Lezioni di Calcolo Numerico
Lezione 01: Aritmetica, errori

Alberto Tibaldi

5 maggio 2018

Indice

Introduzione

Quando ero uno studente, non riuscivo a prendere molto sul serio la parte di corso relativa all'aritmetica del calcolatore. Se infatti da un lato si trattava di un argomento un po' noioso, l'ambiente che mi circondava sembrava suggerire che fosse anche piuttosto inutile. Pensando ai vari esami di Fisica o Ingegneria, infatti, il testo chiedeva di riportare risultati *con due-tre cifre dopo la virgola*. Per vari anni, questo fatto non sembrava essere cambiato: aldilà di esperimenti di interesse puramente accademico (forse al limite della speculazione), nella maggior parte dei casi risultati accurati *alla terza cifra decimale* sono più che soddisfacenti. Alla luce di questo, ho avuto la tentazione di chiedermi

> «Ma che senso ha farsi tanti problemi su quante cifre il computer usi per fare i conti, se tanto a me interessano solo le prime due-tre?!?»

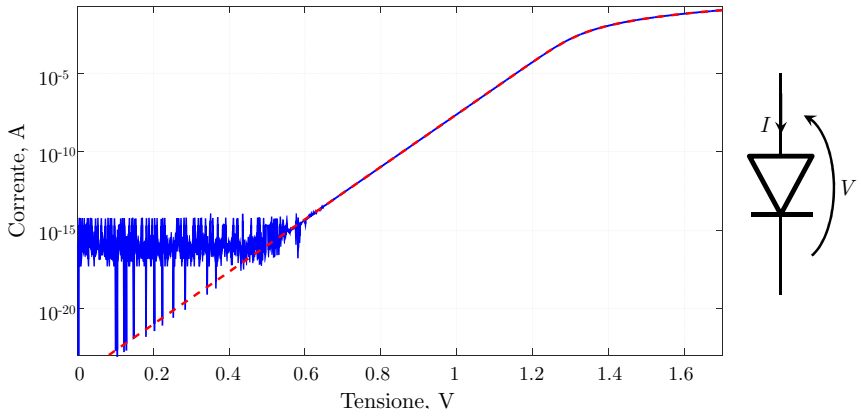

Figura 1: Caratteristica corrente-tensione $I(V)$ di un diodo a semiconduttore in scala logaritmica. Le curve blu e rossa tratteggiata sono state ottenute con simulatori programmati per operare a *doppia* e *quadrupla precisione macchina*, rispettivamente.

Purtroppo, questa tentazione contiene un gigantesco errore di fondo. Generalmente, un problema fisico/ingegneristico richiede uno *svolgimento* composto da diversi calcoli, del cui *risultato* ci interessano solo alcune cifre. La vera domanda da porsi dunque è:

> «Chi ci dice che, per avere un *risultato* accurato alla terza cifra dopo la virgola, sia sufficiente *svolgere i vari passaggi* con la stessa accuratezza?»

La risposta è: **nessuno, perché non è così.**

Credo che una persona debba sbattere -violentemente- su un problema prima di riuscire a riconoscerlo come tale. Per questo, prima di passare alle questioni *serie*, vorrei raccontare una mia esperienza risalente al luglio 2016. Talvolta, capita di dover utilizzare un diodo a semiconduttore con correnti molto ridotte. Questo per esempio può essere il caso di un fotorivelatore, un dispositivo elettronico atto a *misurare la luce* trasformandola in una corrente; la corrente *in condizioni di buio* del dispositivo deve essere molto piccola, in modo da rendere quella *in presenza di luce* più semplice da misurare. In questo contesto, il mio obiettivo era sviluppare un software in grado di simulare un dispositivo di questo tipo, in presenza di correnti molto basse. Al termine del lavoro, il risultato fu la curva blu di Fig. 1. Dopo aver perso **settimane** cercando un purtroppo inesistente errore di programmazione, mi resi conto che il problema poteva essere legato all'accuratezza con cui il computer *svolgeva i passaggi intermedi* finalizzati a ottenere il risultato. Nella stragrande maggioranza dei casi, la doppia precisione (tipica delle architetture a 64 bit, che corrisponde a circa 16 cifre decimali significative) è più che sufficiente per ogni genere di calcolo; tuttavia, esistono situazioni in cui è necessario lavorare con un'aritmetica ancora più precisa. Capito questo e prese le opportune contromisure, in particolare *aumentando l'accuratezza dei calcoli intermedi*, il problema è stato completamente risolto, permettendomi di ottenere la curva rossa tratteggiata di Fig. 1, assolutamente realistica e ragionevole in quanto priva di rumore.

