0.8125 … finitely expressible in binary or no? 0.1101

0.1 …. finitely expressible in binary or no? 0.00011001100….

We can’t do it all

Computers only have finite memory

We can’t accurately store numbers with infinite binary representations

We only have so many bits!

We therefore have to accept **approximations** of values with a **certain precision.**

If we only have 32 bits, how best to use them?

An idea

If we have 32 bits, use 16 bits for the **non-fractional part** and 16 bits for the **fractional part,** e.g.

0101 1111 0001 0010 . 0101 0111 1111 0001

(decimal value: 24338.3435211181640625)

What's the smallest **fraction** we can represent?

An idea

If we have 32 bits, use 16 bits for the **non-fractional part** and 16 bits for the **fractional part,** e.g.

0101 1111 0001 0010 . 0101 0111 1111 0001

(decimal value: 24338.3435211181640625)

What's the smallest fraction we can represent using this

format?

0. 0000 0000 0000 0001

(1/2^16 = 0.000015258789063)

Want to store 0.000000001 ? No can do! **There must be a better way!**

Back to decimal values

If we want to write a number like

0.00000000000000000000000000000000217

we can use….

Scientific Notation!

2.17 x 10^(-33) or 2.17e-33

Exponent

Mantissa: 1.0 <= v < 10.0, with exactly one digit to the left of the decimal point.

(Note: the first digit must be between 1 and 9)

We could use the same idea in binary

-6.84 is written as -1.71×2^2

0.05 is written as 1.6×2^{-5}

Note: The integer part (first digit) of the mantissa is always 1. Why?

Enter a new standard …

There was no floating point standard 40 years ago!

Which meant people were locked into work with particular computers and software was not portable

In the 80s, the IEEE produced a standard that most manufacturers now follow

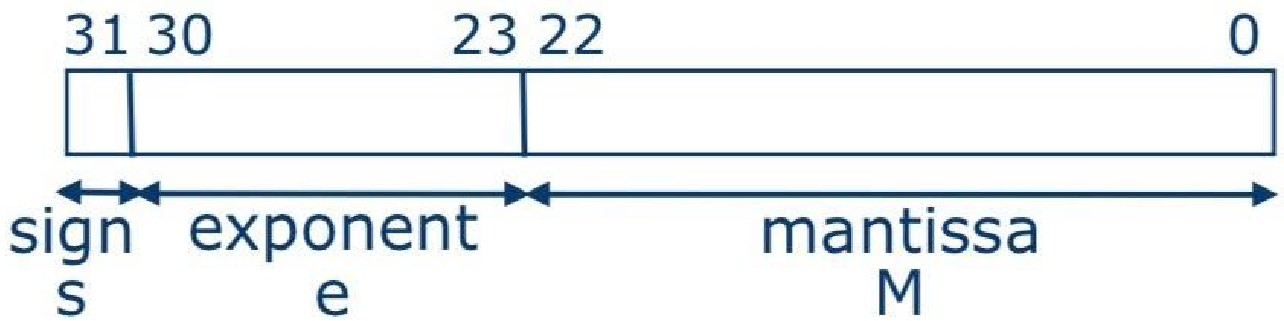
William Kahan spearheaded the effort and won a Turing in 1989 for the work. He's a U of T alum!

The IEEE 754 Format

We can express numbers in 32 bit binary scientific notation as follows:

