

Modulo: Approfondimenti sui Sistemi Aritmetici di un computer: tipo reale floating-point [P2_03]

Unità didattica: Esempi di roundoff [4-AT]

Titolo: Come gli algoritmi possono influenzare l'accuratezza e l'efficienza

Argomenti trattati:

- ✓ Criterio di arresto naturale
- ✓ Calcolo delle radici di un'equazione di 2° grado
- ✓ Esempi di non validità delle proprietà algebriche dei numeri reali
- ✓ Valutazione di un particolare polinomio mediante algoritmo di Horner
- ✓ Somme di molti addendi dello stesso ordine di grandezza
- ✓ Somme di molti addendi ordinati (dello stesso segno)
- ✓ Somme di molti addendi a segno alternato
- ✓ Variabili predefinite dell'ambiente aritmetico

Prerequisiti richiesti: algoritmi di base citati, accuratezza statica e dinamica del Sistema Aritmetico Floating-Point

Nel Sistema Aritmetico Floating-point:

$$\forall a \in F(\beta, t, E_{\min}, E_{\max}),$$

$$\exists \varepsilon > 0 : a \oplus \varepsilon = a$$

↑ addizione floating-point

$$\text{ulp}(a) = \min \{ 0 < \varepsilon \in F(\beta, t, E_{\min}, E_{\max}) : a \oplus \varepsilon > a \}$$

$$\text{ulp}(1) = \text{Epsilon macchina } (\varepsilon_{\text{mach}})$$

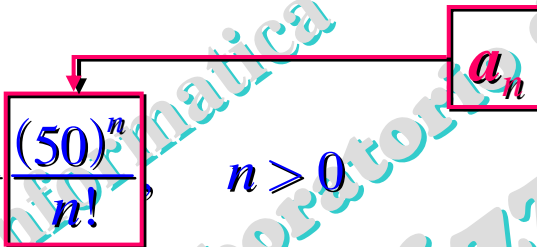
$$\varepsilon_{\text{mach}} = \min \{ 0 < \varepsilon \in F(\beta, t, E_{\min}, E_{\max}) : 1 \oplus \varepsilon > 1 \}$$

in C: FLT_EPSILON
DBL_EPSILON

Criterio di arresto naturale

Esempio 0: perché criterio di arresto naturale ?

Usando per la somma $S_n(x) = \sum_{k=0}^n \frac{x^k}{k!} \approx e^x$ l'algoritmo iterativo:

$$\begin{cases} S_0(50) = 1 \\ S_n(50) = S_{n-1}(50) + \frac{(50)^n}{n!}, \quad n > 0 \end{cases}$$


che approssima il valore e^{50} mediante alcune ridotte $S_n(50)$ della serie esponenziale, nel sistema aritmetico in singola precisione (7÷8 cifre significative decimali equivalenti), si ottiene:

$S_{88}(50)=$	5.184703e+021	0 11000111 00011001000100000011011
$S_{89}(50)=$	5.184704e+021	0 11000111 00011001000100000011101
$S_{90}(50)=$	5.184705e+021	0 11000111 00011001000100000011110
$S_{91}(50)=$	5.184705e+021	0 11000111 00011001000100000011111
$S_{92}(50)=$	5.184705e+021	0 11000111 00011001000100000011111

...e la somma non aumenta più al crescere di n pur eseguendo le addizioni!
... Perché ???

```

main()
{
    int n,k; short bit[MAX_LEN]; float x,a, A[200];
    union sp
    {
        float F;
        unsigned int N;
    } S[200];

    printf("\tFLT EPSILON = %e\n",FLT_EPSILON);
    x=50.0f; S[0].F=1.0f; A[0]=1.0f; S[1].F=a=x; n=1; A[1]=a;
    while (S[n].F != S[n-1].F)
    {
        n++;
        a=a*x/n; A[n]=a;
        S[n].F = S[n-1].F+a;
    }
    printf("n = %d\n",n);
    for (k=n-4; k<=n; k++)
    {
        printf("\na_%3d = %e\tA[k]/S[k-1] = %e",k,A[k],A[k]/S[k-1].F);
        printf("\nS(%3d, %3.2f) = %e \t",k,x,S[k].F);
        mostra_32_bit(S[k].N,bit);
    }
}