Concludo questo breve racconto di piccole[1] disgrazie, sperando che possa servire da monito al lettore, o quantomeno stimolare un po' di curiosità nei confronti delle note che seguiranno nel resto del capitolo.

1 Rappresentazione floating-point

Il primo argomento che affronteremo riguarda la **rappresentazione floating-point**, detta anche **rappresentazione a virgola mobile**[2]. Il nome *rappresentazione* deve far pensare a un *modo di scrivere qualcosa*. In particolare, questa rappresentazione è finalizzata ad avvicinarsi al modo di operare di un computer. Un numero reale $a \in \mathbb{R}$ si può scrivere, nella notazione floating-point, come

$$a = (-1)^s \, p \, N^q, \qquad N \in \mathbb{N}, \qquad p \in \mathbb{R}, \qquad p \geq 0, \qquad q \in \mathbb{Z}, \qquad s \in \{0, 1\} \tag{1}$$

ovvero, N è un numero naturale (intero positivo), p è reale positivo, q è intero, e s può assumere i valori 0 o 1. Il numero N è detto **base** del sistema di numerazione. Per questioni anatomiche[3], gli esseri umani sono soliti fare i conti con $N = 10$, ovvero con il cosiddetto **sistema decimale**. Diversamente, i computer effettuano conti utilizzando il **sistema binario**, ovvero $N = 2$, per via della semplicità nel trattare una logica contenente solo due possibili valori; poi, il risultato delle operazioni ci viene presentato in sistema decimale tramite un *cambio di base* effettuato a posteriori del calcolo.

Per come è stata definita in (1), la rappresentazione **non è univoca**, ovvero, a parità di sistema di numerazione scelto (e quindi di base), è possibile rappresentare lo stesso numero reale a utilizzando diversi valori di p e q. Per esempio, volendo scrivere il numero 125,54, questo può essere scritto tanto come $1,2554 \times 10^2$ ($p = 1,2554$, $q = 2$), quanto come 12554 ($p = 12554$, $q = 0$). **D'ora in poi, in questo testo, si farà uso della notazione americana, per cui la demarcazione tra interi e decimali viene indicata con un punto**. Questa scelta è legata al fatto che il codice MATLAB®, nonché la stragrande maggior parte dei linguaggi di programmazione, usa questa convenzione.

[1] almeno, rispetto alle tragedie descritte in http://ta.twi.tudelft.nl/users/vuik/wi211/disasters.html

[2] nella letteratura inglese/americana, si utilizza *point* per indicare il simbolo di separazione tra unità e decimali, poiché in questi contesti viene utilizzato un punto, mentre la virgola (facoltativamente) viene utilizzata per indicare le migliaia. Al contrario, nella letteratura italiana, la virgola indica l'inizio dei decimali, e (facoltativamente) il punto indica le migliaia. Per esempio, volendo scrivere *duemilacinquecentoventicinque virgola cinquecentosessantasette* in inglese e italiano, si avrebbero, rispettivamente, 2,525.567 e 2.525,567

[3] le mani, strumento di calcolo di base, hanno dieci dita ;-)

