Alma Mater Studiorum-Università di Bologna
Scuola di Ingegneria

Numeri reali: rappresentazione, errori di calcolo, conseguenze

Corso di Laurea in Ingegneria Informatica

Anno accademico 2021/2022

Prof. ENRICO DENTI
Dipartimento di Informatica – Scienza e Ingegneria (DISI)

NUMERI REALI: ILLUSIONE OTTICA ?

- Secondo l'uomo della strada, un calcolatore
  "non sbaglia mai i calcoli"
- In realtà, con i numeri reali non è vero:
  - il calcolatore sbaglia i calcoli praticamente sempre, ma in modo *controllato* e *prevedibile*
  - la sensazione che non sia così è dovuta all' *illusione ottica* delle funzioni di I/O, che gestiscono lettura e scrittura di valori reali in modo da "non spaventare" l'utente finale
  - …ma opportuni esperimenti possono facilmente rivelarlo ☺
- Un ingegnere deve conoscere la verità
  - per comprendere cosa accade *davvero*
  - per poter prendere eventuali *contromisure*

ESPERIMENTO

Java

```
public static void test1() {
   double val1 = 0.1;            // sembra 0.1 .. ma non lo è !
   double val2 = 0.3;            // sembra 0.3 .. ma non lo è !
   System.out.println(val1); // stampa 0.1 .. ma è un'illusione !
   System.out.println(val2); // stampa 0.3 .. ma è un'illusione !

   // … e infatti, il triplo di val1 non è val2 !!!

   double d = val1 + val1 + val1;  // non è il triplo di val1…
   if (d != val2) { System.out.print("ERRORE di ");
    System.out.println(d - val2);} // differenza di 5.5E-17
}
```

```
val1 = 0.1
val2 = 0.3
ERRORE di 5.551115123125783E-17
```

ASTRAZIONE vs REALTÀ

- I numeri reali sono un'astrazione matematica
- Il modello concreto la approssima soltanto
  - MOTIVO: un modello reale non può ospitare infinite cifre, perché nessun dato può essere lungo "infiniti bit"
  - in matematica ce la si cava perché *spesso si trattano i valori in forma simbolica, senza fare calcoli*
  - MA un linguaggio di programmazione imperativo non ha tale capacità: "svolge i calcoli"… e *nel farlo introduce errori*
- di conseguenza
  - i singoli valori sono tipicamente *errati* in una certa misura
  - effettuando operazioni, facilmente *gli errori si accumulano*
  - ciò può portare a risultati errati o perfino assurdi

L'ASTRAZIONE NUMERO REALE

- Un numero reale *può non essere finitamente rappresentabile* come stringa di simboli
  - in nessuna base: numeri irrazionali (π, e, ...)
  - in alcune basi: numeri razionali *periodici*

- In effetti, il fatto che la rappresentazione di un numero *razionale* sia periodica o meno *non dipende dal numero in sé,* ma solo dalla *base di rappresentazione* scelta per rappresentarlo
- OGNI numero razionale è periodico in certe basi e non in altre: è *solo la nostra abitudine a considerare solo la base 10 a far pensare che certi numeri siano "più fortunati" di altri..*

Dunque, da cosa dipende esattamente il "destino" di un numero?

NUMERI REALI .. PERIODICI?

- La rappresentazione di un numero reale è *periodica* se non tutti i fattori primi del denominatore della forma fratta sono presenti nella base di rappresentazione
  - in base 10 sono finitamente rappresentabili *tutti e soli quei valori costruibili con 1/2 e 1/5, combinati/elevati a qualunque esponente*
    - 0.02 non è periodico perché generato da $1/50 = (1/2)(1/5)^2 = 2/100$
    - 0.(3) è periodico perché 1/3 non è costruibile finitamente con 1/2 e 1/5:

      infatti, $\frac{1}{3} = \frac{3}{9} = \frac{3}{10}\sum_{k \geq 0} \left(\frac{1}{10}\right)^k$ che è una somma di infiniti termini:

      $$\frac{1}{3} = \frac{3}{10} + \frac{3}{100} + \frac{3}{1000} + \frac{3}{10000} + \cdots = 0{,}3333 \ldots$$
  - in particolare, quindi, la rappresentazione *non è mai periodica nella base B del denominatore* della forma fratta:
    - $1/3 = (0.333333333333...)_{10} = (0.1)_3$
    - $8/7 = (1.142857142857...)_{10} = (1.1)_7$

NUMERI REALI IN DIVERSE BASI

- A causa di ciò:
  - se la rappresentazione di un numero razionale è periodica in base B, sarà *periodica anche in base B' = B/k* , dato che perdendo fattori primi la situazione può solo peggiorare
  - viceversa, aumentando la quantità di fattori primi (B' = B*k) la situazione può *migliorare* (grazie al nuovo fattore primo k) e dunque un numero periodico potrebbe divenire finitamente rappresentabile
  - ovviamente, ciò non accadrà mai se $B' = B^n$ poiché in tal caso i fattori primi resteranno gli stessi
- In pratica:
  - un numero la cui rappresentazione sia periodica in base 10 sarà sicuramente periodico anche in base 2 o in base 5 (perde fattori..)
  - al contrario, un numero periodico in base 2 può essere o non essere periodico in base 10, ma lo è certamente in base 4, 8, 16, ...