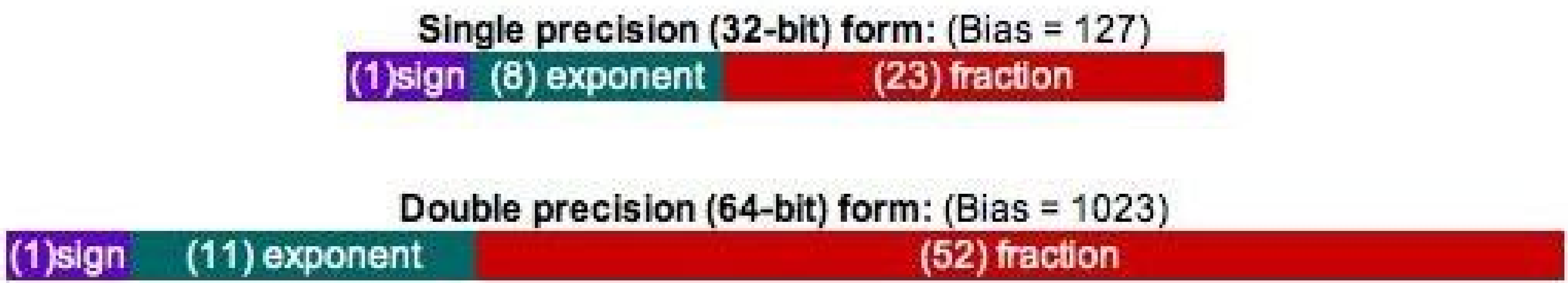
$$(-1)^s * (1 + M) * 2^{e-127}$$

- **1 bit** for the **sign**: 1 for negative and 0 for positive
- **8 bits** for the **exponent** e
  - To allow for negative exponents, 127 is added to that exponent to get the representation. We say that the exponent is "biased" by 127.
  - So the range of possible exponents is not 0 to 2^8-1 = 0 to 255, but (0-127) to (255-127) = -127 to 128.
- **23 bits** for the **mantissa** M
  - Since the first bit must be 1, we don't waste space storing it!



Single vs. Double Precision

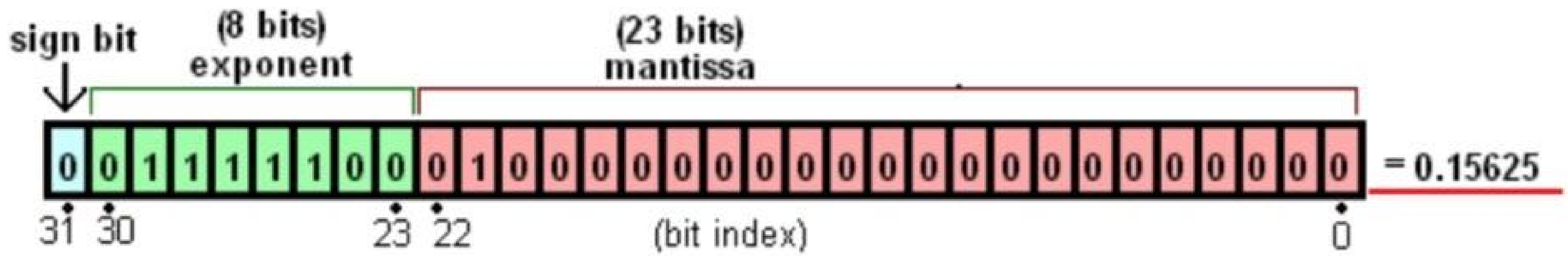
In Java, this data type is called **float**.



In Java, this data type is called **double**.

Half precision is 16 bits! With 5 user for the exponent, 1 for the sign and 10 for the mantissa.

An example



$$(-1)^{\text{sign}}\left(1+\sum_{i=1}^{23} b_{23-i}2^{-i}\right)\times 2^{(e-127)}$$

- $\text{sign} = 0$
- $1+\sum_{i=1}^{23} b_{23-i}2^{-i} = 1+2^{-2} = 1.25$
- $2^{(e-127)} = 2^{124-127} = 2^{-3}$

thus:

- $\text{value} = 1.25 \times 2^{-3} = 0.15625$

How to convert 0.085 to binary?

