Computer arithmetics: integers, binary floating-point, and decimal floating-point

Peter Sestoft
BSWU First-Year Project
Thursday 20 February 2020
Computer arithmetics

• Computer numbers are cleverly designed, but
  – Very different from high-school mathematics
  – There are some surprises

• Choose representation with care:
  – When to use int, short, long, byte, ...
  – When to use double or float
  – When to use decimal floating-point
Overview, number representations

• Integers
  – **Unsigned**, binary, hexadecimal
  – Signed
    • Signed-magnitude
    • Two’s complement (Java and C# int, short, byte, ...)
  – Arithmetic modulo $2^n$

• Floating-point numbers
  – IEEE 754 binary32 and binary64
    • Which you know as `float` and `double` in Java and C#
  – IEEE 754 decimal128
    • and also C#'s `decimal` type
    • and also Java’s `java.math.BigDecimal`
Unsigned integers, binary representation

• Decimal notation
  \[805_{10} = 8 \times 10^2 + 0 \times 10^1 + 5 \times 10^0 = 805\]
  A place is worth 10 times that to the right

• Binary notation
  \[1101_2 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13\]
  A place is worth 2 times that to the right

• Positional number systems:
  – Base is 10 or 2 or 16 or ...

• Any non-positional number systems?

<table>
<thead>
<tr>
<th>Base</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2^0)</td>
<td>1</td>
</tr>
<tr>
<td>(2^1)</td>
<td>2</td>
</tr>
<tr>
<td>(2^2)</td>
<td>4</td>
</tr>
<tr>
<td>(2^3)</td>
<td>8</td>
</tr>
<tr>
<td>(2^4)</td>
<td>16</td>
</tr>
<tr>
<td>(2^5)</td>
<td>32</td>
</tr>
<tr>
<td>(2^6)</td>
<td>64</td>
</tr>
<tr>
<td>(2^7)</td>
<td>128</td>
</tr>
<tr>
<td>(2^8)</td>
<td>256</td>
</tr>
</tbody>
</table>
Binary numbers

• A bit is a binary digit: 0 or 1
• Easy to represent in electronics
  – Some base-10 hardware in the 1960es
  – A Russian base-3 computer in the 1950es
• Counting with three bits:
  000, 001, 010, 011, 100, 101, 110, 111
• Computing:
  \[ 1 + 1 = 10 \]
  \[ 010 + 011 = 101 \]
  “There are 10 kinds of people: those who understand binary and those who don’t”
Hexadecimal numbers

- Hexadecimal numbers have base 16
- Digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F
  \[325_{16} = 3 \times 16^2 + 2 \times 16^1 + 5 \times 16^0 = 805\]
  Each place is worth 16 times that ...

- Useful alternative to binary
  - Because \(16 = 2^4\)
  - So 1 hex digit = 4 binary digits (bits)

- Computing in hex:
  A + B = 15
  AA + 1 = AB
  AA + 10 = BA
Overview, number representations

• Integers
  – Unsigned, binary, hexadecimal
  – **Signed**
    • Signed-magnitude
    • Two’s complement (Java and C# `long, int, short, ...`)
  – Arithmetic modulo $2^n$

• Floating-point numbers
  – IEEE 754 binary32 and binary64
    • Which you know as `float` and `double` in Java and C#
  – IEEE 754 decimal128
    • and also C#'s `decimal` type
    • and also Java’s `java.math.BigDecimal`
Signed integers: negative and positive

• Signed magnitude: A sign bit and a number
  – Problem: Then we have both +0 and -0

• Two’s complement: Negate all bits, add 1
  - Only one zero
  - Easy to compute with
  - Requires known size of number, e.g. 4, 8, 16, 32, 64 bits

• Examples of two’s complement, using 4 bits:
  -3 is represented by 1101 because $3 = 0011_2$ so complement is 1100; add 1 to get $-3 = 1101_2$

  -1 is represented by 1111 because $1 = 0001_2$ so complement is 1110; add 1 to get $-1 = 1111_2$

  -8 is represented by 1000 because $8 = 1000_2$ so complement is 0111; add 1 to get $-8 = 1000_2$
Check: Two's complement

• What decimal number does the 8-bit two's complement number $10001001_2$ represent?