DALL'ASTRAZIONE AL MODELLO

- L'astrazione numero reale può implicare una rappresentazione infinita di cifre, ma in matematica non è un problema perché *non si scrivono!*
  - si adottano rappresentazioni simboliche (4/3, π, e), che in quanto tali sono sempre *esatte e finite*
- Il modello concreto invece deve basarsi sulla *sequenza di cifre*, che tipicamente è *infinita*
  - non c'è 4/3: c'è 1,333333…
  - non c'è π : c'è 3,1415…

Come definire il modello concreto
di numero reale in modo efficiente?

NUMERI REALI: MANTISSA E RESTO

- Fissata una base B, è sempre possibile scrivere un numero reale V come *somma di due contributi:*
  - una MANTISSA di *n cifre*
  - il corrispondente RESTO

$$V \equiv M + R$$

- Ad esempio, se V=31.4357, in base B=10:
  - con n=3 cifre: V = 31.4 + 0.0357
  - con n=4 cifre: V = 31.43 + 0.0057
  - con n=5 cifre: V = 31.435 + 0.0007

MANTISSA E RESTO: PROBLEMI

- Così, però, *la mantissa potrebbe risultare molto grande, molto piccola, o impossibile da esprimere su n cifre,* in base all'ordine di grandezza di V:
- Ad esempio, se V=.014357, in base B=10:
  - con n=2 cifre: V = .01 + .004357
  - con n=3 cifre: V = .014 + .000357
  - con n=4 cifre: V = .0143 + .000057
  - ma attenzione:
    con n=1 cifra: V = .0 + .14357
- Se invece V=1567.35, in base B=10:
  - con n=3 cifre: V = 156? + ???

MANTISSA E RESTO: RIDEFINIZIONE

- Per evitare questi assurdi, si conviene di scrivere la *mantissa* in forma *esponenziale:*

$$M \equiv m * B^{esp}$$

da cui:

$$V \equiv m * B^{esp} + R$$

scegliendo esp in modo che la mantissa risulti *normalizzata, ossia compresa fra 0 e 1 (escluso):*

$$1/B \leq m < 1$$

- Per analogia, spesso si esprime anche il resto R in forma esponenziale, ponendo $R = r * B^{esp-n}$
così, anche per r vale l'analoga relazione $1/B \leq r < 1$

MANTISSA E RESTO NORMALIZZATI

- Se V=0.014357, in base B=10, risulta esp = -1:
  - con n=2 cifre: $V = .14 * 10^{-1} + .000357$
  - con n=3 cifre: $V = .143 * 10^{-1} + .000057$
  - con n=4 cifre: $V = .1435 * 10^{-1} + .000007$

  ovvero, normalizzando anche il resto (esp-n = -4):
  - con n=3 cifre: $V = .143 * 10^{-1} + .57 * 10^{-4}$

- Se invece V=1567.35, in base B=10, risulta esp = +4:
  - con n=3 cifre: $V = .156 * 10^{+4} + 7.35$
  - con n=4 cifre: $V = .1567 * 10^{+4} + .35$

  ovvero, normalizzando anche il resto (esp-n = 1):
  - con n=3 cifre: $V = .156 * 10^{+4} + .735 * 10^{1}$
  - con n=4 cifre: $V = .1567 * 10^{+4} + .35 * 10^{0}$

NUMERI REALI: IL VINCOLO

- Un numero reale ha spesso una rappresentazione infinita in una data base, ma rappresentare infinite cifre è impossibile
- Perciò, assumiamo come rappresentazione approssimata del numero V il solo contributo $m * B^{esp}$

$$V \approx m * B^{esp}$$

- Il resto si trascura $\rightarrow$ Errore di *troncamento*

SCELTE OPERATIVE

- In pratica dobbiamo stabilire:
  - il numero N di cifre binarie (bit) per la mantissa
  - il numero P di cifre binarie (bit) per l’esponente
  - la codifica da adottare per l'esponente
  - come rappresentare *il segno* del numero
- Osservazione: nel caso B=2, $1/2 \leq m < 1$
  ossia *il primo bit dopo la virgola è sempre 1*
- Quindi, *si può evitare di scriverlo esplicitamente,* perché *un bit prefissato non porta informazione*
- N cifre binarie si rappresentano con soli N-1 bit

LE SPECIFICHE IEEE-754

	`float`	`double`
bit di segno	1 bit (il più significativo) 0 = **+** 1 = **-**	
mantissa	**N = 24 bit** **rappresentata con 23 bit, dal meno al più significativo**	**N = 53 bit** **rappresentata con 52 bit, dal meno al più significativo**
esponente	**P = 8 bit**	**P = 11 bit**
rappresent. esponente	rappresentazione con **eccesso 2^{P-1}-2**	
	eccesso 126 • **da 127 a 254 ↔ esp. [1..128]** • **da 1 a 125 ↔ esp. [-125..-1]** • **0 e 255 ↔ *casi speciali***	**eccesso 1022** • **da 1023 a 2046 ↔ esp. [1..1024]** • **da 1 a 1021 ↔ esp. [-1021..-1]** • **0 e 2047 ↔ *casi speciali***
valori rap-presentabili	**MIN = .1 * 2^{1-126}** **MAX = .1111..111 * $2^{254-126}$** **RANGE ≈ [2^{-126} ... 2^{128}]** **[1.2 * 10^{-38} … 3.4 * 10^{38}]**	**MIN = .1 * 2^{1-1022}** **MAX = .1111..111 * $2^{2046-1022}$** **RANGE ≈ [2^{-1022} ... 2^{1024}]** **[1.3 * 10^{-308} … 0.7 * 10^{308}]**

LE SPECIFICHE IEEE-754

Come mai questa scelta?

