`n != 0 && -n == n`

`z+1 == z`

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

`v+w-w != v`

`y != y`

Peter Sestoft

BSWU First-Year Project

Thursday 20 February 2020

`x+1 < x`

`p == n && 1/p != 1/n`

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

  $805_{10} = 8*10^2 + 0*10^1 + 5*10^0 = 805$

  A place is worth 10 times that to the right

- Binary notation

  $1101_2 = 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0 = 13$

  A place is worth 2 times that to the right

2^0	1
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256

- Positional number systems:
  - Base is 10 or 2 or 16 or …
- Any non-positional number systems?

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

  $325_{16} = 3 * 16^2 + 2 * 16^1 + 5*16^0 = 805$

  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

16^0	1
16^1	16
16^2	256
16^3	4096
16^4	65536

0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

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 $-n = n$

```
00000000000000000000000000000000 = 0
00000000000000000000000000000001 = 1
00000000000000000000000000000010 = 2
00000000000000000000000000000011 = 3
01111111111111111111111111111111 = 2147483647
11111111111111111111111111111111 = -1
11111111111111111111111111111110 = -2
11111111111111111111111111111101 = -3
10000000000000000000000000000000 = -2147483648
```

An obviously non-terminating loop?

```
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, …

$0.5 = 0.1_2$ $2.125 = 10.001_2$

$0.25 = 0.01_2$ $7.625 = 111.101_2$

$0.75 = 0.11_2$ $118.625 = 1110110.101_2$

$0.125 = 0.001_2$

- 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 * 2
  - B) 59.3125 = 118.625 / 2
  - C) 1186.25 = 118.625 * 10
  - D) 11.8625 = 118.625 / 10

Check: Binary fractions

- What is 27.375 represented as a binary fraction?

  A) 11010.011_2

  B) 11011.011_2

  C) 11011.101_2

  D) 11011.001_2

Some nasty fractions

- Some numbers are not representable as finite *decimal* fractions:

  $1/7 = 0.142857142857142857\ldots_{10}$

- Same problem with *binary* fractions:

  $1/10 = 0.0001100110011001100110011100\ldots_{2}$

- 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;
```

Does **not** terminate!

Values of d:
0.10000000000000000000
0.20000000000000000000
0.30000000000000004000
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 * 2^{e-b} * (f + 1)
- Representation:
  - Sign s (0=positive, 1=negative), exponent e, fraction f

```
s eeeeeeee fffffffffffffffffffffff
0 01111111 00000000000000000000000 = 1.0
```

`float`

Java, C#	bits	e bits	f bits	range	bias b	sign. digits
`float`, `binary32`	32	8	23	$\pm10^{-44}$ to $\pm10^{38}$	127	7
`double`, `binary64`	64	11	52	$\pm10^{-323}$ to $\pm10^{308}$	1023	15
Intel extended	80	15	64	$\pm10^{-4932}$ to $\pm10^{4932}$	16635	19

Understanding the representation

- *Normalized* numbers
  - Choose exponent e so the significand is 1.ffffff…
  - Hence we need only store the .ffffff… not the 1.
- Exponent is unsigned but a bias is subtracted
  - For 32-bit float the bias b is 127

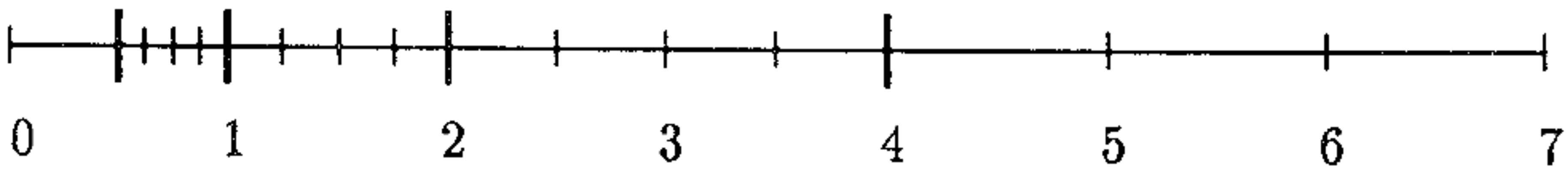
```
s eeeeeeee fffffffffffffffffffffff
0 00000000 00000000000000000000000 = 0.0
1 00000000 00000000000000000000000 = -0.0
0 01111111 00000000000000000000000 = 1.0
0 01111110 00000000000000000000000 = 0.5
1 10000101 11011010100000000000000 = -118.625
0 01111011 10011001100110011001101 = 0.1
0 01111111 00000000000000000000001 = 1.0000001
```