A) $137 = 2^7 + 2^3 + 2^0$
B) $-118 = -(2^6 + 2^5 + 2^4 + 2^2 + 2^1)$
C) $119 = 2^6 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0$
D) $-119 = -(2^6 + 2^5 + 2^4 + 2^2 + 2^1 + 2^0)$
**Integer arithmetics modulo $2^n$**

- Java and C# `int` is 32-bit two’s complement
  - Max int is $2^{31}-1 = 2147483647$
  - Min int is $-(2^{31}) = -2147483648$
  - If $x = 2147483647$ then $x+1 = -2147483648 < x$
  - If $n = -2147483648$ then $-n = n$

<table>
<thead>
<tr>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000000000000000000000000000 = 0</td>
<td></td>
</tr>
<tr>
<td>00000000000000000000000000000001 = 1</td>
<td></td>
</tr>
<tr>
<td>00000000000000000000000000000010 = 2</td>
<td></td>
</tr>
<tr>
<td>00000000000000000000000000000011 = 3</td>
<td></td>
</tr>
<tr>
<td>01111111111111111111111111111111 = 2147483647</td>
<td></td>
</tr>
<tr>
<td>11111111111111111111111111111111 = -1</td>
<td></td>
</tr>
<tr>
<td>11111111111111111111111111111110 = -2</td>
<td></td>
</tr>
<tr>
<td>11111111111111111111111111111101 = -3</td>
<td></td>
</tr>
<tr>
<td>10000000000000000000000000000000 = -2147483648</td>
<td></td>
</tr>
</tbody>
</table>
An obviously non-terminating loop?

```java
int i = 1;
while (i > 0)
    i++;
System.out.println(i);

Does terminate!
```

Values of i:

```
1
2
3
...
2147483646
2147483647
-2147483648
```
Overview, number representations

• Integers
  – Unsigned, binary, hexadecimal
  – Signed
    • Signed-magnitude
    • Two’s complement (Java and C# int, short, byte, ...)
  – Arithmetic modulo $2^n$

• Floating-point numbers
  – IEEE 754 binary32 and binary64
    • Which you know as float and double in Java and C#
  – IEEE 754 decimal128
    • and also C#’s decimal type
    • and also Java’s java.math.BigDecimal
Binary fractions

• Before the point: ..., 16, 8, 4, 2, 1
• After the point: 1/2, 1/4, 1/8, 1/16, ...

0.5 = 0.1 \_2

0.25 = 0.01 \_2

0.75 = 0.11 \_2

0.125 = 0.001 \_2

2.125 = 10.001 \_2

7.625 = 111.101 \_2

118.625 = 1110110.101 \_2

• But
  – how many digits are needed before the point?
  – how many digits are needed after the point?

• Answer: Binary floating-point (double, float)
  – The point is placed dynamically
Check: Binary fractions

• Binary $1110110.101_2$ represents 118.625
• What does $11101101.01_2$ represent?
  A) $237.250 = 118.625 \times 2$
  B) $59.3125 = 118.625 \div 2$
  C) $1186.25 = 118.625 \times 10$
  D) $11.8625 = 118.625 \div 10$
Check: Binary fractions

• What is 27.375 represented as a binary fraction?
  A) 11010.011₂
  B) 11011.011₂
  C) 11011.101₂
  D) 11011.001₂
Some nasty fractions

• Some numbers are not representable as finite decimal fractions:
  \[ \frac{1}{7} = 0.142857142857142857\ldots_{10} \]

• Same problem with binary fractions:
  \[ \frac{1}{10} = 0.00011001100110011001100\ldots_{2} \]

• Quite unfortunate:
  – Float 0.10 is 0.100000001490116119384765625
  – So cannot represent 0.10 krone or $0.10 exactly
  – Nor 0.01 krone or $0.01 exactly

• **Do not** use binary floating-point (float, double) for accounting!
An obviously terminating loop?

double d = 0.0;
while (d != 1.0)
    d += 0.1;

Values of d:
0.10000000000000000000
0.20000000000000000000
0.30000000000000000000
0.40000000000000000000
0.50000000000000000000
0.60000000000000000000
0.70000000000000000000
0.79999999999999990000
0.89999999999999990000
0.99999999999999990000
1.09999999999999990000
1.20000000000000000000
1.30000000000000000000

\(d\) never equals 1.0
Overview, number representations

• Integers
  – Unsigned, binary, hexadecimal
  – Signed
    • Signed-magnitude
    • Two’s complement (Java and C# int, short, byte, …)
  – Arithmetic modulo $2^n$

• Floating-point numbers
  – **IEEE 754 binary32 and binary64**
    • Which you know as float and double in Java and C#
  – IEEE 754 decimal128
    • and also C#’s decimal type
    • and also Java’s java.math.BigDecimal
History of floating-point numbers

• Until 1985: Many different designs, anarchy
  – Difficult to write portable (numerical) software

• Standard IEEE 754-1985 binary fp
  – Implemented by all modern hardware
  – Assumed by modern programming languages
  – Designed primarily by William Kahan for Intel

• Revised standard IEEE 754-2008
  – Binary floating-point, much as in IEEE 754-1985
  – Decimal floating-point, new