Perché così è immediato confrontare due valori!

Basta controllare bit a bit gli esponenti..

		…e
mantissa	**N = 24 bit** rappresentata con **23 bit**, dal meno al più significativo	**N = 53 bit** rappresentata con **52 bit**, dal meno al più significativo
esponente	**P = 8 bit**	**P = 11 bit**
rappresent. esponente	rappresentazione con **eccesso 2^{P-1}-2**	
	eccesso 126 • da 127 a 254 ↔ esp. [1..128] • da 1 a 125 ↔ esp. [-125..-1] • 0 e 255 ↔ *casi speciali*	**eccesso 1022** • da 1023 a 2046 ↔ esp. [1..1024] • da 1 a 1021 ↔ esp. [-1021..-1] • 0 e 2047 ↔ *casi speciali*
valori rap-presentabili	MIN = $.1 * 2^{1-126}$ MAX = $.1111..111 * 2^{254-126}$ RANGE ≈ [2^{-126} ... 2^{128}] [$1.2 * 10^{-38}$ … $3.4 * 10^{38}$]	MIN = $.1 * 2^{1-1022}$ MAX = $.1111..111 * 2^{2046-1022}$ RANGE ≈ [2^{-1022} ... 2^{1024}] [$1.3 * 10^{-308}$ … $0.7 * 10^{308}$]

LE SPECIFICHE IEEE-754

- Casi speciali (esp_{MAX}=255 per i float, 2047 per i double):

esp=0,	m=0	rappresenta 0.0
$esp=esp_{MAX}$	m=0	rappresenta $\pm\infty$
$esp=esp_{MAX}$	$m\neq0$	rappresenta *NaN*
esp=0	$m\neq0$	rappresenta *valori non normalizzati*

dove

- NaN *(Not a Number)* è il risultato di operazioni *indeterminate* o *impossibili:*

  $0 / 0$ $\quad$ $0 * \infty$ $\quad$ $\infty - \infty$ $\quad$ ∞ / ∞

- i valori non normalizzati consentono di rappresentare numeri "più piccoli del minimo valore normalizzato possibile", ampliando così il range di valori rappresentabili a parità di costo.

NUMERI REALI: CIFRE SIGNIFICATIVE

- Assumendo $V \approx m * B^{esp}$, si trascura il resto $r * B^{esp-N}$
- Poiché nella forma normalizzata r<1, l'errore in termini assoluti vale:

$$E_{assoluto} \leq B^{esp-N}$$

- Tuttavia, esso non è molto significativo *in sé:* lo è molto di più se *rapportato al valore del numero*

$$E_{relativo} \leq B^{esp-N} / (m * B^{esp})$$

- da cui, poiché $1/B \leq m < 1$,

$$E_{relativo} \leq B^{esp-N} / B^{esp-1} = B^{1-N}$$

NUMERI REALI: CIFRE SIGNIFICATIVE

- L'errore relativo $E_{relativo} \leq B^{1-N}$ permette di *stimare le cifre significative* di un numero in base ai bit della mantissa
- In un float, la mantissa ha 24 bit
  - $E_{relativo} \leq 2^{-23} = 10^{-23*\log 2} = 10^{-7}$
  - ovvero, il numero ha circa 7-8 cifre decimali significative *(più facilmente 8, se si arrotonda anziché troncare)*
- In un double, la mantissa ha 53 bit
  - $E_{relativo} \leq 2^{-52} = 10^{-52*\log 2} = 10^{-16}$
  - ovvero, il numero ha circa 16-17 cifre decimali significative

Per questo **`println`** stampa 8 cifre per i **`float`** e 16 cifre per i **`double`** ☺

```
jshell> System.out.println(1/3.0F)
0.33333334

jshell> System.out.println(1/3.0)
0.3333333333333333
```

NUMERI REALI: ESEMPIO 1

Rappresentazione come float di V = 1.0

- *rappr. normalizzata:* $V = 1.0_{10} = 0.1_2 * 2^1$
- *segno (1 bit):* 0
- *mantissa (24 bit):* .10000000 00000000 00000000
- *esponente (8 bit con eccesso 126)*

esp=1 → 126+1 = 127 → 01111111

segno	esponente	mantissa normalizzata (23 bit, MSB escluso)
0	0111 1111	000 0000 0000 0000 0000 0000

in memoria:

byte 1	byte 2	byte 3	byte 4
0011 1111	1000 0000	0000 0000	0000 0000

NUMERI REALI: ESEMPIO 2

Rappresentazione come float di V = 5.875

- *rappr. normalizzata:* $V = 101.111_2 = .101111_2 * 2^3$
- *segno (1 bit):* 0
- *mantissa (24 bit):* .10111100 00000000 00000000
- *esponente (8 bit con eccesso 126)*

esp=3 → 126+3 = 129 → 10000001

segno	esponente	mantissa normalizzata (23 bit, MSB escluso)
0	1000 0001	011 1100 0000 0000 0000 0000

in memoria:

byte 1	byte 2	byte 3	byte 4
01000000	1011 1100	0000 0000	0000 0000

NUMERI REALI: ESEMPIO 3

Rappresentazione come float di V = -29.1875

- *rappr. normalizzata:* $V = -\ .111010011_2 * 2^5$
- *segno (1 bit):* 1
- *mantissa (24 bit):* .111101001 10000000 00000000
- *esponente (8 bit con eccesso 126)*