```

somma a finché S si incrementa

FLT_EPSILON = 1.192093e-007

FLT_EPSILON/2 = 5.960464e-008

n = 92

a_88 = 1.742036e+015 A[k]/S[k-1] = 3.359954e-007

S(88, 50.00) = 5.184703e+021 0 11000111 00011001000100000011011

a_89 = 9.786718e+014 A[k]/S[k-1] = 1.887614e-007

S(89, 50.00) = 5.184704e+021 0 11000111 00011001000100000011101

a_90 = 5.437066e+014 A[k]/S[k-1] = 1.048674e-007

S(90, 50.00) = 5.184705e+021 0 11000111 00011001000100000011110

a_91 = 2.987399e+014 A[k]/S[k-1] = 5.761946e-008

S(91, 50.00) = 5.184705e+021 0 11000111 00011001000100000011111

a_92 = 1.623586e+014 A[k]/S[k-1] = 3.131492e-008

S(92, 50.00) = 5.184705e+021 0 11000111 00011001000100000011111

criterio di arresto naturale

while (a >= ULP(S))

{S=S+a; a=...;}

come approssimare **ULP(S)** ?

$\frac{a_{88}}{S_{87}(50)} =$	3.3599 e-007
$\frac{a_{89}}{S_{88}(50)} =$	1.8876 e-007
$\frac{a_{90}}{S_{89}(50)} =$	1.0486 e-007
$\frac{a_{91}}{S_{90}(50)} =$	5.7619 e-008
$\frac{a_{92}}{S_{91}(50)} =$	3.1314e-008

$$\frac{|a_n|}{S_{n-1}(50)} > \text{FLT_EPSILON}/2$$

FLT_EPSILON/2
5.960464e-008

$$\frac{|a_n|}{S_{n-1}(50)} \leq \text{FLT_EPSILON}/2$$

...
while (a > S* $\epsilon_{\text{mach}}/2$)
{S=S+a;
a=...;
}

per il round to nearest

per evitare il problema, usare
il criterio di arresto naturale!

Nonostante il Sistema Aritmetico Standard IEEE 754 assicuri massima accuratezza statica e dinamica nelle singole operazioni floating-point, l'errore totale di roundoff può comunque aumentare in maniera abnorme ...

risultato di massima
accuratezza

def

$$E_R \leq \frac{1}{2} \varepsilon_{mach}$$

Errore Relativo
del risultato floating-point
rispetto a quello esatto

Esempi

Esempio 1: equazione di 2° grado

Le radici di $x^2 + 10^4 x - 1 = 0$ sono

$$x_1 = 0.00009999$$

$$x_2 = -10000.0001$$

```
format long e; roots([1 10^4 -1])'
```

MATLAB

ans =

```
-1.000000010000000e+004  9.999999900000002e-005
```

Usando la ben nota formula $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$ (instabile) si ha

???

$\tilde{x}_1 = 0.0000000000e+000$	$E_r = 1.00e+000$
$\tilde{x}_2 = -1.0000000000e+004$	$E_r = 1.00e-008$

singola precisione $\frac{1}{2} \epsilon_{mach} = 5.9e-8$

Non è di massima accuratezza

invece mediante

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

$$x_1 = \frac{c}{ax_2} \quad (\text{stabile})$$

$\tilde{x}_1 = 9.999999747e-005$	$E_r = 1.82e-008$
$\tilde{x}_2 = -1.0000000000e+004$	$E_r = 1.00e-008$