- **Write 0.085 in base-2 scientific notation**
  - 0.085 = 1.36 / 16 = 1.36 x 2^(-4)
- Determine the **sign bit**
  - it's 0 because positive number
- Determine the **exponent**
  - -4 = 123 - 127, so it's 8-bit binary for 123: 01111011
- Determine the **mantissa**
  - convert 0.36 to binary by repeatedly multiplying by 2
  - keep 23 bits, we get 0.01011100001010001111011

```
0.36 x 2 = 0.72
0.72 x 2 = 1.44
0.44 x 2 = 0.88
0.88 x 2 = 1.76
0.76 x 2 = 1.52
0.52 x 2 = 1.04
0.04 x 2 = 0.08
0.08 x 2 = 0.16
0.16 x 2 = 0.32
0.32 x 2 = 0.64
0.64 x 2 = 1.28
0.28 x 2 = 0.56
0.56 x 2 = 1.12
0.12 x 2 = 0.24
0.24 x 2 = 0.48
0.48 x 2 = 0.96
0.96 x 2 = 1.92
0.92 x 2 = 1.84
0.84 x 2 = 1.68
0.68 x 2 = 1.36
0.36 x 2 =  ...
```

Finally, the result is: 0[01111011]01011100001010001111011

Rounding

- If we have to lose some digits, we don't just truncate, we round.
- In rounding a decimal to a whole number, an issue arises: If we have a 0.5, do we round up or down?
- If we always round up, we are biasing towards higher values.
- "Proper" rounding: round to the nearest even number.
  E.g., 17.5 is rounded up to 18 but 16.5 is rounded down to 16.
- The IEEE standard uses proper rounding also.

Rounding to Even

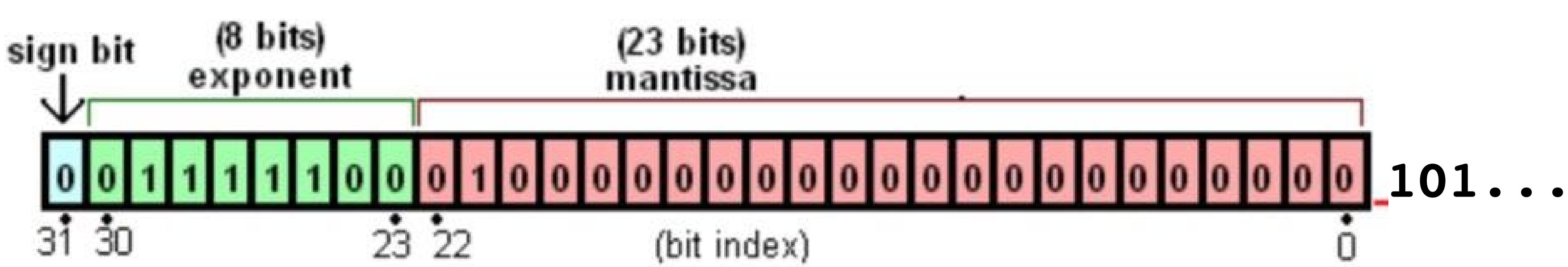
At the **23rd** bit, round to the nearest **even**.

When rounding, take a look at what the **3 bits following the 23rd bit** would have been.

Cases to consider:

- If the 24th bit would be a **0**, round down (i.e. do nothing)
- If the 24th bit would be a **1 followed by 10, 01, or 11** round up (i.e. add 1 to the mantissa's least significant bit)
- If the 24th bit would be a **1 followed by 00**:
  - If the 23rd bit is a **1**, round up
  - If the 23rd bit is a **0**, round down (i.e. do nothing)

Means, if the mantissa is odd we round up, else we round down.
We're splitting the difference!



Rounding Example

Consider rounding 0.1

$0.1 = 1.6 * 2^{-4}$

23

0[01111011] 1001 1001 1001 1001 1001 10**0****1** **10**01

Rounding Example

Consider rounding 0.1

$0.1 = 1.6 * 2^{-4}$

23

0[01111011] 1001 1001 1001 1001 1001 10**01** **10**01

after rounding (case round-up)

0[01111011] 1001 1001 1001 1001 1001 101

Rounding Example

Consider rounding 0.1

$0.3 = 1.2 * 2^{-2}$

23

`0[01111101] 0011 0011 0011 0011 0011 0011 0011`

Rounding Example

Consider rounding 0.1

$0.3 = 1.2 * 2^{-2}$

23

0[01111101] 0011 0011 0011 0011 0011 0011 0011

after rounding (case round-up)