Al fine di rendere univoca la rappresentazione, ovvero di poter scrivere il numero reale a in un **unico** modo, si deve fissare una convenzione, un vincolo; in particolare, la nostra scelta sarà di avere sempre e comunque

$$N^{-1} \leq p < 1, \tag{2}$$

ovvero, la prima cifra di p è sempre diversa da zero, ma allo stesso tempo p deve essere minore di 1. Assieme a (1), il vincolo (2) permette di definire la **rappresentazione floating-point normalizzata**. Fissata la convenzione (2), il numero p viene detto **mantissa**, mentre q viene detto **esponente** o **caratteristica**; infine s è il **segno** del numero (poiché se $s = 0$ il segno del numero è positivo, se $s = 1$ sarà negativo). Disporre di questa rappresentazione significa che il numero reale a è **univocamente** determinato, nella base N, dalla conoscenza dei tre numeri s, p e q. Tuttavia,

> «Per quale motivo stiamo decidendo di complicarci la vita in questo modo? Cioè, invece di dover lavorare con un singolo numero, stiamo scegliendo di lavorare con tre numeri!»

Per rispondere a questa domanda, cerchiamo di ricordare prima di tutto qual è il nostro scopo: **lavorare con un computer**. Considerando un generico numero reale a, per esempio π o $\sqrt{2}$, questo andrebbe scritto, *esattamente*, tenendo conto di infinite cifre. Volendo lavorare con infinite cifre, sarebbe necessario, a un certo punto, memorizzarlo, e per far questo servirebbe *infinita memoria*, ancora prima di iniziare ad agire su di esso: non avrebbe molto senso. In questa direzione, la rappresentazione floating-point nasce da un compromesso tra voler descrivere un numero tenendo conto di una certa accuratezza, ovvero *usando una certa quantità di cifre*, e non volersi precludere la possibilità di trattare numeri molto grandi o molto piccoli. Immaginiamo per esempio di dover fare dei conti su un condensatore. L'unità di misura della capacità di un condensatore è il farad (F) ma, in microelettronica, valori tipici sono dell'ordine dei picofarad, quindi 10^{-12} F. Avrebbe senso scrivere 3.839 pF come 0.000000000003839 F, ovvero, salvare undici zeri dopo la virgola e poi la cifre utili, o salvare *esclusivamente* le cifre *utili* e *quanti zeri ci sono*? Per esempio, questo stesso numero, in notazione floating-point normalizzata, si potrebbe scrivere come

$$0.3839 \times 10^{-11},$$

ovvero, $s = 0$, $p = 0.3839$, $q = -11$. La stessa situazione potrebbe verificarsi in chimica, volendo quantificare il numero di molecole in una mole (6.023×10^{23}): dovremmo scrivere (e memorizzare) venti zeri?! Disporre dell'esponente q, quindi, ci permette di rappresentare numeri molto grandi o molto piccoli con semplicità, evitando di salvare inutili zeri.

2 Numeri di macchina, errori

2.1 L'insieme dei numeri di macchina

Un computer può lavorare con un numero finito di cifre: sia per quanto riguarda la mantissa, sia per quanto riguarda l'esponente. Tuttavia, è raro trovare problemi che non coinvolgano numeri reali; si pensi anche soltanto al calcolo dell'area di un cerchio di raggio r: $A = \pi r^2$. Per questo motivo, ha senso lavorare sui cosiddetti **numeri di macchina**, ovvero numeri rappresentabili **esattamente** da un calcolatore. Questo insieme è definito come:

$$\mathcal{F} = \{0\} \bigcup \{ \underbrace{(-1)^s}_{\text{segno}} \times \underbrace{(0.a_1a_2a_3...a_t)}_{\text{mantissa}} \times \underbrace{N^q}_{\text{esponente}} \quad , \ 0 \leq a_i < N, \ \ a_1 \neq 0, \ \ L \leq q \leq U \}. \tag{3}$$