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 eeeeeeee fffffffffffffffffffffff
1 10000101 11011010100000000000000 = -118.625
```

The normalized number line



- Representable with 2 f bits and 2 e bits:

  (If bias = 1, then exponent is -1, 0, 1, or 2)

  $1.00_2 \times 2^{-1} = 0.5$
  $1.01_2 \times 2^{-1} = 0.625$
  $1.10_2 \times 2^{-1} = 0.75$
  $1.11_2 \times 2^{-1} = 0.875$
  $1.00_2 \times 2^{0} = 1$
  $1.01_2 \times 2^{0} = 1.25$
  $1.10_2 \times 2^{0} = 1.5$
  $1.11_2 \times 2^{0} = 1.75$

  $1.00_2 \times 2^{1} = 2$
  $1.01_2 \times 2^{1} = 2.5$
  $1.10_2 \times 2^{1} = 3$
  $1.11_2 \times 2^{1} = 3.5$
  $1.00_2 \times 2^{2} = 4$
  $1.01_2 \times 2^{2} = 5$
  $1.10_2 \times 2^{2} = 6$
  $1.11_2 \times 2^{2} = 7$

- 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 eeeeeeee fffffffffffffffffffffff
0 01111111 00000000000000000000000 = 1.0
0 01111111 00000000000000000000001 = 1.0000001
```

1 ulp difference

- 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?

	s	eeeeeeeee	fffffffffffffffffffffff
A	0	000000100_2 = 4	10110110000000000000000
B	1	100000011_2 = 131	10110110000000000000000
C	0	100000101_2 = 5	10110110000000000000000
D	1	100000010_2 = 130	10110110000000000000000
E	1	100000010_2 = 2	10110110000000000000000

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), …

Exponent e-b	Represented number
−126…127	Normal: $\pm 10^{-38}$ to $\pm 10^{38}$
−127	Denormal, or zero: $\pm 10^{-44}$ to $\pm 10^{-38}$, and ± 0.0
128	Infinities, when f=0…0
128	NaNs, when f=1xx…xx

```
s eeeeeeee fffffffffffffffffffffff
1 10000101 11011010100000000000000 = -118.625
0 00000000 00010000000000000000000 = 7.346E-40
0 11111111 00000000000000000000000 = +Infinity
1 11111111 00000000000000000000000 = -Infinity
s 11111111 10000000000000000000000 = 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 $x=1.01_2 * 2^{-126}$ and $y=1.00_2 * 2^{-126}$:

```
s eeeeeeee fffffffffffffffffffffff
0 00000001 01000000000000000000000 = x
0 00000001 00000000000000000000000 = y
0 00000000 01000000000000000000000 = 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

```
s eeeeeeeeeee ffffffffffffffffffffffffffffffffffffffffffffffffffff
0 00000000000 0000000000000000000000000000000000000000000000000000 = +0.0
1 00000000000 0000000000000000000000000000000000000000000000000000 = -0.0
0 01111111111 0000000000000000000000000000000000000000000000000000 = 1.0
0 01111111110 0000000000000000000000000000000000000000000000000000 = 0.5
1 10000000101 1101101000000000000000000000000000000000000000000000 = -118.625
0 11111111111 0000000000000000000000000000000000000000000000000000 = +Infinity
1 11111111111 0000000000000000000000000000000000000000000000000000 = -Infinity
s 11111111111 1000000000000000000000000000000000000000000000000000 = NaN
0 00000000000 0001000000000000000000000000000000000000000000000000 = 1.39E-309
0 01111111011 1001100110011001100110011001100110011001100110011010 = 0.1
0 01111111110 1111111111111111111111111111111111111111111111111111 = 0.999...9
```