radice di massimo modulo

è di massima accuratezza

Esempio 1: equazione di 2° grado – codice C

```
1  #include <stdio.h>
2  #include <math.h>
3  #include <float.h>
4
5
6  main()
7  {
8      float a=1.f, b=1.e+4, c=-1.f, x1, x2, Er1, Er2;
9      long double A=1.0, B=1.e+4, C=-1.0, X1, X2, swap;
10
11      printf("\tFLT_EPSILON = %e\tFLT_EPSILON/2 = %e\n\n", FLT_EPSILON, FLT_EPSILON/2);
12      // Radici in long double (anche con la formula sbagliata!)
13      X2=(-B-sqrt(B*B-4.0*A*C))/(2.0*A);
14      X1=(-B+sqrt(B*B-4.0*A*C))/(2.0*A);
15      printf("\n\tRadici esatte: \n\nX1 = %24.16e,\tX2 = %24.16e\n", (double)X1, (double)X2);
16
17      // Radici in float (formula sbagliata)
18      x2=(-b-(float)sqrt(b*b-4.f*a*c))/(2.f*a);
19      x1=(-b+(float)sqrt(b*b-4.f*a*c))/(2.f*a);
20      Er1 = (float)fabs(X1-x1)/fabs(X1);
21      Er2 = (float)fabs(X2-x2)/fabs(X2);
22      printf("\n\nRadici formula sbagliata \tErrori relativi\n");
23      printf("\nx1 = %+e,\t\tEr1 = %e\nx2 = %+e,\t\tEr2 = %e\n\n", x1, Er1, x2, Er2);
24
25      // Radici in float (formula corretta)
26      if (b>0)
27          x2=(-b-(float)sqrt(b*b-4.f*a*c))/(2.f*a); // radice di massimo modulo
28      else
29      {
30          x2=(-b+(float)sqrt(b*b-4.f*a*c))/(2.f*a); // radice di massimo modulo
31          swap=X1; X1=X2; X2=swap; // solo per gli errori
32      }
33      x1= c/a/x2;
34      Er1 = (float)fabs(X1-x1)/fabs(X1);
35      Er2 = (float)fabs(X2-x2)/fabs(X2);
36      printf("\n\nRadici formula corretta \tErrori relativi\n");
37      printf("\nx1 = %+e,\t\tEr1 = %e\nx2 = %+e,\t\tEr2 = %e\n\n", x1, Er1, x2, Er2);
38  }
```


Esempio 1: equazione di 2° grado – output

FLT_EPSILON = 1.192093e-007

FLT_EPSILON/2 = 5.960464e-008

Radici esatte:

x1 = 9.9999999292776920e-005, x2 = -1.0000000099999999e+004

Radici formula sbagliata

Errori relativi

Er > FLT_EPSILON/2

x1 = +0.000000e+000,

Er1 = 1.000000e+000

x2 = -1.000000e+004,

Er2 = 1.000000e-008

Radici formula corretta

Errori relativi

x1 = +1.000000e-004,

Er1 = 1.818989e-008

x2 = -1.000000e+004,

Er2 = 1.000000e-008

risultato non di massima accuratezza

risultato di massima accuratezza ($Er \leq \text{FLT_EPSILON}/2$)

Esempio 2: non vale la Proprietà Associativa

Nel S.A. Floating-Point può succedere che

$$(a+b)+c \neq a+(b+c)$$

$$a = 2^{-24} \approx 5e-8$$

$$b = 2^{-25} \approx 3e-8$$

$$c=1$$

$$x1 = (a+b)+c = 1+2^{-23}$$

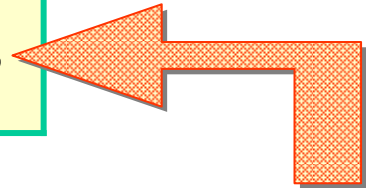
0 01111111 00000000000000000000000000000001

$$E_r = 2.98e-008$$

$$x2 = a+(b+c) = 1$$

0 01111111 00000000000000000000000000000000

$$E_r = 8.94e-008$$



Non è di massima accuratezza

singola precisione $\frac{1}{2} \epsilon_{mach} = 5.9e-8$

Quiz: perché?