Descriviamo a parole la definizione (3). Qui, rivediamo le definizioni di segno, mantissa ed esponente. Un primo dettaglio molto importante riguarda a_t: come si può vedere, la mantissa è sempre costituita da t cifre a_i (da a_1 a a_t), poiché il computer viene progettato per operare solo con esse. Seguono nella definizione le condizioni di normalizzazione, tali per cui ciascuna cifra della mantissa deve essere maggiore o uguale di 0, e strettamente minore della base[4]; come già detto si richiede, per rendere univoca la rappresentazione, che la prima cifra della mantissa, a_1, sia diversa da zero. Infine, esistono limiti anche per quanto riguarda il numero di cifre dell'**esponente**

[4] per esempio, per $N = 10$, una cifra di un numero può essere al minimo 0, e al massimo 9

che si possono memorizzare. In particolare, L e U rappresentano il minimo e il massimo valore che un esponente possa assumere. Detto in altre parole, l'aritmetica con cui si intende lavorare è determinata una volta fissati N, t, L e U.

Una conseguenza di lavorare unicamente con t cifre di mantissa e con un intervallo finito di esponenti è la **finitezza** dell'insieme dei numeri di macchina $\mathcal{F}$. Infatti, né un generico numero razionale né tantomeno un generico numero reale è un numero macchina. In altre parole,

$$\mathcal{F} \subset \mathbb{Q} \subset \mathbb{R}.$$

La nostra incapacità di lavorare con un numero arbitrario, possibilmente infinito, di cifre, ci porta a dover considerare l'insieme dei numeri di macchina definito in (3) come l'unica piattaforma ragionevole per i nostri scopi. Di conseguenza, un numero che non appartiene già a questo insieme dovrà essere **approssimato** in qualche modo, e dietro a ogni approssimazione abbiamo degli **errori** che, in determinate situazioni, possono compromettere gravemente il risultato delle nostre operazioni. Nel seguito del testo cercheremo di capire più precisamente quali siano le cause nascoste dietro a questi errori, quali siano le situazioni che possono degenerare, e come cercare di porre rimedio a questi problemi.

2.2 Errori legati alla *grandezza* del numero

Al fine di identificare la natura del primo tipo di errore, poniamoci due domande:

1. Qual è, in una data aritmetica, il numero rappresentabile più piccolo?
2. Qual è, in una data aritmetica, il numero rappresentabile più grande?

Queste domande possono essere poste in modo lievemente diverso: per *numero rappresentabile*, infatti, è corretto intendere *numero di macchina*. In particolare, il più piccolo numero di macchina m è

$$m = 0.1 \times N^L.$$

Infatti, il minimo valore che la mantissa possa avere è $p = 0.1$, poiché la prima cifra dopo la virgola deve essere diversa da 0, ma quindi al minimo può valere 1. Infine, come già scritto, L è il minimo esponente accettabile nell'aritmetica macchina. Per quanto riguarda il numero macchina M più grande che si possa rappresentare, questo avrà tutte le cifre della mantissa pari a $N-1$, ed esponente pari a U. Per esempio, con $t = 5$, $N = 10$,

$$M = 0.\underbrace{99999}_{t=5} \times 10^U.$$

L'interesse per il massimo e il minimo numero rappresentabili in una data aritmetica è legato alla possibilità che il numero a di partenza sia *troppo piccolo* o *troppo grande* per essere rappresentato. In particolare,

- per $a \in (-m, m)$, di fatto si dice che $a = 0$ (regione di **underflow**);
- per $a \in (-\infty, -M) \bigcup (+M, +\infty)$, si dice che $a = \pm\infty$ (regione di **overflow**).