• IEEE = “Eye-triple-E” = Institute of Electrical and Electronics Engineers (USA)
IEEE floating point representation

- **Signed-magnitude**
  - Sign, exponent, significand: number = \( s \times 2^{e-b} \times (f + 1) \)

- **Representation:**
  - Sign \( s \) (0=positive, 1=negative), exponent \( e \), fraction \( f \)

\[
\begin{array}{cccccc}
0 & 01111111 & 00000000000000000000000000000000 &= 1.0 \\
\end{array}
\]

<table>
<thead>
<tr>
<th>Java, C#</th>
<th>bits</th>
<th>e bits</th>
<th>f bits</th>
<th>range</th>
<th>bias b</th>
<th>sign. digits</th>
</tr>
</thead>
<tbody>
<tr>
<td>float, binary32</td>
<td>32</td>
<td>8</td>
<td>23</td>
<td>( \pm 10^{-44} ) to ( \pm 10^{38} )</td>
<td>127</td>
<td>7</td>
</tr>
<tr>
<td>double, binary64</td>
<td>64</td>
<td>11</td>
<td>52</td>
<td>( \pm 10^{-323} ) to ( \pm 10^{308} )</td>
<td>1023</td>
<td>15</td>
</tr>
<tr>
<td>Intel extended</td>
<td>80</td>
<td>15</td>
<td>64</td>
<td>( \pm 10^{-4932} ) to ( \pm 10^{4932} )</td>
<td>16635</td>
<td>19</td>
</tr>
</tbody>
</table>
Understanding the representation

• *Normalized* numbers
  – Choose exponent e so the significand is 1.ffffffff...
  – Hence we need only store the .fffffff... not the 1.

• Exponent is unsigned but a bias is subtracted
  – For 32-bit float the bias b is 127

<table>
<thead>
<tr>
<th>s</th>
<th>eeeeeeee</th>
<th>fffffffffffffffffffffffffffffffff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>00000000000000000000000000000000 = 0.0</td>
</tr>
<tr>
<td>1</td>
<td>00000000</td>
<td>00000000000000000000000000000000 = -0.0</td>
</tr>
<tr>
<td>0</td>
<td>01111111</td>
<td>00000000000000000000000000000000 = 1.0</td>
</tr>
<tr>
<td>0</td>
<td>01111110</td>
<td>00000000000000000000000000000000 = 0.5</td>
</tr>
<tr>
<td>1</td>
<td>10000101</td>
<td>11011010101000000000000000000000 = -118.625</td>
</tr>
<tr>
<td>0</td>
<td>01111011</td>
<td>1001100110011001100110011001 = 0.1</td>
</tr>
<tr>
<td>0</td>
<td>01111111</td>
<td>000000000000000000000000000000001 = 1.0000001</td>
</tr>
</tbody>
</table>
A detailed example

- Consider $x = -118.625$
- We know that $118.625 = 1110110.101_2$
- Normalize to $2^6 * 1.110110101_2$
- So
  - exponent is 6, represented by $e = 6+127 = 133$
  - significand is $1.110110101_2$
  - so fraction $f = .110110101_2$
  - sign is 1 for negative

```
s eeeeee ffffffffffffffffffffffffffffffffff
1 10000101 11011010100000000000000000000000 = -118.62522
```
The normalized number line

- Representable with 2 f bits and 2 e bits:
  (If bias = 1, then exponent is -1, 0, 1, or 2)
  
- Same relative precision for all numbers
- Lower absolute precision for large numbers
Units in the last place (ulp)

• The distance between two neighbor numbers is called 1 ulp = unit in the last place

s eeeeeee ffffffffffffffffffffffffff
0 01111111 00000000000000000000000 = 1.0
0 01111111 00000000000000000000001 = 1.0000001

• A good measure of
  – relative representation error
  – relative computation error

• Eg java.lang.Math.log documentation says
  "The computed result must be within 1 ulp of the exact result."
Check: floating-points numbers

- If 27.375 is represented as the binary fraction $11011.011_2$, what is the 32-bit floating point representation of -27.375?

<table>
<thead>
<tr>
<th></th>
<th>s</th>
<th>eeeeeeeeee</th>
<th>fffffffffffffffffffffffffffffffff</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>000000100_2 = 4</td>
<td>10110110000000000000000000000000</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>100000011_2 = 131</td>
<td>10110110000000000000000000000000</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>100000101_2 = 5</td>
<td>10110110000000000000000000000000</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>100000010_2 = 130</td>
<td>10110110000000000000000000000000</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>100000010_2 = 2</td>
<td>10110110000000000000000000000000</td>
</tr>
</tbody>
</table>
Special “numbers”

- Denormal (and zero) numbers, resulting from underflow
- Infinite numbers, resulting from 1.0/0.0, Math.log(0), ...
- NaNs (not-a-number), resulting from 0.0/0.0, Math.sqrt(-1), ...