Esempio 2: proprietà associativa di + – codice C

```
4  #include <stdio.h>
5  #include <math.h>
6  #include <float.h>
7  #define MAX_LEN 32
8  void mostra_32_bit(long num, short bit[32])
9  { char k;
19 main()
20 { float a, b, c, temp, x1, x2, Er1, Er2;
21   long double A, B, C, X1, X2;
22   union sp
23   { float F;
24     unsigned int N;
25   } u1, u2; short bit[MAX_LEN];

27   printf("\tFLT_EPSILON = %e\tFLT_EPSILON/2 = %e\n\n", FLT_EPSILON, FLT_EPSILON/2);
28   A=pow(2.0,-24); a=(float)A;
29   B=pow(2.0,-25); b=(float)B;
30   C=1.0;          c=(float)C;

32   // Espressioni esatte
33   X1 = (A + B) + C; X2 = A + (B + C);
34   printf("\n\tEspressioni esatte: \n\nx1 = %24.16e,\tx2 = %24.16e\n", (double)X1, (double)X2);

36   // Espressioni in float
37   temp = a+b; x1 = temp + c;
38   temp = b+c; x2 = a + temp;
39   Er1 = (float) fabs(X1-x1)/fabs(X1);
40   Er2 = (float) fabs(X2-x2)/fabs(X2);
41   printf("\n\nEspressioni float \tErrori relativi\n");
42   printf("\nx1 = %+1.8e,\tx\tEr1 = %e\nx2 = %+1.8e,\tx\tEr2 = %e\n\n", x1, Er1, x2, Er2);

44   u1.F=x1; printf("\nx1 binario = "); mostra_32_bit(u1.N,bit);
45   u2.F=x2; printf("\nx2 binario = "); mostra_32_bit(u2.N,bit);
46 }
```

forza la singola precisione!

usando invece:

$x1 = (a+b)+c;$
 $x2 = a+(b+c);$

i calcoli sono eseguiti nella ALU e gli errori ...

Esempio 2: proprietà associativa di + – output

FLT_EPSILON = 1.192093e-007

FLT_EPSILON/2 = 5.960464e-008

Espressioni esatte:

x1 = 1.0000000894069672e+000, x2 = 1.0000000894069672e+000

Espressioni float

Errori relativi

x1 = +1.00000012e+000, Er1 = 2.980232e-008

x2 = +1.00000000e+000, Er2 = 8.940696e-008

Er > FLT_EPSILON/2

x1 binario = 0 01111111 000000000000000000000001

x2 binario = 0 01111111 000000000000000000000000

$$x2 = a + (b + c)$$

$$x1 = (a + b) + c$$

risultato non di massima accuratezza

risultato di massima accuratezza ($Er \leq \text{FLT_EPSILON}/2$)

Esempio 3: valutazione di un polinomio

$$P(x) = 512x^{10} - 1280x^8 + 1120x^6 - 400x^4 + 50x^2 - 1$$

mediante *algoritmo di Horner* in s.p.

$$P(x) = (((((512x^2 - 1280)x^2 + 1120)x^2 - 400)x^2 + 50)x^2 - 1$$

$x_1 = 0.99$	$E_r[P(x_1)] = 2.4424 \text{ e}-005$
$x_2 = 1.$	$E_r[P(x_2)] = 0$

singola precisione $\frac{1}{2} \varepsilon_{mach} = 5.9\text{e}-8$

Non è di massima
accuratezza

Quiz: perché?