esp=5 → 126+5 = 131 → 10000011

segno	esponente	mantissa normalizzata (23 bit, MSB escluso)
1	1000 0011	110 1001 1000 0000 0000 0000

in memoria:

byte 1	byte 2	byte 3	byte 4
1100 0001	1110 1001	1000 0000	0000 0000

NUMERI REALI: ESEMPIO 4

Rappresentazione come float di $V = 0.1_{10}$

- *rappr. normalizzata:* $V = .0(0011)_2$ *periodico!*
- *segno (1 bit):* 0
- *mantissa (24 bit):* .11001100 11001100 11001100
- *esponente (8 bit con eccesso 126)*

esp=-3 → 126-3 = 123 → 01111011

segno	esponente	mantissa normalizzata (23 bit, MSB escluso)
0	0111 1011	100 1100 1100 1100 1100

in memoria:

byte 1	byte 2	byte 3	byte 4
0011 1101	1100 1100	1100 1100	1100 1100

NUMERI REALI: ESEMPIO 4

Rappresentazione come float

Errore di troncamento
o si tronca o si arrotonda
Di solito si arrotonda

- *rappr. normalizzata:* V =.0(0011
- *segno (1 bit):* 0
- *mantissa (24 bit):* .11001100 11001100 11001101
- *esponente (8 bit con eccesso 126)*

esp=-3 → 126-3 = 123 → 01111011

segno	esponente	mantissa normalizzata (23 bit, MSB escluso)
0	0111 1011	100 1100 1100 1100 110**1**

in memoria:

byte 1	byte 2	byte 3	byte 4
0011 1101	1100 1100	1100 1100	1100 110**1**

NUMERI REALI: ESEMPIO 5

Rappresentazione come float di V = 0.15_{10}

- *rappr. normalizzata:* V =$.00(1001)_2$ *periodico!*
- *segno (1 bit):* 0
- *mantissa (24 bit):* .100110011001100110011010
- *esponente (8 bit con eccesso 126)*

esp=-2 $\rightarrow$ 126-2 = 124 $\rightarrow$ 01111100

segno	esponente	mantissa normalizzata (23 bit, MSB escluso)
0	0111 1100	001 1001 1001 1001 1001 1010

in memoria:

byte 1	byte 2	byte 3	byte 4
0011 1110	0001 1001	1001 1001	1001 1010

NUMERI REALI: ESEMPIO 6

Rappresentazione come float di $V = -1/3_{10}$

- *rappr. normalizzata:* $V = -.(01)_2$ *periodico!*
- *segno (1 bit):* 1
- *mantissa (24 bit):* .10101010 10101010 10101011
- *esponente (8 bit con eccesso 126)*

$$esp=-1 \rightarrow 126-1 = 125 \rightarrow 01111101$$

segno	esponente	mantissa normalizzata (23 bit, MSB escluso)
1	0111 1101	010 1010 1010 1010 1010 1011

in memoria:

byte 1	byte 2	byte 3	byte 4
1011 1110	1010 1010	1010 1010	1010 1011

ESPERIMENTI: SPIARE DENTRO JAVA

- Come verificare se quanto calcolato è esatto?
  - occorre un modo per "spiare" dentro la macchina virtuale del linguaggio (Java, C, Pascal, …)
- In Java, la classe wrapper **`Float`** (risp. **`Double`**) fornisce la funzione **`floatToIntBits`** (risp. **`doubleToLongBits`**)
  - tale funzione "estrae" la rappresentazione di un **`float`** (risp. **`double`**) bit per bit e la pone in un **`int`** (risp. **`long`**) pronta per la stampa o altre elaborazioni.

ESEMPIO: SPIARE UN FLOAT in Java

Esempio di codice che "spia" un float **f**:

```
System.out.println("Numero float: " + f);
int internalRep = Float.floatToIntBits(f);
String binaryString =
          Integer.toBinaryString(internalRep);
// estetica: aggiunta di opportuni '0' davanti
while (binaryString.length()<32)
    binaryString = "0" + binaryString;
System.out.println(binaryString);
```

Stampa una stringa di 32 bit:
10111110101010101010101010101011

ESPERIMENTI: MIGLIORIE ESTETICHE

- Per renderla più leggibile per noi, può essere opportuno *separare le varie parti:*

  1 01111101 01010101010101010101011

  o magari, meglio:

  1 01111101 0101010 10101010 10101011

  Per farlo:

```
System.out.println(
    binaryString.charAt(0)  + " " +
    binaryString.substring(1,9) + " " +
    binaryString.substring(9,16) + " " +
    binaryString.substring(16,24) + " " +
    binaryString.substring(24,32) );
```

segno: 1 bit

esponente: 8 bit

mantissa:
7 + 8 + 8 bit

ESPERIMENTI: CODICE

```
public static void decodeFloat(float f) {
  System.out.println("Numero float: " + f);
  int internalRep = Float.floatToIntBits(f);
  String binString = Integer.tobinString(internalRep);
  while (binString.length()<32) binString = '0' + binString;
  System.out.println("Segno:      " + binString.charAt(0));
  System.out.println("Esponente: "
    + binString.substring(1,9) + " ("
    + Integer.valueOf(binString.substring(1,9),2)  + ")");
  System.out.println("Mantissa:  "
    + binString.substring(9,16) + " "
    + binString.substring(16,24) + " "
    + binString.substring(24,32));
 }
```

ESPERIMENTI CON FLOAT (1/2)