<table>
<thead>
<tr>
<th>Exponent e-b</th>
<th>Represented number</th>
</tr>
</thead>
<tbody>
<tr>
<td>-126...127</td>
<td>Normal: ±10^{-38} to ±10^{38}</td>
</tr>
<tr>
<td>-127</td>
<td>Denormal, or zero: ±10^{-44} to ±10^{-38}, and ±0.0</td>
</tr>
<tr>
<td>128</td>
<td>Infinities, when f=0...0</td>
</tr>
<tr>
<td>128</td>
<td>NaNs, when f=1xx...xx</td>
</tr>
</tbody>
</table>

s eeeeeeee fFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
1 10000101 11011010100000000000000000 = -118.625
0 00000000 0001000000000000000000000 = 7.346E-40
0 11111111 0000000000000000000000000 = +Infinity
1 11111111 0000000000000000000000000 = -Infinity
s 11111111 1000000000000000000000000 = NaN
Why denormal numbers?

• To allow gradual underflow, small numbers
• To ensure that \( x-y == 0 \) if and only if \( x == y \)
• Example denormal result in float:
  - Smallest non-zero *normal* number is \( 2^{-126} \)
  - So choose \( x = 1.01_2 \times 2^{-126} \) and \( y = 1.00_2 \times 2^{-126} \):

  ```
  s   eeeeee   fffffffffffffffffffffffffffffffffffffffff
  0 00000001 01000000000000000000000000000000 = x
  0 00000001 00000000000000000000000000000000 = y
  0 00000000 01000000000000000000000000000000 = x-y
  ```

• What would happen without denormal?
  - Since \( x-y \) is \( 2^{-128} \) it is less than \( 2^{-126} \)
  - So result of \( x-y \) would be represented as 0.0
  - But clearly \( x != y \), so this would be confusing
Why infinities?

• 1: A simple solution to overflow
  - Math.exp(100000.0) gives +Infinity

• 2: To make “sensible” expressions work
  - Example: Compute f(x) = x/(x^2+1.0)
  - But if x is large then x^2 may overflow
  - Better compute: f(x) = 1.0/(x+1.0/x)
  - But if x=0 then 1.0/x looks bad, yet want f(0)=0

Solution:
  - Let 1.0 / 0.0 be Infinity
  - Let 0.0 + Infinity be Infinity
  - Let 1.0 / Infinity be 0.0
  - Then 1.0/(0.0+1.0/0.0) gives 0 as should for x=0
Why NaNs?

• An efficient way to report and propagate error
  – Languages like C do not have exceptions
  – Exceptions are 10,000 times slower than (1.2+x)

• Even weird expressions must have a result
  0.0/0.0 gives NaN
  Infinity − Infinity gives NaN
  Math.sqrt(-1.0) gives NaN
  Math.log(-1.0) gives NaN

• Operations must preserve NaNs
  NaN + 17.0 gives NaN
  Math.sqrt(NaN) gives NaN
  ... and so on
What about double (binary64)?

- The same, just with 64=1+11+52 bits instead of 32

<table>
<thead>
<tr>
<th>s</th>
<th>eeeeeeeeeee</th>
<th>fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000000</td>
<td>0000000000000000000000000000000000000000000000000000000000000000 = +0.0</td>
</tr>
<tr>
<td>1</td>
<td>00000000000</td>
<td>0000000000000000000000000000000000000000000000000000000000000000 = -0.0</td>
</tr>
<tr>
<td>0</td>
<td>01111111111</td>
<td>0000000000000000000000000000000000000000000000000000000000000000 = 1.0</td>
</tr>
<tr>
<td>0</td>
<td>01111111110</td>
<td>0000000000000000000000000000000000000000000000000000000000000000 = 0.5</td>
</tr>
<tr>
<td>1</td>
<td>10000000101</td>
<td>1101101010000000000000000000000000000000000000000000000000000000 = -118.625</td>
</tr>
<tr>
<td>0</td>
<td>11111111111</td>
<td>0000000000000000000000000000000000000000000000000000000000000000 = +Infinity</td>
</tr>
<tr>
<td>1</td>
<td>11111111111</td>
<td>0000000000000000000000000000000000000000000000000000000000000000 = -Infinity</td>
</tr>
<tr>
<td>s</td>
<td>11111111111</td>
<td>1000000000000000000000000000000000000000000000000000000000000000 = NaN</td>
</tr>
<tr>
<td>0</td>
<td>00000000000</td>
<td>0001000000000000000000000000000000000000000000000000000000000000 = 1.39E-309</td>
</tr>
<tr>
<td>0</td>
<td>01111111011</td>
<td>1001100110011001100110011001100110011001100110011001100110011010 = 0.1</td>
</tr>
<tr>
<td>0</td>
<td>01111111110</td>
<td>1111111111111111111111111111111111111111111111111111111111111111 = 0.999...9</td>
</tr>
</tbody>
</table>