0[01111101] 0011 0011 0011 0011 0011 010

Rounding Example

Consider rounding 0.1

$4.2 = 1.05 * 2^2$

23

`0[10000001] 0000 1100 1100 1100 1100 1100 11…`

Rounding Example

Consider rounding 0.1

$4.2 = 1.05 * 2^2$

23

```
0[10000001] 0000 1100 1100 1100 1100 1100 11…
```

after rounding (case round-down)

```
0[10000001] 0000 1100 1100 1100 1100 110
```

Special Values

Some Special Values

```
0[00000000]00000000000000000000000: Zero

0[11111111]00000000000000000000000: +infinity

1[11111111]00000000000000000000000: -infinity

*[11111111]-anything-but-all-zero-: Not a Number (NaN)
```

Overflow

O**verflow** is the largest representable number

For IEEE-754 single precision, it is

0[11111110]11111111111111111111111

= +1.11111111111111111111111(binary) x 2^(127)

= overflow

Underflow

Underflow is the smallest positive representable number

Example: For single precision, it appears that underflow is

```
          0[00000001]00000000000000000000000
    = 1.00000000000000000000000(binary) x 2^(-126)
```

But not really, as IEEE-754 allows **denormalized** numbers (numbers that do not follow **scientific notation**)! So the real underflow is:

```
          0[00000000]00000000000000000000001
    = 0.00000000000000000000001(binary) x 2^(-126)
         = 2^(-23) x 2^(-126) = 1 x 2^(-149)
```

Denormalized examples (in blue)

0[00000011]00000000000000000000000=
+1.00000000000000000000000x2^(-124)=4.7019774E-38

0[00000010]00000000000000000000000=
+1.00000000000000000000000x2^(-125)=2.3509887E-38

0[00000001]00000000000000000000000=
+1.00000000000000000000000x2^(-126)=1.17549435E-38

0[00000000]10000000000000000000000=
+0.10000000000000000000000x2^(-126)=5.877472E-39

0[00000000]01000000000000000000000=
+0.01000000000000000000000x2^(-126)=2.938736E-39

0[00000000]00100000000000000000000=
+0.00100000000000000000000x2^(-126)=1.469368E-39

Machine Epsilon

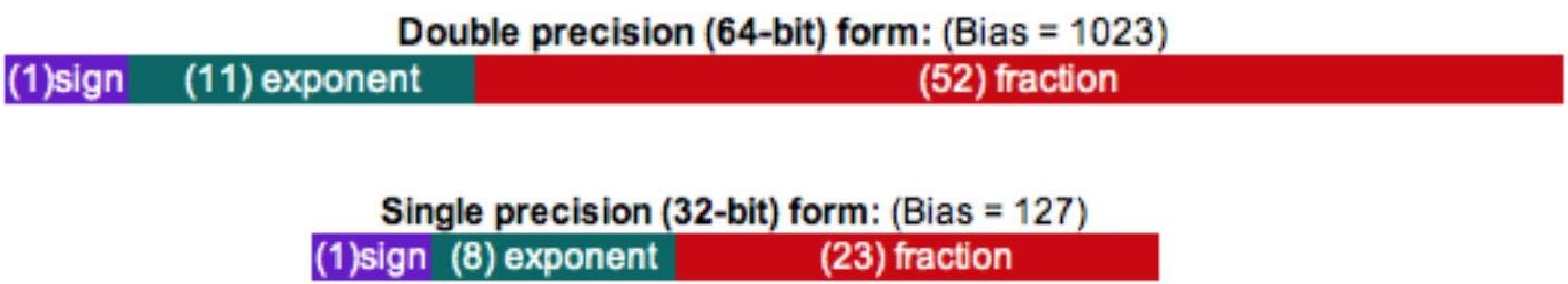
Machine Epsilon (eps) is a number such that 1 + eps is the smallest mantissa that you can get that is > 1

Machine Epsilon is the best precision you can have in the mantissa.

For single precision: eps = $1 \times 2^{-23} \approx$ 1.19e-7

If you add 1.0 by 1e-7, nothing will change.