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:
0.1000000000000000055511151231257827021181583404541015625

IEEE addition

+	-Inf	-2.0	-0.0	0.0	2.0	+Inf	NaN
-Inf	-Inf	-Inf	-Inf	-Inf	-Inf	NaN	NaN
-2.0	-Inf	-4.0	-2.0	-2.0	0.0	+Inf	NaN
-0.0	-Inf	-2.0	-0.0	0.0	2.0	+Inf	NaN
0.0	-Inf	-2.0	0.0	0.0	2.0	+Inf	NaN
2.0	-Inf	0.0	2.0	2.0	4.0	+Inf	NaN
+Inf	NaN	+Inf	+Inf	+Inf	+Inf	+Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

IEEE subtraction

-	-Inf	-2.0	-0.0	0.0	2.0	+Inf	NaN
-Inf	NaN	-Inf	-Inf	-Inf	-Inf	-Inf	NaN
-2.0	+Inf	0.0	-2.0	-2.0	-4.0	-Inf	NaN
-0.0	+Inf	2.0	0.0	-0.0	-2.0	-Inf	NaN
0.0	+Inf	2.0	0.0	0.0	-2.0	-Inf	NaN
2.0	+Inf	4.0	2.0	2.0	0.0	-Inf	NaN
+Inf	+Inf	+Inf	+Inf	+Inf	+Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

IEEE multiplication

*	-Inf	-2.0	-0.0	0.0	2.0	+Inf	NaN
-Inf	+Inf	+Inf	NaN	NaN	-Inf	-Inf	NaN
-2.0	+Inf	4.0	0.0	-0.0	-4.0	-Inf	NaN
-0.0	NaN	0.0	0.0	-0.0	-0.0	NaN	NaN
0.0	NaN	-0.0	-0.0	0.0	0.0	NaN	NaN
2.0	-Inf	-4.0	-0.0	0.0	4.0	+Inf	NaN
+Inf	-Inf	-Inf	NaN	NaN	+Inf	+Inf	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

IEEE division

/	-Inf	-2.0	-0.0	0.0	2.0	+Inf	NaN
-Inf	NaN	+Inf	+Inf	-Inf	-Inf	NaN	NaN
-2.0	0.0	1.0	+Inf	-Inf	-1.0	-0.0	NaN
-0.0	0.0	0.0	NaN	NaN	-0.0	-0.0	NaN
0.0	-0.0	-0.0	NaN	NaN	0.0	0.0	NaN
2.0	-0.0	-1.0	-Inf	+Inf	1.0	0.0	NaN
+Inf	NaN	-Inf	-Inf	+Inf	+Inf	NaN	NaN
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

IEEE equality and ordering

==	-Inf	-2.0	-0.0	0.0	2.0	+Inf	NaN
-Inf	true	false	false	false	false	false	false
-2.0	false	true	false	false	false	false	false
-0.0	false	false	true	true	false	false	false
0.0	false	false	true	true	false	false	false
2.0	false	false	false	false	true	false	false
+Inf	false	false	false	false	false	true	false
NaN	false	false	false	false	false	false	false

- Equality (==, !=)
  - A NaN is not equal to anything, not even itself
  - So if **`y`** is NaN, then **`y != 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

`sqrt(-2.0) = NaN`	`sqrt(NaN) = NaN`
`log(0.0) = -Inf`	`log(NaN) = NaN`
`log(-1.0) = NaN`	
`sin(Inf) = NaN`	`sin(NaN) = NaN`
`asin(2.0) = NaN`	
`exp(10000.0) = Inf`	`exp(NaN) = NaN`
`exp(-Inf) = 0.0`	
`pow(0.0, -1.0) = Inf`	`pow(NaN, 0.0) = 1 in Java`