Esempio 4: somme di molti addendi

(dello stesso ordine di grandezza)

Somma Test $S_n = \sum_{k=1}^n a_k = \sum_{k=1}^{10^8} 10^{-6} = 100$

```
S=0;
for (k=0; k<n; k++)
    S = S + a[k];
```

k	S_k
25107766	3.1999 e+1
25107767	3.2000 e+1
...	...
100000000	3.2000 e+1

Da un certo indice in poi la somma non si incrementa più !!!

```
k = 25107765  S = 3.199999e+001
0 10000011 11111111111111111111111111111100
k = 25107766  S = 3.199999e+001
0 10000011 11111111111111111111111111111101
k = 25107767  S = 3.200000e+001
0 10000011 11111111111111111111111111111110
k = 25107768  S = 3.200000e+001
0 10000011 11111111111111111111111111111111
k = 25107769  S = 3.200000e+001
0 10000100 000000000000000000000000000000
k = 25107770  S = 3.200000e+001
0 10000100 000000000000000000000000000000
k = 100000000 S = 3.200000e+001
0 10000100 000000000000000000000000000000
```

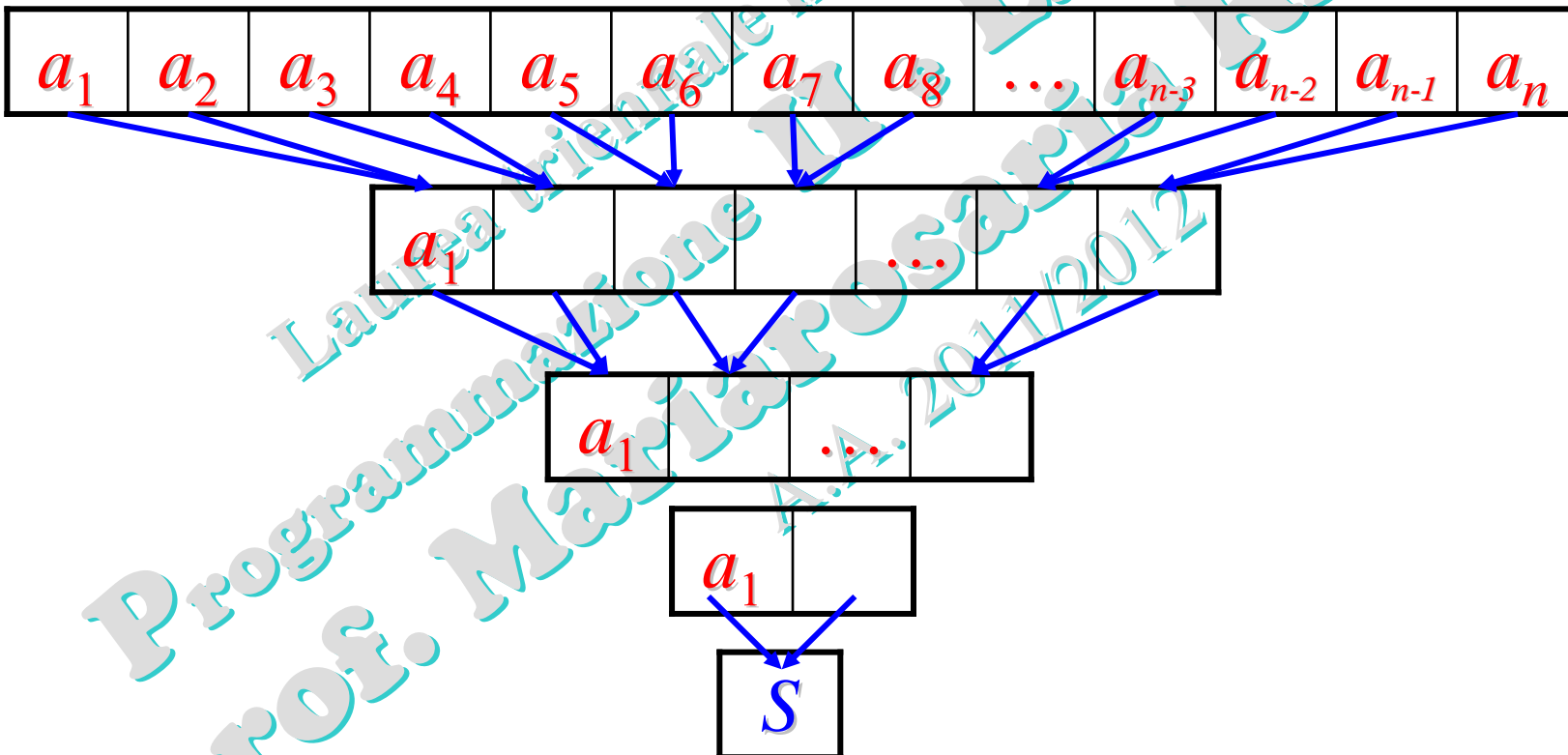
Quiz: perché?