V = 1.0

- *segno (1 bit):* 0
- *esponente:* 01111111
- *mantissa (24 bit):* .10000000 00000000 00000000

```
Numero float: 1.0
Segno:      0
Esponente: 01111111 (127)
Mantissa:  0000000 00000000 00000000
```

V = 5.875

- *segno (1 bit):* 0
- *esponente:* 10000001
- *mantissa (24 bit):* .10111100 00000000 00000000

```
Numero float: 5.875
Segno:      0
Esponente: 10000001 (129)
Mantissa:  0111100 00000000 00000000
```

V = -29.1875

- *segno (1 bit):* 1
- *esponente:* 10000011
- *mantissa (24 bit):* .11101001 10000000 00000000

```
Numero float: -29.1875
Segno:      1
Esponente: 10000011 (131)
Mantissa:  1101001 10000000 00000000
```

ESPERIMENTI CON FLOAT (2/2)

V = 0.1

```
Numero float: 0.1
Segno:      0
Esponente: 01111011 (123)
Mantissa:  1001100 11001100 11001101
```

- *segno (1 bit):* 0
- *esponente:* 01111011
- *mantissa (24 bit):* .11001100 11001100 11001101 *(arrotondata)*

V = 0.15

```
Numero float: 0.15
Segno:      0
Esponente: 01111100 (124)
Mantissa:  0011001 10011001 10011010
```

- *segno (1 bit):* 0
- *esponente:* 01111100
- *mantissa (24 bit):* .10011001 10011001 10011010 *(arrotondata)*

V = -1/3

```
Numero float: -0.33333334
Segno:      1
Esponente: 01111101 (125)
Mantissa:  0101010 10101010 10101011
```

- *segno (1 bit):* 1
- *esponente:* 01111101
- *mantissa (24 bit):* .10101010 10101010 10101011 *(arrotondata)*

ESPERIMENTI CON DOUBLE

V = 0.1

- *segno (1 bit):* 0
- *esponente:* 01111111011
- *mantissa (53 bit):* .11001100 11001100 …11010 *(arrotondata)*

```
Numero float: 0.1
Segno:       0
Esponente: 01111011 (123)
Mantissa:  1001100 11001100 11001101
```

```
Numero double: 0.1
Segno:       0
Esponente: 01111111011 (1019)
Mantissa:  1001100 11001100 11001100 11001100 11001100 11001100 11010
```

V = -1/3

- *segno (1 bit):* 1
- *esponente:* 01111111101
- *mantissa (53 bit):* .10101010 10101010 …10101 *(arrotond./troncata)*

```
Numero float: -0.33333334
Segno:       1
Esponente: 01111101 (125)
Mantissa:  0101010 10101010 10101011
```

```
Numero double: -0.33333333333333333
Segno:       1
Esponente: 01111111101 (1021)
Mantissa:  0101010 10101010 10101010 10101010 10101010 10101010 10101
```

ESPERIMENTI … MIXED!

V = 0.1 (valore **`float`** convertito in **`double`**)

- *segno (1 bit):* 0
- *esponente:* 01111111011
- *mantissa (53 bit):* . 11001100 11001100 11001101 00000000 …

```
Numero float: 0.1
Segno:     0
Esponente: 01111011 (123)
Mantissa:  1001100 11001100 11001101
```

```
Numero double: 0.1
Segno:     0
Esponente: 01111111011 (1019)
Mantissa:  1001100 11001100 11001100 11001100 11001100 11001100 11010
```

```
Numero double: 0.100000001490116 12
Segno:     0
Esponente: 01111111011 [illegible]019)
Mantissa:  1001100 110[illegible]00 11001101 00000000 00000000 00000000 00000
```

ATENZIONE: poiché provengono da un float, le cifre oltre la sesta (settima) *non sono significative*

Bit tutti a zero, perché il valore originale, float, era rappresentato su soli 24 bit

OPERAZIONI & ERRORI

- Negli interi:
  - si possono creare errori nelle operazioni
  - MA *i singoli operandi* sono sempre *rappresentati esattamente* (entro il range corrispondente al numero di bit disponibili)
- Nei reali:
  - gli operandi possono già essere affetti da errore *prima ancora* di iniziare le operazioni a causa dell'impossibilità di rappresentare le infinite cifre dei numeri periodici e irrazionali
- È il famigerato *errore di troncamento*, che si manifesta quando il numero di cifre disponibili è insufficiente a rappresentare compiutamente il valore.

ERRORE DI TRONCAMENTO

- Si manifesta quando:
  - il numero è periodico
  - il numero non è periodico ma ha troppe cifre
  - il risultato di un'operazione, a causa un riporto, richiede comunque troppe cifre per essere rappresentato
- Esempi (mantissa di 8 bit, per semplicità)
  - $15.8_{10} = .1111110011001100... * 2^4$ *(è 15.75)*
  - $300.5_{10} = .1001011001_2 * 2^9$ *(è 300)*
  - $151_{10} + 160_{10} =$
    $= .10010111 * 2^8 + .10100000 * 2^8 =$
    $= .100110111 * 2^9$ *(è 310)*

OPERAZIONI FRA REALI & ERRORI

Oltre al troncamento, vi sono altri due sorgenti di errore

- Errore di *incolonnamento*: è causato dalla *necessità di incolonnare i numeri* per poterli sommare o sottrarre
- Errore di *cancellazione*: è la *conseguenza a valle* della presenza di errori di troncamento *a monte,* quando si sottraggono numeri simili fra loro
  - si chiama così perché, a causa del troncamento a monte, alcune cifre del risultato vengono *"virtualmente cancellate",* ossia sono *inaffidabili*