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) 1.0×10^{-6}

B) 1.0×10^{8}

C) 1.0×10^{-8}

D) 1.0×10^{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==z
  - because only 52 digits in fraction

```
0 10000110100 0000000000000000000000000000000000000000000000000000=z
0 10000110100 0000000000000000000000000000000000000000000000000000=z+1
```

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

```
v   = 9876543210.200000
w   = 9876543210.100000
v-w =            0.10000038146972656
```

Garbage, why?

```
v   = 9876543210.20000076293945312500
w   = 9876543210.10000038146972656250
v-w =            0.10000038146972656250
```

The exact actual numbers

```
0 10000100000 0010011001011000000010110111010100011001100110011010 = v
0 10000100000 0010011001011000000010110111010100001100110011001101 = w

0 01111111011 1001100110011010000000000000000000000000000000000000 = v-w
```

Would be non-zero in full-precision 0.1

Case: Solving a quadratic equation

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

$$x_1 = \frac{-b + \sqrt{d}}{2a} \qquad x_2 = \frac{-b - \sqrt{d}}{2a}$$

  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 $\sqrt{d} >= 0$, 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

```
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);
}
```

Bad

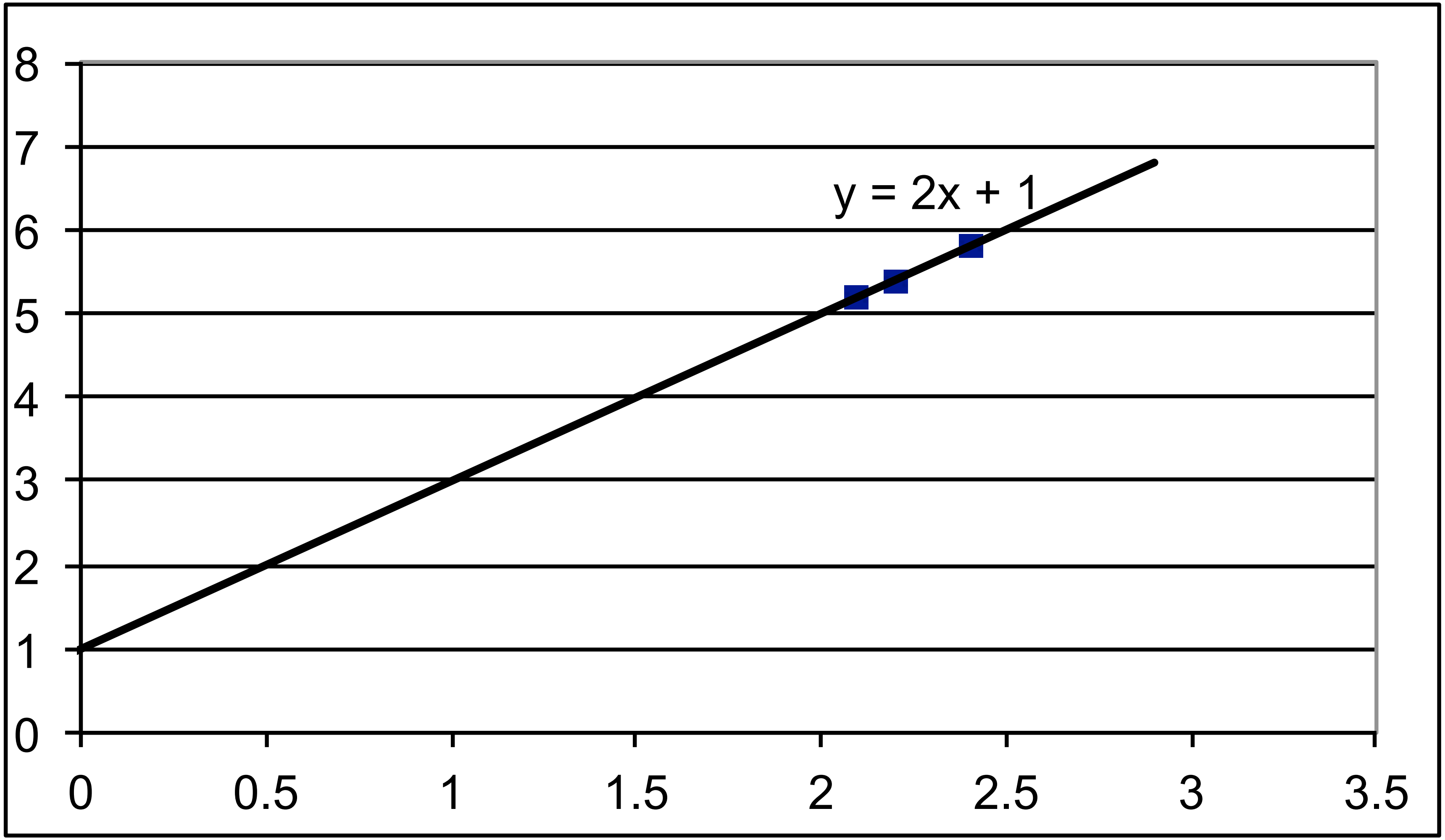
```
double d = b * b - 4 * a * c;
if (d > 0) {
  double y = Math.sqrt(d);
  double x1 = b > 0 ? (-b - y)/(2*a) : (-b + y)/(2*a);
  double x2 = c / (x1 * a);
} else ...
```