Esempio 5: somme di molti addendi

(dello stesso ordine di grandezza)

$$S_n = \sum_{k=1}^n a_k = \sum_{k=1}^{10^8} 10^{-6} = 100$$

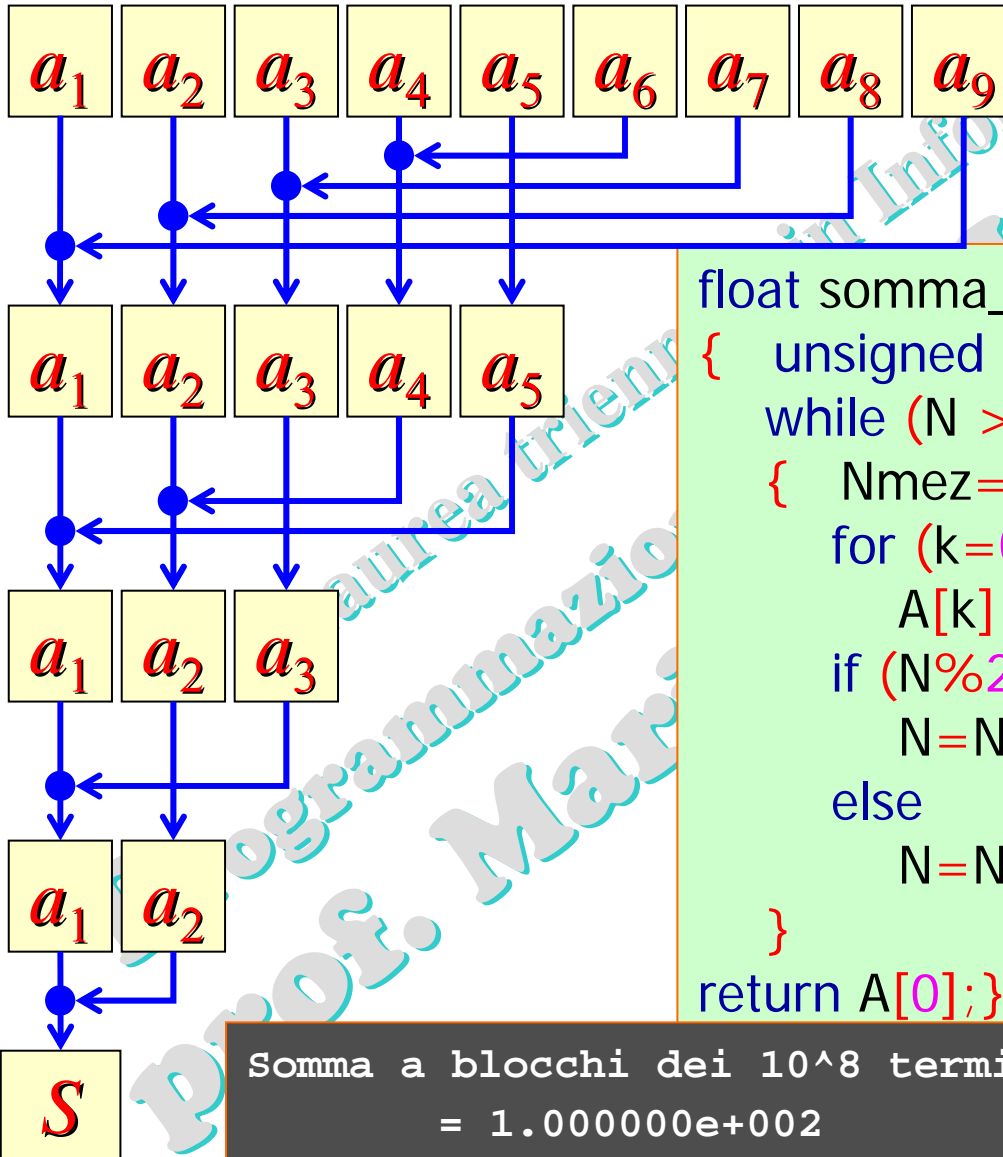
Soluzione: sommare a gruppi (*raddoppiamento ricorsivo*)