0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1, clearly not equal to 1.0

- Double 0.1 is really this exact number:

0.100000000000000055511151231257827021181583404541015625
## IEEE addition

<table>
<thead>
<tr>
<th>+</th>
<th>-Inf</th>
<th>-2.0</th>
<th>-0.0</th>
<th>0.0</th>
<th>2.0</th>
<th>+Inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Inf</td>
<td>-Inf</td>
<td>-Inf</td>
<td>-Inf</td>
<td>-Inf</td>
<td>-Inf</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>-2.0</td>
<td>-Inf</td>
<td>-4.0</td>
<td>-2.0</td>
<td>-2.0</td>
<td>0.0</td>
<td>+Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>-0.0</td>
<td>-Inf</td>
<td>-2.0</td>
<td>-0.0</td>
<td>0.0</td>
<td>2.0</td>
<td>+Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>0.0</td>
<td>-Inf</td>
<td>-2.0</td>
<td>0.0</td>
<td>0.0</td>
<td>2.0</td>
<td>+Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>2.0</td>
<td>-Inf</td>
<td>0.0</td>
<td>2.0</td>
<td>2.0</td>
<td>4.0</td>
<td>+Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>+Inf</td>
<td>NaN</td>
<td>+Inf</td>
<td>+Inf</td>
<td>+Inf</td>
<td>+Inf</td>
<td>+Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
### IEEE subtraction

<table>
<thead>
<tr>
<th>-</th>
<th>-Inf</th>
<th>-2.0</th>
<th>-0.0</th>
<th>0.0</th>
<th>2.0</th>
<th>+Inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Inf</td>
<td>NaN</td>
<td>-Inf</td>
<td>-Inf</td>
<td>-Inf</td>
<td>-Inf</td>
<td>-Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>-2.0</td>
<td>+Inf</td>
<td>0.0</td>
<td>-2.0</td>
<td>-2.0</td>
<td>-4.0</td>
<td>-Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>-0.0</td>
<td>+Inf</td>
<td>2.0</td>
<td>0.0</td>
<td>-0.0</td>
<td>-2.0</td>
<td>-Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>0.0</td>
<td>+Inf</td>
<td>2.0</td>
<td>0.0</td>
<td>0.0</td>
<td>-2.0</td>
<td>-Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>2.0</td>
<td>+Inf</td>
<td>4.0</td>
<td>2.0</td>
<td>2.0</td>
<td>0.0</td>
<td>-Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>+Inf</td>
<td>+Inf</td>
<td>+Inf</td>
<td>+Inf</td>
<td>+Inf</td>
<td>+Inf</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
### IEEE multiplication

<table>
<thead>
<tr>
<th>*</th>
<th>-Inf</th>
<th>-2.0</th>
<th>-0.0</th>
<th>0.0</th>
<th>2.0</th>
<th>+Inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Inf</td>
<td>+Inf</td>
<td>+Inf</td>
<td>NaN</td>
<td>NaN</td>
<td>-Inf</td>
<td>-Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>-2.0</td>
<td>+Inf</td>
<td>4.0</td>
<td>0.0</td>
<td>-0.0</td>
<td>-4.0</td>
<td>-Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>-0.0</td>
<td>NaN</td>
<td>0.0</td>
<td>0.0</td>
<td>-0.0</td>
<td>-0.0</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>0.0</td>
<td>NaN</td>
<td>-0.0</td>
<td>-0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>2.0</td>
<td>-Inf</td>
<td>-4.0</td>
<td>-0.0</td>
<td>0.0</td>
<td>4.0</td>
<td>+Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>+Inf</td>
<td>-Inf</td>
<td>-Inf</td>
<td>NaN</td>
<td>NaN</td>
<td>+Inf</td>
<td>+Inf</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
## IEEE division