Good

- When a=1, b=10^9, c=1 we get
  - Bad algorithm: x1 = -1.00000e+09 and x2 = 0.00000
  - Good algorithm: x1 = -1.00000e+09 and x2 = -1.00000e-09

Case: Linear regression

- Points (2.1, 5.2), (2.2, 5.4), (2.4, 5.8) have regression line y = α + β x with α = 1 and β = 2



Bad way to compute α and β

```
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;
```

Large and nearly identical

- 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*SX/n`**

Better way to compute α and β

```
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

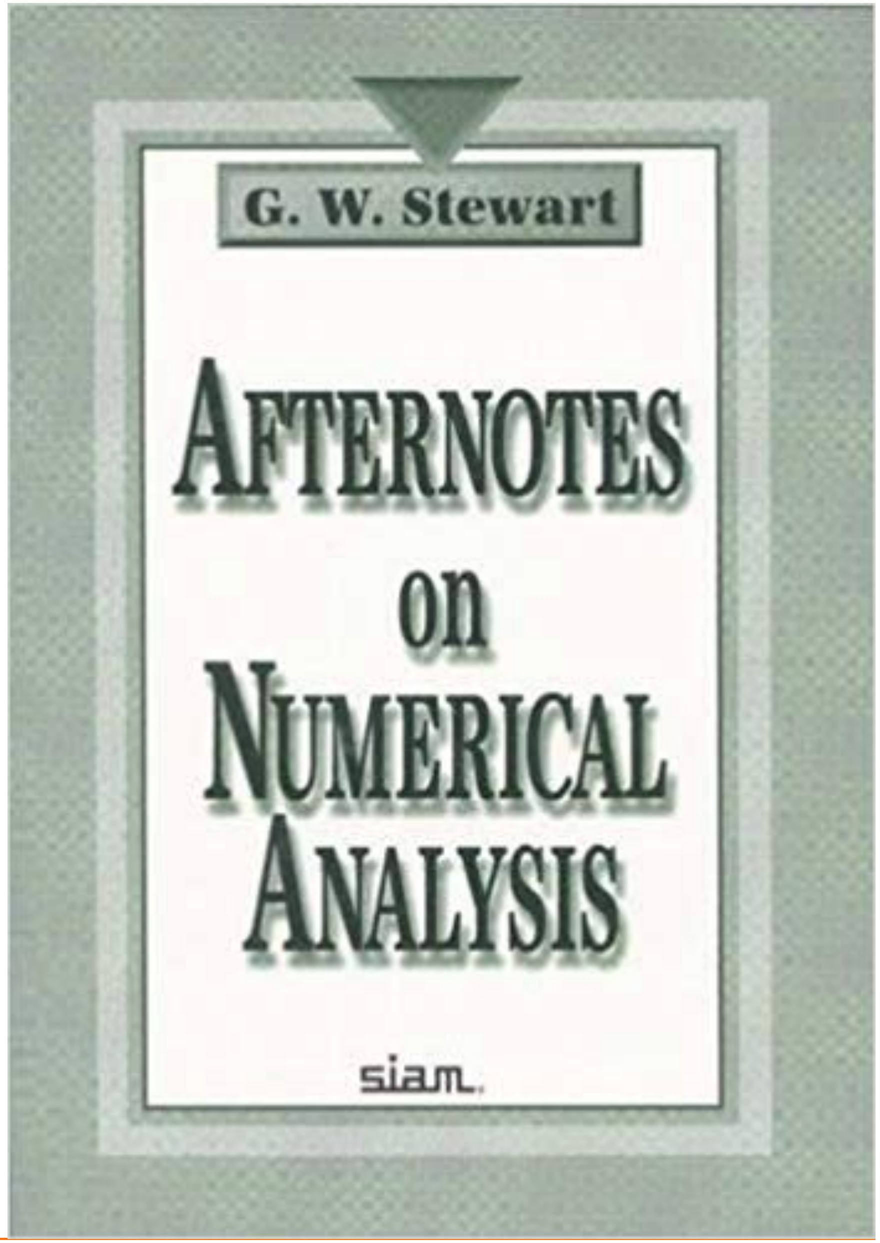
Move		Bad	Good	Correct
0	α	1.000000	1.000000	**1.000000**
	β	2.000000	2.000000	**2.000000**
10 M	α	3.233333	-9999998.99	**-9999999.00**
	β	1.000000	2.000000	**2.000000**
50 M	α	50000005.47	-49999999.27	**-49999999.00**
	β	-0.000000	2.000000	**2.000000**

Wrong

Very wrong!!

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

20,000 elements

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

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

Naïve sum, error = 992

```
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

C is the error in the sum S

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 <= $2^{53}-1 \approx 8*10^{15}$ 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:

```
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) ændrer 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 1100011011100011101010000100001001100110110110011011 = 0.22211390926973426