For double precision, eps = $1 \times 2^{-52} \approx$ 2.22e-16



Arithmetic Operations

1.23450x10^12 + 1.50000x10^10 =

$1.23450\text{x}10^{12} + 0.01500\text{x}10^{12} = 1.24950\text{x}10^{12}$
(We convert to the **same exponent** and add/subtract)

$1.23450\text{x}10^{12} - 1.23400\text{x}10^{12} = 0.00050\text{x}10^{12} = 5.00000\text{x}10^{8}$
(We **normalize** the result if necessary)

Demo

```
addingDemo(); //different results via different ways of adding
totallingDemo(); //different ways of accumulating values
```

What did we learn?

0.1 can’t be represented exactly so we use an approximated value (rounded up) that may lead to an unexpected result.

Addition a very small quantity to a large quantity may mean the small quantity falls off the end of the mantissa

When we add two small quantities together, this doesn’t happen. And if we then accumulate the sum into a large quantity the small values may not be lost when we finally add the big quantity to the total.

Demo

```java
//another demonstration of different ways of accumulating values
ArrayTotal total = new ArrayTotal(1000000);
double v1 = total.sum1();
double v2 = total.sum2();
```

What did we learn?

When adding floating point numbers, add the smallest ones first

Try avoiding additions of dissimilar quantities

When addition a list of floating points, sort them first!

Demo

```
loopingCountDemo(); //demonstration of looping conditions
```

What did we learn?

Don’t use floating point variables to control a counted loop

Use less arithmetic operations where possible, as fewer operations means less error is accumulated

Avoid checking equality between numbers with ==, i.e. instead of this:

x == 0.207

Write one of these:

(x >= 0.207-0.0001) && (x <= 0.207+0.0001)

abs(x - 0.207) <= 0.0001

Demo

```
examineDemo(); //Floating point numbers and rounding
```

UNIVERSITY OF TORONTO MISSISSAUGA

What did we learn?

4/5 = 1.10011001 10011001 10011001 1001100... x 2^(-1)

When rounded, this is

1.10011001 10011001 1001101(binary) x 2^(-1)

When we print it gets converted back to decimal which is:

0.80000001192092895507812500000000

However, the best precision you have here is just

2^{-23}*2^{-1} ≈ 6e-8

And only the 7 blue digits are significant

Don’t print more precision in your output than you are holding.

Why does it matter?

UNIVERSITY OF TORONTO MISSISSAUGA

Patriot missile accident

- In 1991, an American missile failed to track and destroy an incoming missile. Instead it hit a US Army barracks, killing 28.
- The system tracked time in tenths of seconds. The error in approximating 0.1 with 24 bits was magnified in its calculations.
- At the time of the accident, the error corresponded to 0.34 seconds. A Patriot missile travels about half a km in that time.

http://www-users.math.umn.edu/~arnold/disasters/patriot.html

Ariane 5 rocket explosion

- In 1996, the European Space Agency's Ariane 5 rocket exploded 40 seconds after launch.
- During conversion of a 64-bit to a 16-bit format, overflow occurred: the number was too big to store in 16 bits.
- This hadn't been expected because the data (acceleration reported by sensors) had never been this large before. But this new rocket was faster than its predecessor.
- $7 billion of R&D had been invested in this rocket.

https://around.com/ariane.html

Sinking of an oil rig

- In 1992, the Sleipner A oil and gas platform sank in the North Sea near Norway.
- Numerical issues in modelling the structure caused shear stresses to be underestimated by 47%.
- As a result, concrete walls were not built thick enough.
- Cost: $700 million

http://www-users.math.umn.edu/~arnold/disasters/sleipner.html

If you like this stuff ….

Take CSC336! Or any other numerical methods class.

“95% of folks out there are completely clueless about floating point”
James Gosling, designer of Java

References

http://www.oxfordmathcenter.com/drupal7/node/43

https://www.youtube.com/watch?v=PZRI1IfStY0

https://www.cs.umd.edu/class/sum2003/cmsc311/Notes/Data/float.html