Non è molto difficile cadere in una di queste situazioni. Consideriamo per esempio di voler calcolare il numero a definito come

$$a = \frac{\overline{M} + M}{2},$$

dove si è implicitamente definito

$$\overline{M} = \frac{M}{2},$$

e M è il massimo numero di macchina. Calcolare mediante un computer il valore di a non è banale: se infatti prima si calcola la somma di $\overline{M}$ e M, il risultato parziale sarà più grande del

massimo numero rappresentabile M, e quindi, per il computer, uguale a ∞; a questo punto, $\infty/2 = \infty$. Se invece calcolo l'espressione con il passaggio

$$a = \frac{\overline{M}}{2} + \frac{M}{2},$$

ossia prima dimezzando i singoli addendi e poi sommandoli, l'operazione si può effettuare. Infatti, in questo modo, si avrebbe:

$$a = \frac{\overline{M}}{2} + \frac{M}{2} = \frac{M}{4} + \frac{M}{2} = \frac{3}{4}M,$$

dal momento che nessuno dei *passaggi intermedi*[5] risulta essere affetto da fenomeni di overflow.

Un esempio un po' subdolo

Per concludere questo argomento, si propone ora un esempio un po' più cattivello; sarà possibile capirlo più a fondo al termine di questa sezione, quando si parlerà esattamente delle potenzialità di MATLAB®. Segue il testo del problema.
Dati $x_1 = 10^{10}$, $x_2 = 10^{170}$, calcolare mediante MATLAB®la quantità

$$R = \sqrt{x_1^2 + x_2^2}.$$

Viene riportato ora un breve script implementante due soluzioni, R_1 e R_2.

```
clear
close all
clc

x1=1e10; % notazione esponenziale indicante 10^(10)
x2=1e170; % notazione esponenziale indicante 10^(170)

R1 = sqrt(x1^2 + x2^2) % soluzione errata
R2 = abs(x2)*sqrt(1 + (x1/x2)^2) % soluzione corretta
```

Provando questo script su MATLAB®, si può notare che una delle soluzioni produce un overflow (si capisce dal risultato infinito), l'altra no. Infatti, dal momento che x_2 è un numero molto grande, elevandolo al quadrato diviene ancora più grande e, quindi, genera un overflow. Se al contrario si raccoglie x_2^2 e lo si porta fuori dalla radice, usando il trucco

$$R = \sqrt{x_1^2 + x_2^2} = \sqrt{x_2^2\left(\frac{x_1^2}{x_2^2} + 1\right)} = |x_2|\sqrt{1 + \left(\frac{x_1}{x_2}\right)^2}$$

non si verificherà più alcun fenomeno di overflow. Ovviamente, questa soluzione è valida perché $x_2 \gg x_1$; nel caso opposto, sarebbe stato necessario raccogliere x_1^2 dentro al segno di radice, e procedere di conseguenza.

2.3 Errori legati al numero di cifre della mantissa; rounding to even

Come già accennato in precedenza, un generico numero reale non è un numero macchina, in quanto servirebbe un computer a precisione t infinita. Tuttavia, se volessimo per esempio introdurre in un calcolo il numero $\sqrt{2}$, come potremmo fare? La soluzione è **approssimare** $\sqrt{2}$ al numero macchina *più vicino*, *più simile*. Per poter effettuare questa approssimazione, occorre quindi stabilire un processo di approssimazione, ovvero un metodo in grado di associare a un numero reale $a \in \mathbb{R}$ un numero macchina $\overline{a} \in \mathcal{F}$. In particolare, a partire dalla rappresentazione floating-point normalizzata di a,

$$a = (-1)^s\, p\, N^q,$$

il numero macchina $\overline{a}$ che lo approssima avrà una rappresentazione floating-point del tipo

$$\overline{a} = (-1)^s\, \overline{p}\, N^q.$$

[5] quelli di cui parlavo nell'introduzione ;-)

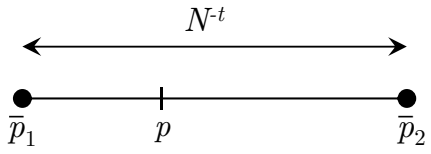

Figura 2: Illustrazione della mantissa p associata al numero reale a, e delle mantisse dei due numeri macchina più vicine a p, $\overline{p}_1$ e $\overline{p}_2$ Nella figura si evidenzia che $\overline{p}_2 - \overline{p}_1 = N^{-t}$.