ERRORE DI INCOLONNAMENTO

- L'errore di incolonnamento è dovuto al fatto che, per sommare/sottrarre due numeri, bisogna prima *incolonnarli*
  - se hanno *esponente diverso,* per incolonnarli bisogna per forza *"de-normalizzarne" uno*
    - per minimizzare il danno, si allinea quello di *valore assoluto minore* a quello di valore assoluto maggiore (che resta intonso)
  - ciò causa una *perdita di cifre significative* nel numero che viene "de-normalizzato", a causa dello shift verso destra di tante posizioni quante quelle necessarie per incolonnarlo all'altro

ERRORE DI INCOLONNAMENTO

- Esempio: 96.5 + 1.75
  - Ipotesi: mantissa di 8 bit (per semplicità)
  - $96.5_{10} = .11000001_2 * 2^7$ *(senza errore)*
  - $1.75_{10} = .11100000_2 * 2^1$ *(senza errore)*
- Somma:

$$.11000001_2 * 2^7 +$$
$$.11100000_2 * 2^1 =$$

Cifre condannate: è l'errore di incolonnamento

$$.11000001_2 * 2^7 +$$
$$.00000011_2 * 2^7 =$$
$$.11000100_2 * 2^7$$

Il risultato è 98 anziché *98.25* a causa dello shift a destra di 6 posizioni nel secondo addendo, causato dall'incolonnamento.

ERRORE DI INCOLONNAMENTO: ESPERIMENTO

```
public static void test() {
  double val1 = 12345.0; // numero di 5 cifre..
  double val2 = 1e-13; // proprio al limite: 18 cifre
  // errore di incolonnamento:
  // val1 + val2 COINCIDE con val1 !
  if (val1 + val2 == val1)
     System.out.println("val1 + val2 == val1 !!");
}
```

Con `val2=1e-12`, invece, ce la fa: siamo al limite!

```
val1 + val2 == val1 !!
```

ERRORI: ESPERIMENTO

```
public static void test1() {
   double val1 = 0.1;            // sembra 0.1 .. ma non lo è !
   double val2 = 0.3;            // sembra 0.3 .. ma non lo è !
   System.out.println(val1);     // stampa 0.1 .. ma è un'illusione !
   System.out.println(val2);     // stampa 0.3 .. ma è un'illusione !

   // … e infatti, il triplo di val1 non è val2 !!!
   double d = val1 + val1 + val1;  // non è il triplo di val1…
   if (d != val2) { System.out.print("ERRORE di ");
    System.out.println(d - val2);} // differenza di 5.5E-17
}
```

```
val1 = 0.1
val2 = 0.3
ERRORE di 5.551115123125783E-17
```

ERRORE DI CANCELLAZIONE

- L’errore di cancellazione avviene quando *si sottraggono* due numeri «piuttosto vicini» fra loro
  - accade solo se almeno uno dei due operandi *all’inizio* era stato *troncato,* in quanto ciò gli ha «amputato» alcune cifre
  - la sottrazione di valori «piuttosto vicini" causa infatti *l'introduzione da destra di zeri* per normalizzare il risultato,
MA *quegli zeri non sono significativi* appunto perché discendono dal troncamento iniziale
    - se l'operando non fosse stato troncato, ora quelle cifre entrerebbero in gioco e il risultato sarebbe diverso.

ERRORE DI CANCELLAZIONE

- Esempio: 15.8 - 15.5
  - Ipotesi: mantissa di 8 bit (per semplicità)
  - $15.8_{10} = .11111100_2 * 2^4$ *(con errore tronc.)*
  - $15.5_{10} = .11111000_2 * 2^4$ *(senza errore)*
- Differenza:

$$
\begin{array}{l}
.11111100_2 * 2^4 - \\
.11111000_2 * 2^4 = \\
\hline
.00000100_2 * 2^4 = \;.10000000_2 * 2^{-1}
\end{array}
$$

Cifre cancellate:
è l'errore di cancellazione

Queste cifre sono 0 solo perché abbiamo troncato il 15.8 all'inizio: avrebbero dovuto essere 11001

ERRORI: CONSEGUENZE

- A causa di questi errori, la proprietà associativa *può non essere più verificata*
- Esempio (mantissa di 8 bit, per semplicità)
  - X = 0,75 $\rightarrow$ $.11000000 * 2^0$ *(senza errori)*
  - Y = 65,6 $\rightarrow$ $.10000011 * 2^7$ *(err. troncamento)*
  - Z = 64,0 $\rightarrow$ $.10000000 * 2^7$ *(senza errori)*
- Orrore:

  (X + Y) - Z è *diverso* da X + (Y - Z)

ERRORI: CONSEGUENZE

(X + Y) - Z   *è diverso da*   X + (Y - Z)