0 01111111100 1100011011100011101010000100001001100110110110011100 = 0.22211390926973429
```

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

```
  0.22211390926973426
< 0.222113909269734274 57...
< 0.22211390926973429
```

EXP-lanation 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`**:

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

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

```
double 0.1
       is 0.1000000000000000055511151231257827021181583404541015625
float 0.1f
       is 0.100000001490116119384765625
```

```
double 0.01
       is 0.01000000000000000020816681711721685132943093776702880859375
float 0.01f
       is 0.009999999776482582092285156250
```

References

- David Goldberg: *What every computer scientist should know about floating-point arithmetics*. ACM Comp Surv 23 (1) 1991. http://www.itu.dk/people/sestoft/bachelor/IEEE754_article.pdf
- Ole Østerby: Numerical analysis. Aarhus University 2002 http://daimi.au.dk/~oleby/notes/nae.pdf
- R. Mak: *Java Number Cruncher: The Java Programmer's Guide to Numerical Computing.* Prentice-Hall 2002.
- 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
- http://en.wikipedia.org/wiki/IEEE_754-1985
- 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):
  http://www.ecma-international.org/publications/standards/Ecma-334.htm
- How to compare floating-point numbers (in C):
  https://randomascii.wordpress.com/2012/02/25/comparing-floating-point-numbers-2012-edition/