<table>
<thead>
<tr>
<th></th>
<th>-Inf</th>
<th>-2.0</th>
<th>-0.0</th>
<th>0.0</th>
<th>2.0</th>
<th>+Inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Inf</td>
<td>NaN</td>
<td>+Inf</td>
<td>+Inf</td>
<td>-Inf</td>
<td>-Inf</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>-2.0</td>
<td>0.0</td>
<td>1.0</td>
<td>+Inf</td>
<td>-Inf</td>
<td>-1.0</td>
<td>-0.0</td>
<td>NaN</td>
</tr>
<tr>
<td>-0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>NaN</td>
<td>NaN</td>
<td>-0.0</td>
<td>-0.0</td>
<td>NaN</td>
</tr>
<tr>
<td>0.0</td>
<td>-0.0</td>
<td>-0.0</td>
<td>NaN</td>
<td>NaN</td>
<td>0.0</td>
<td>0.0</td>
<td>NaN</td>
</tr>
<tr>
<td>2.0</td>
<td>-0.0</td>
<td>-1.0</td>
<td>-Inf</td>
<td>+Inf</td>
<td>1.0</td>
<td>0.0</td>
<td>NaN</td>
</tr>
<tr>
<td>+Inf</td>
<td>NaN</td>
<td>-Inf</td>
<td>-Inf</td>
<td>+Inf</td>
<td>+Inf</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
<td>NaN</td>
</tr>
</tbody>
</table>
IEEE equality and ordering

<table>
<thead>
<tr>
<th>==</th>
<th>-Inf</th>
<th>-2.0</th>
<th>-0.0</th>
<th>0.0</th>
<th>2.0</th>
<th>+Inf</th>
<th>NaN</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Inf</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>-2.0</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>-0.0</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>0.0</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>2.0</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>+Inf</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>NaN</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>

- Equality (==, !=)
  - A NaN is not equal to anything, not even itself
  - So if \( y \) is NaN, then \( y \neq y \), and vice versa

- Ordering: \( -\infty < -2.0 < -0.0 == 0.0 < 2.0 < +\infty \)
  - All comparisons involving NaNs give false
Java and C# mathematical functions

- In general, functions behave sensibly
  - Give +Infinity or -Infinity on extreme arguments
  - Give NaN on invalid arguments
  - Preserve NaN arguments, with few exceptions

<table>
<thead>
<tr>
<th>Function</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt(-2.0)</td>
<td>NaN</td>
</tr>
<tr>
<td>sqrt(NaN)</td>
<td>NaN</td>
</tr>
<tr>
<td>log(0.0)</td>
<td>-Inf</td>
</tr>
<tr>
<td>log(NaN)</td>
<td>NaN</td>
</tr>
<tr>
<td>log(-1.0)</td>
<td>NaN</td>
</tr>
<tr>
<td>sin(Inf)</td>
<td>NaN</td>
</tr>
<tr>
<td>sin(NaN)</td>
<td>NaN</td>
</tr>
<tr>
<td>asin(2.0)</td>
<td>NaN</td>
</tr>
<tr>
<td>exp(10000.0)</td>
<td>Inf</td>
</tr>
<tr>
<td>exp(NaN)</td>
<td>NaN</td>
</tr>
<tr>
<td>exp(-Inf)</td>
<td>0.0</td>
</tr>
<tr>
<td>pow(0.0, -1.0)</td>
<td>Inf</td>
</tr>
<tr>
<td>pow(NaN, 0.0)</td>
<td>1 in Java</td>
</tr>
</tbody>
</table>
Rounding modes

• High-school: round 0.5 upwards
  – Rounds 0,1,2,3,4 down and rounds 5,6,7,8,9 up
• Looks fair
• But dangerous: may introduce drift in loops

• IEEE-754:
  – Rounds 0,1,2,3,4 down and rounds 6,7,8,9 up
  – Rounds 0.5 to nearest even number (or more generally, to zero least significant bit)
• So both 1.5 and 2.5 round to 2.0
Basic principle of IEEE floating-point

“Each of the computational operations ... shall be performed as if it first produced an intermediate result correct to infinite precision and unbounded range, and then rounded that intermediate result to fit in the destination’s format” (IEEE 754-2008 § 5.1)

- So the machine result of $x*y$ is the rounding of the “real” result of $x*y$
- This is simple and easy to reason about
- ... and quite surprising that it can be implemented in finite hardware
Check: IEEE surprises

• For which values of $x$ is $1.0 + x = 1.0$ with 32-bit floating point arithmetic?
  A) $1.0 \times 10^{-6}$
  B) $1.0 \times 10^{8}$
  C) $1.0 \times 10^{-8}$
  D) $1.0 \times 10^{6}$

Hint: $\log_{10}(2^{23}) \approx 6.92$ significant digits
Loss of precision 1 (ex: double)

• Let **double** \( z = 2^{53} \), then \( z + 1.0 \neq z \)
  
  - because only 52 digits in fraction

\[
\begin{align*}
0 \ 10000110100 & \quad 0000000000000000000000000000000000000000000000000000000000000000000 = z \\
0 \ 10000110100 & \quad 0000000000000000000000000000000000000000000000000000000000000000000 = z + 1
\end{align*}
\]
Loss of precision 2 (ex: double) Catastrophic cancellation

• Let \( v=9876543210.2 \) and \( w=9876543210.1 \)
• Big and nearly equal; correct to 16 decimal places
• But their difference \( v-w \) is correct only to 6 places
• Because fractions were correct only to 6 places

\[
\begin{align*}
 v &= 9876543210.200000 \\
w &= 9876543210.100000 \\
v-w &= 0.10000038146972656
\end{align*}
\]

Garbage, why?