Si noti che questa sezione non ci concentriamo sugli esponenti, poiché gli unici problemi che potrebbero riguardarli sono underflow e overflow, già trattati in precedenza.

La Fig. 2 mostra che la mantissa p del numero reale a è compresa tra le mantisse di due numeri macchina $\overline{p}_1, \overline{p}_2$, in cui si assume $\overline{p}_1 > \overline{p}_2$. La mantissa $\overline{p}$ è scelta tra una tra $\overline{p}_1$ e $\overline{p}_2$.

Dobbiamo a questo punto capire due cose: quale sia la mantissa di macchina più opportuna, e qual è l'errore che commettiamo approssimando p con essa.

Innanzitutto, si osservi che

$$\overline{p}_2 = \overline{p}_1 + N^{-t},$$

ovvero, per passare da una mantissa di macchina alla successiva, è necessario aggiungere N^{-t}. Questo concetto, illustrato in Fig. 2, può essere chiarito mediante un esempio. Si consideri un calcolatore con $t = 5$, $N = 10$. Si consideri $\overline{p}_1 = 0.85472$. Per calcolare $\overline{p}_2$, dovremo scrivere $\overline{p}_2 = \overline{p}_1 + 10^{-5}$, ossia

$$\begin{array}{r} 0.8\ 5\ 4\ 7\ 2 \\ +\ 0.0\ 0\ 0\ 0\ 1 \\ \hline 0.8\ 5\ 4\ 7\ 3. \end{array}$$

In altre parole, si aggiunge un'unità alla cifra meno significativa della mantissa $\overline{p}_1$.

La procedura di approssimazione che si utilizza nei calcolatori è il **rounding to even**, ovvero, **arrotondamento a pari**. Questa tecnica si può riassumere nei seguenti passi.

1. Nel caso la mantissa p sia equidistante alle due più vicine, $\overline{p}_1$ e $\overline{p}_2$, essa va approssimata alla mantissa $\overline{p}$ con ultima cifra pari.
2. Altrimenti,
   (a) si somma a p il termine $\frac{1}{2}N^{-t}$
   (b) si tronca il risultato della precedente operazione alla t-esima cifra.

Al fine di chiarire questo algoritmo, verranno ora proposti alcuni esempi.

Esempio 1

Si consideri un calcolatore operante in sistema decimale con tre cifre. Quindi, sia $p = 0.15814$. Arrotondarlo secondo l'algoritmo rounding to even.

Prima di tutto, $N = 10$, $t = 3$. Di conseguenza, le due mantisse di macchina più vicine sono $\overline{p}_1 = 0.158$ e $\overline{p}_2 = 0.159$. Inoltre, si può capire facilmente che p non sia equidistante da esse, quindi si deve passare alla parte (2) dell'algoritmo. Si consideri quindi

$$p + \frac{1}{2}N^{-t} = 0.15814 + \frac{1}{2}10^{-3},$$

che produce

$$\begin{array}{r} 0.1\ 5\ 8\ 1\ 4 \\ +\ 0.0\ 0\ 0\ 5 \\ \hline 0.1\ 5\ 8\ 6\ 4. \end{array}$$

A questo punto, si tronchi 0.15864 alle prime $t = 3$ cifre, ottenendo $\overline{p} = 0.158$.

Esempio 2

Si consideri un calcolatore operante in sistema decimale con tre cifre. Quindi, sia $p = 0.1585432$. Arrotondarlo secondo l'algoritmo rounding to even.