Quiz: perché così funziona?

Esempio 5: somme di molti addendi dello stesso ordine di grandezza – algoritmo

Invece di sommare le componenti adiacenti dell'array, somma quelle diametralmente opposte (simmetriche rispetto al centro)



```
float somma_blocchi(unsigned int N, float A[])
{
    unsigned int k, Nmez;
    while (N > 1)
    {
        Nmez = N/2;
        for (k=0; k<Nmez; k++)
            A[k] = A[k] + A[N-1-k];
        if (N%2 == 0)
            N = Nmez;
        else
            N = Nmez + 1;
    }
    return A[0];
}
```

Somma a blocchi dei 10^8 termini uguali a 10^{-6}
= 1.000000e+002

OK!

Esempio 6: somme di molti addendi ordinati (> 0)

$$S_n = \sum_{k=1}^n \frac{1}{k^2} \approx \frac{\pi^2}{6} \quad \left\{ \frac{1}{k^2} \right\} \text{ decrescente}$$

Gli addendi possono essere sommati in ...

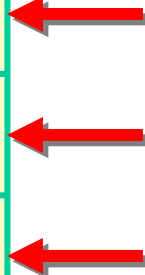
ordine **decrescente** dei valori

ordine **crescente** dei valori

$$S_n = 1 + \frac{1}{2^2} + \frac{1}{3^2} + \frac{1}{4^2} + \dots + \frac{1}{n^2}$$

Errore relativo di discretizzazione

<i>n</i>	PIÙ ACCURATO crex	decrex
5000	1.2153e-004	1.2690e-004
10000	6.0803e-005	1.2690e-004
20000	3.0365e-005	1.2690e-004



Quiz: perché?

Esempio 7: somme di addendi ordinati alternanti

$$S_n(x) = \sum_{k=0}^n \frac{x^k}{k!} \approx e^x$$

Usando l'algoritmo efficiente per calcolare $S_n(-5.5) \approx e^{-5.5}$ si ha

$$E_r = 7.26\text{e-}005$$

- accurato!

Sfruttando invece la proprietà dell'esponenziale

$$e^{-5.5} = \frac{1}{e^{+5.5}}$$

si ha

$$E_r = 1.38\text{e-}007$$

+ accurato!

Quiz: perché?

Riassumendo

- quando un algoritmo contiene un modulo per calcolare una somma di molti addendi (prodotti scalari, formule di quadratura, ...), in assenza di algoritmi specifici, conviene eseguire la computazione intermedia, se possibile, ad una precisione maggiore (ad es. usando in C il tipo `long double`);
- evitare quando possibile la cancellazione nelle somme algebriche;
- nella somma iterativa

```
while (...) { a=...; S=S+a; }
```

usare il **criterio di arresto naturale** per evitare somme inutili di addendi non significativi rispetto a **S**;

Data type constants

The constants listed below give the ranges for the **integral data types** and are defined in **LIMITS.H**.