The exact actual numbers

Would be non-zero in full-precision 0.1
Case: Solving a quadratic equation

• The solutions to $ax^2 + bx + c = 0$ are

$$x_1 = \frac{-b + \sqrt{d}}{2a}$$

$$x_2 = \frac{-b - \sqrt{d}}{2a}$$

when $d = b^2 - 4ac > 0$.

• But subtraction $-b \pm \sqrt{d}$ may lose precision when $b^2$ is much larger than $4ac$; in this case the square root is nearly $b$.

• Fix: Since $\sqrt{d} \geq 0$, compute $x_1$ first if $b < 0$, else compute $x_2$ first

• Then compute $x_2$ from $x_1$; or $x_1$ from $x_2$
Bad and good quadratic solutions

```java
double d = b * b - 4 * a * c;
if (d > 0) {
    double y = Math.sqrt(d);
    double x1 = (-b - y)/(2 * a);
    double x2 = (-b + y)/(2 * a);
}
```

```java
double d = b * b - 4 * a * c;
if (d > 0) {
    double y = Math.sqrt(d);
    double x1 = b > 0 ? (-b - y)/(2*a) : (-b + y)/(2*a);
    double x2 = c / (x1 * a);
} else ...
```

- When $a=1$, $b=10^9$, $c=1$ we get
  - Bad algorithm: $x_1 = -1.00000e+09$ and $x_2 = 0.00000$
  - Good algorithm: $x_1 = -1.00000e+09$ and $x_2 = -1.00000e-09$
Case: Linear regression

Points (2.1, 5.2), (2.2, 5.4), (2.4, 5.8) have regression line $y = \alpha + \beta x$ with $\alpha = 1$ and $\beta = 2$

\[ y = 2x + 1 \]
Bad way to compute $\alpha$ and $\beta$

```java
double SX = 0.0, SY = 0.0, SSX = 0.0, SXY = 0.0;
for (int i=0; i<n; i++) {
    Point p = ps[i];
    SX += p.x;
    SY += p.y;
    SXY += p.x * p.y;
    SSX += p.x * p.x;
}

double beta = (SXY - SX*SY/n) / (SSX - SX*SX/n);
double alpha = SY/n - SX/n * beta;
```

- This recipe was used for computing by hand
- OK for scattered points near (0,0)
- But otherwise may lose precision because it computes difference between large similar numbers $SSX$ and $SX^2/n$
Better way to compute $\alpha$ and $\beta$

```java
double SX = 0.0, SY = 0.0;
for (int i=0; i<n; i++) {
    Point p = ps[i];
    SX += p.x;
    SY += p.y;
}
double EX = SX/n, EY = SY/n;
double SDXDY = 0.0, SSDX = 0.0;
for (int i=0; i<n; i++) {
    Point p = ps[i];
    double dx = p.x - EX, dy = p.y - EY;
    SDXDY += dx * dy;
    SSDX += dx * dx;
}
double beta = SDXDY/SSDX;
double alpha = SY/n - SX/n * beta;
```

- Mathematically equivalent to previous one, but much more precise on the computer
## Example results

- Consider (2.1, 5.2), (2.2, 5.4), (2.4, 5.8)
- And same with 10 000 000 or 50 000 000 added to each coordinate

<table>
<thead>
<tr>
<th>Move</th>
<th>Bad</th>
<th>Good</th>
<th>Correct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>α</td>
<td>1.000000</td>
<td>1.000000</td>
</tr>
<tr>
<td></td>
<td>β</td>
<td>2.000000</td>
<td>2.000000</td>
</tr>
<tr>
<td><strong>Wrong</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 M</td>
<td>α</td>
<td>3.233333</td>
<td>-9999998.99</td>
</tr>
<tr>
<td></td>
<td>β</td>
<td>1.000000</td>
<td>2.000000</td>
</tr>
<tr>
<td><strong>Very wrong!!</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 M</td>
<td>α</td>
<td>50000005.47</td>
<td>-49999999.27</td>
</tr>
<tr>
<td></td>
<td>β</td>
<td>-0.000000</td>
<td>2.000000</td>
</tr>
</tbody>
</table>
Numerical analysis, neat intro

- Compact, easy to read, by a real expert
- Focus on linear algebra, matrix inversion, condition number, ...