Di nuovo, $N = 10$, $t = 3$. Di conseguenza, le due mantisse di macchina più vicine sono $\overline{p}_1 = 0.158$ e $\overline{p}_2 = 0.159$. Inoltre, si può capire facilmente che p non sia equidistante da esse, quindi si deve passare alla parte (2) dell'algoritmo. Si consideri quindi

$$p + \frac{1}{2}N^{-t} = 0.1585432 + \frac{1}{2}10^{-3},$$

che produce

$$\begin{array}{r} \overset{1}{} \\ 0.1\,5\,8\,5\,4\,3\,2 \\ +\ 0.0\,0\,0\,5 \\ \hline 0.1\,5\,9\,0\,4\,3\,2. \end{array}$$

A questo punto, si tronchi 0.1590432 alle prime $t = 3$ cifre, ottenendo $\overline{p} = 0.159$.

Esempio 3

Si consideri un calcolatore operante in sistema decimale con tre cifre. Quindi, sia $p = 0.1585$. Arrotondarlo secondo l'algoritmo rounding to even.

In questo caso, p è equidistante da $\overline{p}_1 = 0.158$ e $\overline{p}_2 = 0.159$. Di conseguenza, la mantissa di macchina è quella più vicina e che termina con una cifra pari, ovvero $\overline{p} = 0.158$.

2.4 Errore assoluto, errore relativo

Dato un numero $x \in \mathbb{R}$, sia $\widetilde{x}$ il numero associato a x tramite la procedura descritta nella sezione precedente. Una volta memorizzato $\widetilde{x}$ e poi lavorando con esso, commettiamo un errore, ovvero introduciamo una differenza tra ciò che avremmo usando x e ciò che abbiamo usando $\widetilde{x}$. Esistono sostanzialmente due definizioni di errore:

- Si definisce **errore assoluto** e_a la grandezza

$$e_\mathrm{a} = |x - \widetilde{x}|.$$

- Si definisce **errore relativo** e_r la grandezza

$$e_\mathrm{r} = \frac{|x - \widetilde{x}|}{|x|}, \quad x \neq 0.$$

In entrambi i casi, queste definizioni permettono di quantificare la differenza tra *la verità* e ciò che otteniamo dalla nostra approssimazione. L'errore relativo, generalmente, è una quantità più significativa, perché non solo dice *quanto si sbaglia*, ma anche rispetto all'entità del numero. Volendo fare un esempio un po' stupido, si immagini di sbagliare una domanda in un esame contenente trenta domande, e una domanda in un esame contenente tre domande. In entrambi i casi, si sbaglia una sola domanda. Ma il voto finale in un caso sarà 29, nell'altro sarà 18. :-D

È possibile stimare l'errore che si commette nel momento in cui si effettua un arrotondamento. In particolare, quando si parla di *stima*, si intende cercare una *maggiorazione*, ossia un numero che ci permetta di capire *al peggio* quanto possa valere questo errore. Dati un numero reale a e il suo arrotondamento $\overline{a}$, si è detto che

$$a = (-1)^s\, p\, N^q$$
$$\overline{a} = (-1)^s\, \overline{p}\, N^q.$$

Quindi, si può calcolare l'errore assoluto come

$$e_a = |a - \overline{a}| = |(-1)^s pN^q - (-1)^s \overline{p} N^q| = |(-1)^s N^q (p - \overline{p})| = \\ = N^q |p - \overline{p}|.$$

Maggiorare l'errore assoluto sulla mantissa è, in questo caso, molto semplice! Infatti, ricordandoci la Fig. 2, sappiamo che, nel caso peggiore, p può essere esattamente al centro tra $\overline{p}_1$ e $\overline{p}_2$; in questo caso, sapendo che $\overline{p}_2 - \overline{p}_1 = N^{-t}$, l'errore *di caso peggiore* è $\frac{1}{2}N^{-t}$. Di conseguenza, abbiamo appena capito che

$$|p - \overline{p}| \leq \frac{1}{2} N^{-t}.$$

Quindi, considerando l'errore assoluto sull'intero numero a,

$$|a - \overline{a}| = N^q |p - \overline{p}| \leq N^q N^{-t} \frac{1}{2} = \frac{1}{2} N^{q-t}.$$