Primo caso: (X + Y) - Z	**Secondo caso: X + (Y - Z)**
Prima operazione: A = X + Y	*Prima operazione: A = Y - Z*
$.11000000 * 2^0 +$	$.10000011 * 2^7 -$
$\underline{.10000011 * 2^7 =}$	$\underline{.10000000 * 2^7 =}$
$.00000001 * 2^7 +$ (err. incolonnamento)	$\underline{.00000011 * 2^7 =}$ (da rinormalizzare)
$\underline{.10000011 * 2^7 =}$	$\underline{.11?????? * 2^1 =}$ (errore cancellazione)
$.10000100 * 2^7 \rightarrow A$	$.11000000 * 2^1 \rightarrow A$
Seconda operazione: R = A - Z	*Seconda operazione: R = X + A*
$.10000100 * 2^7 -$	$.11000000 * 2^0 +$
$\underline{.10000000 * 2^7 =}$	$\underline{.11000000 * 2^1 =}$
$\underline{.00000100 * 2^7 =}$ (da rinormalizzare)	$.01100000 * 2^1 +$ (err. incolonnamento)
$\underline{.100????? * 2^2 =}$ (errore cancellazione)	$\underline{.11000000 * 2^1 =}$
$.10000000 * 2^2 \rightarrow R$	$\underline{1.00100000 * 2^1}$ (da rinormalizzare)
	$.10010000 * 2^2 \rightarrow R$ (err. tronc. potenziale)

$R = .10000000 * 2^2$ $R = .10010000 * 2^2$

ERRORE DI CANCELLAZIONE: ESEMPIO

- Teoricamente, 1/3-1/4 = 1/12
- MA in realtà…

```
jshell> System.out.println(1/3.0-1/4.0)
0.08333333333333331

jshell> System.out.println(1/12.0)
0.08333333333333333

jshell> System.out.println(1/3F-1/4F)
0.08333334

jshell> System.out.println(1/12F)
0.083333336
```

ERRORE DI CANCELLAZIONE: CONTROESEMPIO

- L'unico caso in cui il problema *non* si manifesta è se *nessuno dei due operandi* è stato *inizialmente troncato*
  - ad esempio, 1/4-1/64 = 15/64 è rappresentato esattamente:

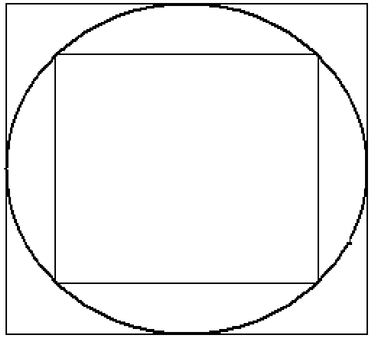
```
jshell> System.out.println(1/4F-1/64F)
0.234375

jshell> System.out.println(1/4.0-1/64.0)
0.234375
```

  - in questo caso la rappresentazione è esatta sia usando un float, sia usando un double

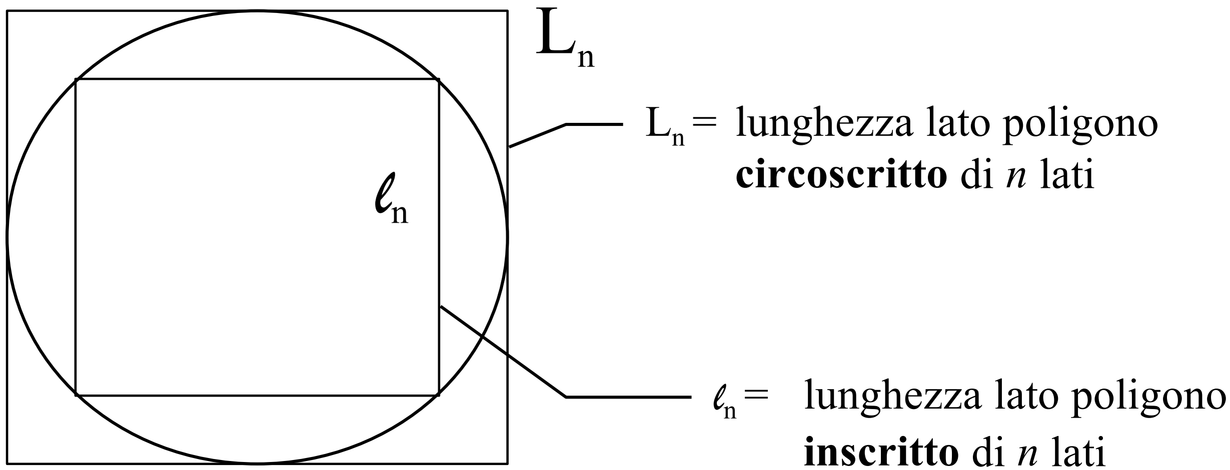
ACCUMULAZIONE DI ERRORI

- La presenza di *errori che si accumulano* può portare a *risultati assurdi*
- Esempio: *calcolo di π con l'algoritmo di Euclide*
  - una circonferenza di raggio 1 è lunga 2π
  - essa può essere approssimata *dall'interno* o *dall'esterno*
    - dall'interno, dal perimetro del poligono regolare di *n* lati *inscritto*
    - dall'esterno, dal perimetro del poligono regolare di *n* lati *circoscritto*
  - più è alto il numero N di lati del poligono, maggiore sarà la precisione del calcolo



ACCUMULAZIONE DI ERRORI

- Valgono le relazioni:
  - ℓ_n = lato del poligono di *n* lati *inscritto*
  - L_n = lato del poligono di *n* lati *circoscritto* $= 2\,\ell \,/\, \sqrt{(4 - \ell^2)}$



L'ALGORITMO IN JAVA (1/2)

```
import java.io.*;
public class PigrecoEuclide {
  public static void main(String[] args) {
  System.out.println("Calcolo di pigreco con FLOAT.");
  System.out.print("Precisione [1e-8] ? ");
  float eps = 1E-8F; // inizializz. richiesta da eccezione
  try{
    BufferedReader kbd = new BufferedReader(
                          new InputStreamReader(System.in));
    eps = Float.parseFloat(kbd.readLine());
  }
  catch(IOException e){
    System.err.println("Errore di input - Program exit");
    System.exit(1);
  }
  ...
```