- For misprints, see my review at amazon.com
An accurate computation of sums

- Let `double[] xs = { 1E12, -1, 1E12, -1, ... }`
- The true array sum is 9,999,999,999,990,000.0

```java
double S = 0.0;
for (int i=0; i<xs.length; i++)
    S += xs[i];
```

Naïve sum, error = 992

```java
double S = 0.0, C = 0.0;
for (int i=0; i<xs.length; i++) {
    double Y = xs[i] - C, T = S + Y;
    C = (T - S) - Y;
    S = T;
}
```

Kahan sum, error = 0

Note that C = (T-S)-Y = ((S+Y)-S)-Y may be non-zero
Floating-point tips and tricks

• Do not compare floating-point using ==, !=
  – Use Math.abs(x–y) < 1E-9 or similar
  – Or better, compare difference in ulps (next slide)

• Do not use floating-point for currency ($, kr)
  – Use C# `decimal` or java.math.BigDecimal
  – Or use `long`, and store amount as cents or øre

• A `double` stores integers <= $2^{53}-1 \approx 8\times10^{15}$ exactly

• To compute with very small positive numbers (probabilities) or very large positive numbers (combinations), use their logarithms
Approximate comparison

- Often useless to compare with "=="
- Fast relative comparison: difference in ulps
- Consider x and y as longs, subtract:

```java
static boolean almostEquals(double x, double y, int maxUlps) {
    long xBits = Double.doubleToRawLongBits(x),
            yBits = Double.doubleToRawLongBits(y),
            MinValue = 1L << 63;
    if (xBits < 0)
        xBits = MinValue - xBits;
    if (yBits < 0)
        yBits = MinValue - yBits;
    long d = xBits - yBits;
    return d != MinValue && Math.abs(d) <= maxUlps;
}
```

1.0 == 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1 is false
almostEquals(1.0, 0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1+0.1, 16) is true
Transcendental function surprise

• The `exp()` function changed unexpectedly:

```
Math::Exp(float64) ænderer opførsel med applikation af KB3083185 og KB3098785
... double c = 0.22211390926973425; double x = -1.5045649238845; double calc = Math.Exp(x);
... Bygget med release x64 [...] bliver c og calc ens før bemeldte updates, mens de efter er minimalt forskellige [...]”
```

Troels Damgaard 16 Nov 2015

• Indeed, the two numbers differ by 1 ulp:

```
0 01111111100 110001101110001110101000010000100110011011011001101101100111011 = 0.22211390926973426
0 01111111100 1100011011100011101010000100001001100110110110011011001110100 = 0.22211390926973429
```

• ... and are on either side of the precise result:

```
0.22211390926973426 < 0.22211390926973427457... < 0.22211390926973429
```
EXP-lation from Microsoft

“The underlying implementation of some of the math functions changed due to an update in the CLR between .net versions 4.5.1. and 4.5.2 some of the base math implementation is licensed code from Intel and AMD and periodic changes are incorporated to facilitate more performant operations on newer chip sets – there may be a marginal change in accuracy but MS ensure this stays within +-1ulp.

The following may have marginal differences between versions within +-1ULP: cos, cosf, exp, expf, log, log10, log10f, logf, pow, powf, sin, sinf, tan, and tanf. [...]”

Mail from Holly Muenchow via Mads Torgersen, 18 Nov 2015
What is that number really?

- Java's `java.math.BigDecimal` can display the exact number represented by double `d`:

  ```java
  new java.math.BigDecimal(d).toString()
  ```

```java
double 0.125 = 0.125
float 0.125f = 0.125

double 0.1
  is 0.1000000000000000055511151231257827021181583404541015625
float 0.1f
  is 0.10000001490116119384765625

double 0.01
  is 0.01000000000000000020816681711721685132943093776702880859375
float 0.01f
  is 0.009999999977648258209228515625
```
References

• Java example code and more:
  http://www.itu.dk/people/sestoft/bachelor/Numbers.cs
  http://www.itu.dk/people/sestoft/bachelor/Numbers.java
  http://www.itu.dk/people/sestoft/javaprecisely/java-floatingpoint.pdf
  http://www.itu.dk/people/sestoft/papers/numericperformance.pdf
• William Kahan notes on IEEE 754:
  http://www.cs.berkeley.edu/~wkahan/ieee754status/
  http://www.cs.berkeley.edu/~wkahan/ieee754status/754story.html
• General Decimal Arithmetic (Mike Cowlishaw, IBM)
  http://speleotrove.com/decimal/
• C# specification (Ecma International standard 334):
• How to compare floating-point numbers (in C):