Per quanto riguarda l'errore relativo,

$$e_r = \frac{|a - \overline{a}|}{|a|} = \frac{N^q |p - \overline{p}|}{N^q |p|}.$$

Tuttavia, è possibile affermare, per via della normalizzazione che permette di rendere univoca la rappresentazione floating point (2), che

$$|p| \geq N^{-1},$$

e quindi, sostituendo al denominatore $|p|$ il suo minorante N^{-1}, si ottiene un'ulteriore maggiorazione dell'errore relativo, scrivibile in modo compatto:

$$\frac{|p - \overline{p}|}{|p|} \leq \frac{\frac{1}{2}N^{-t}}{N^{-1}} = \frac{1}{2} N^{1-t} \triangleq \varepsilon_m.$$

Il termine ε_m che abbiamo definito nella riga precedente è detto *precisione di macchina*, ed è la stima che stavamo cercando per quanto riguarda l'errore che si commette ogni volta che si effettua un arrotondamento. Infatti, abbiamo appena dimostrato che

$$\frac{|a - \overline{a}|}{|a|} = \frac{|p - \overline{p}|}{|p|} \leq \varepsilon_m.$$

L'importanza di questa formula è assoluta. Infatti, essa ci insegna che, nel momento in cui arrotondiamo un numero, al massimo l'errore che commettiamo è pari a ε_m; in altre parole, abbiamo la garanzia che non sia possibile un errore superiore!

2.5 Alcune note su MATLAB

È importante a questo punto sapere quali siano questi parametri per quanto riguarda MATLAB®, software di Calcolo Numerico impiegato in queste lezioni. MATLAB®lavora con lo standard IEEE754 binario in doppia precisione. Parlando di sistema binario dunque $N = 2$, si parla di bit. Lo standard prevede:

- 1 bit di segno,
- 52 bit di mantissa,
- 11 bit di esponente.

Per quanto riguarda la precisione di macchina, possiamo calcolare

$$\varepsilon_m = \frac{1}{2} N^{1-t} = \frac{1}{2} 2^{1-52} \simeq 2.204 \times 10^{-16}.$$

In effetti, provando a scrivere il comando `eps` sulla console di MATLAB®, scopriremmo che...

```
>> eps
ans =
   2.2204e-16
>>
```

... tutto sommato quello che stiamo dicendo ha un suo perché. ;-) Volendo ragionare in base 10, ciò che abbiamo appena mostrato ci dice che MATLAB® è in grado di lavorare con circa 16 cifre decimali.

Per quanto riguarda gli esponenti, i numeri massimi e i numeri minimi, si può scoprire[6] che $L = -1021$, $U = +1024$, e quindi che:

$$M = \left(1 - 2^{-52}\right) \times 2^{1024} = \left(2 - 2^{-51}\right) \times 2^{1023} \simeq 1.7977 \times 10^{308}$$
$$m = 2^{-1} \times 2^{-1021} = 2.2251 \times 10^{-308}.$$

In particolare, se proviamo a lanciare i comandi `realmax` e `realmin` sulla console di MATLAB®, scopriamo che la teoria appena sviluppata è effettivamente consistente.

```
>> realmax
ans =
  1.7977e+308
>> realmin
ans =
  2.2251e-308
>>
```

Questo dimostra che MATLAB® tratta come infiniti i numeri più grandi di $M(1+\varepsilon_{\mathrm{m}})$. Tuttavia, per quanto riguarda il minimo reale rappresentabile m, vengono effettuate delle rinormalizzazioni che permettono di rappresentare numeri ancora minori.

Riferimenti bibliografici

[1] A. Quarteroni e F. Saleri, "Introduzione al calcolo scientifico," 3ª edizione, Springer-Verlag Italia, Milano, 2006.

[6] maggiori dettagli possono essere per esempio trovati in [1, pag. 3]