Constant	Value	Meaning
SCHAR_MAX	127	Maximum signed char value
SCHAR_MIN	-128	Minimum signed char value
UCHAR_MAX	255 (0xff)	Maximum unsigned char value
CHAR_BIT	8	Number of bits in a char
USHRT_MAX	65535 (0xffff)	Maximum unsigned short value
SHRT_MAX	32767	Maximum (signed) short value
SHRT_MIN	-32768	Minimum (signed) short value
UINT_MAX	4294967295(0xffffffff)	Maximum unsigned int value
ULONG_MAX	4294967295(0xffffffff)	Maximum unsigned long value
INT_MAX	2147483647	Maximum (signed) int value
INT_MIN	-2147483648	Minimum (signed) int value
LONG_MAX	2147483647	Maximum (signed) long value
LONG_MIN	-2147483648	Minimum (signed) long value
CHAR_MAX	127(255 if /J option used)	Maximum char value
CHAR_MIN	-128(0 if /J option used)	Minimum char value
MB_LEN_MAX	2	Maximum number of bytes in multibyte char

Data type constants

The constants listed below give the ranges and other characteristics of the **float data type** and are defined in **FLOAT.H**.

Constant	Value	Meaning
FLT_DIG	6	Number of decimal digits of precision
FLT_EPSILON	1.192092896e-07F	Smallest such that $1.0 + \text{FLT_EPSILON} \neq 1.0$
FLT_MANT_DIG	24	Number of bits in mantissa
FLT_MAX	3.402823466e+38F	Maximum value
FLT_MAX_10_EXP	38	Maximum decimal exponent
FLT_MAX_EXP	128	Maximum binary exponent
FLT_MIN	1.175494351e-38F	Minimum positive value
FLT_MIN_10_EXP	-37	Minimum decimal exponent
FLT_MIN_EXP	-125	Minimum binary exponent
FLT_RADIX	2	Exponent radix
FLT_ROUNDS	1	Addition rounding: near

Data type constants

The constants listed below give the ranges and other characteristics of the **double data type** and are defined in **FLOAT.H**.

Constant	Value	Meaning
DBL_DIG	15	# of decimal digits of precision
DBL_EPSILON	2.2204460492503131e-016	Smallest such that $1.0 + \text{DBL_EPSILON} \neq 1.0$
DBL_MANT_DIG	53	# of bits in mantissa
DBL_MAX	1.7976931348623158e+308	Maximum value
DBL_MAX_10_EXP	308	Maximum decimal exponent
DBL_MAX_EXP	1024	Maximum binary exponent
DBL_MIN	2.2250738585072014e-308	Minimum positive value
DBL_MIN_10_EXP	-307	Minimum decimal exponent
DBL_MIN_EXP	-1021	Minimum binary exponent
_DBL_RADIX	2	Exponent radix
_DBL_ROUNDS	1	Addition rounding: near

Esercizi: visualizzare gli errori di roundoff delle versioni "naive" dei programmi e, dove possibile, generare il risultato corretto motivando le scelte effettuate.

- 1 Scrivere *function C* per valutare un polinomio mediante *algoritmo di Horner* (se non è noto cercarlo sul web). [liv. 2]
- 2 Scrivere *function C* per calcolare una somma di molti addendi dello stesso ordine di grandezza. A scelta la versione dell'algoritmo di somma a gruppi.
- 3 Scrivere *function C* di somma iterativa con criterio di arresto naturale.
- 4 Scrivere *function C* per sommare addendi ordinati rispettivamente in ordine crescente e decrescente.
- 5 Scrivere *function C* che somma addendi di segno alternato evitando l'eventuale cancellazione catastrofica. [liv. 2]