L'ALGORITMO IN JAVA (2/2)

```
...
 float nlati = 4.0F, ln = (float) Math.sqrt(2.0);
 float smpinf, smpsup;

 do {
   float OC2 = (float) Math.sqrt(4.0 - ln * ln);
   nlati *= 2.0;
   ln = (float) Math.sqrt(2.0F - OC2);
   smpinf = ln * nlati / 2.0F;
   smpsup = ln * nlati / OC2;
   System.out.println("nl=" + nlati + " d2=" + OC2 +
               " piInf=" + smpinf + " piSup=" + smpsup);
 } while ((smpsup-smpinf >= eps) && (nlati < 1e+19));

}
}
```

L'OUTPUT con EPS = 1E-04 (ok)

```
Calcolo di pigreco con FLOAT.
Precisione (suggerito 1e-8)? 1e-4
nl=8.0    d2=1.4142137 piInf=3.0614672 piSup=4.329568
nl=16.0   d2=1.8477591 piInf=3.1214445 piSup=3.378627
nl=32.0   d2=1.9615706 piInf=3.1365461 piSup=3.1979947
nl=64.0   d2=1.9903694 piInf=3.1403334 piSup=3.155528
nl=128.0  d2=1.9975909 piInf=3.1412857 piSup=3.1450741
nl=256.0  d2=1.9993976 piInf=3.1415188 piSup=3.1424654
nl=512.0  d2=1.9998494 piInf=3.1412080 piSup=3.1414444
nl=1024.0 d2=1.9999623 piInf=3.1424513 piSup=3.1425104
```

PRIMO COLLAUDO POSITIVO!

quindi va tutto bene…. (forse..)

L'OUTPUT con EPS = 1E-08: ARGH!

```
Calcolo di pigreco con FLOAT.
Precisione (suggerito 1e-8)? 1e-8
nl=8.0      d2=1.4142137 piInf=3.0614672 piSup=4.329568
nl=16.0     d2=1.8477591 piInf=3.1214445 piSup=3.378627
nl=32.0     d2=1.9615706 piInf=3.1365461 piSup=3.1979947
nl=64.0     d2=1.9903694 piInf=3.1403334 piSup=3.155528
nl=128.0    d2=1.9975909 piInf=3.1412857 piSup=3.1450741
n...          ...976 piInf=3.1415188 piSup=3.1424654
n...          ...494 piInf=3.1412080 piSup=3.1414444
n...          ...623 piInf=3.1424513 piSup=3.1425104
nl=2048.0   d2=...99906 piInf=3.1424513 piSup=3.142466
nl=4096.0   d2=1....9976 piInf=3.1622777 piSup=3.1622815
nl=8192.0   d2=1.999...94 piInf=3.1622777 piSup=3.1622787
nl=16384.0 d2=1.9999999 piInf=2.8284270 piSup=2.8284273
nl=32768.0 d2=2.0 piInf=0.0 piSup=0.0
```

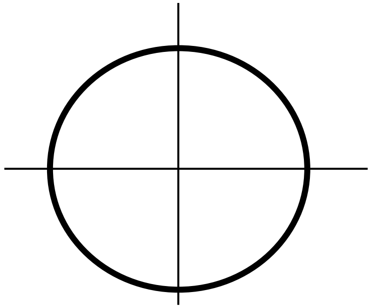
ORRORE!
ora pigreco = 0 !!

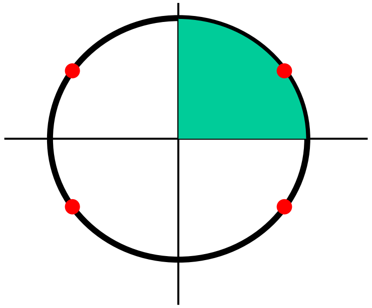
E L'EFFICIENZA?

- Spesso gli algoritmi più «ovvii» per fare un calcolo non sono i più efficienti
  – la matematica del computer non è quella dell'analisi…
  – le funzioni irrazionali e trascendenti sono calcolate con *sviluppi in serie*
- Se l'efficienza è un aspetto chiave, a volte può essere consigliabile *riformulare gli algoritmi*
  – ricorrendo all'aritmetica intera quando possibile
  – oppure, *impostando il problema in un modo diverso*

ESEMPIO:
DISEGNARE UNA CIRCONFERENZA

- L'algoritmo più ovvio prevedrebbe l'uso della relazione $y = \sqrt{R^2 - x^2}$
  - esso però *contiene una radice quadrata*
  - con molti punti da calcolare, è assai inefficiente
- Un approccio «smart» alternativo
  - IDEA: calcolare solo un (mezzo) quadrante e ottenere gli altri punti per simmetria
  - ma soprattutto, *sfruttare la derivata* per calcolare «il prossimo punto» evitando del tutto la radice!





ESEMPIO: DISEGNARE UNA CIRCONFERENZA

- Infatti, poiché $x = R\cos t$ e $y = R\sin t$, vale la relazione $\frac{dy}{dx} = -\frac{x}{y}$
- Essa consente facilmente di determinare il punto adiacente *senza ricorrere ai radicali*
  - si parte da (0,R) e ci si muove solo fino a (R/2, R/2), per evitare punti a tangente verticale

$$\triangle y = -\frac{x}{y} \triangle x$$

  - gli altri 7 pixel si ottengono per simmetria
  - quanto si guadagna dipende dal contesto