Capitolo 1

Analisi degli errori

1.1 Rappresentazione dei numeri

L'utilizzo in modo corretto del calcolatore per fare calcoli di tipo scientifico, richiede la conoscenza di come sono rappresentati i numeri e degli errori che derivano da questa rappresentazione. L'uso dei numeri reali richiede una attenzione particolare, essendo questi infiniti, mentre il calcolatore ci da la possibilità di rappresentarne solo un numero finito.

La nostra notazione per rappresentare i numeri è una notazione posizionale a base 10. Ciò significa che se scriviamo 123 intendiamo esprimere il numero:

$$1 \cdot 10^2 + 2 \cdot 10^1 + 3 \cdot 10^0.$$

Cioè la cifra che occupa la posizione più a destra deve essere moltiplicata per 10^0 $(= 1)$, quella che occupa la posizione centrale per 10, ecc. Questa convenzione è ormai universalmente adottata, al contrario di altre "convenzioni" che quotidianamente usiamo per la comunicazione di nostre idee ed esperienze come, ad esempio, le lingue, gli alfabeti, le direzioni di scrittura, la musica, ecc.

Proprio per la sua universale adozione, la notazione posizionale a base 10 può apparire a molti come l'unica possibile. Ciò non è vero. Anzi, dal punto di vista storico risulta che tale rappresentazione si è affermata solo da pochi secoli. In precedenza erano state usate tante altre rappresentazioni, alcune in base 10 ma non posizionali (ad esempio quella romana), altre posizionali ma

non a base 10 (ad esempio la sumerica). Utilizzando la notazione posizionale possiamo esprimere un numero reale nella forma:

$$\pm\, \alpha_p\alpha_{p-1}\ldots\alpha_0.\beta_1\beta_2\ldots\beta_q\ldots \tag{1.1}$$

sottointendendo:

$$\pm\alpha_p\cdot 10^p+\ldots+\alpha_0\cdot 10^0+\beta_1\cdot 10^{-1}+\ldots+\beta_q\cdot 10^{-q}\ldots$$

I coefficienti α_i e β_i sono uno dei seguenti simboli: 0, 1, 2, ..., 9. Nulla vieta di usare al posto di 10 un altro numero, diciamo N. In tale caso la stringa (intesa come successione ordinata di simboli) descritta dalla (1.1) assumerebbe un altro significato. Infatti, supposto che i simboli α e β siano compresi tra 0 ed $N-1$, essa sarebbe equivalente a:

$$\pm\alpha_p\cdot N^p+\ldots+\alpha_0\cdot N^0+\beta_1\cdot N^{-1}+\ldots+\beta_q\cdot N^{-q}\ldots$$

Si ottiene così la rappresentazione posizionale di un numero in base N.

Nel campo scientifico si preferisce usare una notazione teoricamente equivalente alla ((1.1)), cioè[1]:

$$\pm\, \gamma_0.\gamma_1\gamma_2\ldots\gamma_t\ldots\cdot N^q \tag{1.2}$$

con $\gamma_0\neq 0$, e $0\leq\gamma_i\leq N-1$. Quindi il numero $3.57\cdot 10^2$ è espresso in notazione esponenziale normalizzata, mentre i numeri $73.54\cdot 10^2$ o $0.57\cdot 10^{-3}$ non lo sono. Con questa rappresentazione si possono rappresentare tutti i numeri eccetto lo zero. Essa non è univoca. Ad esempio in base 10 i due numeri:

1.200.....
1.199.....

rappresentano lo stesso numero. La parte $\gamma_0.\gamma_1\gamma_2\ldots\gamma_t\ldots$ viene chiamata mantissa e noi la indicheremo con m. Il numero q viene chiamato esponente, mentre N è la base. Ogni numero reale non nullo sarà quindi indicato in questa rappresentazione, detta *rappresentazione normalizzata*, con:

$$\pm m\cdot N^q$$

[1]Seguiamo la convenzione più recente in alternativa a quella equivalente $\pm.\gamma_0\gamma_1\ldots\gamma_t\ldots\cdot N^{q+1}$ che si trova più frequentemente nei libri.

Il numero zero sarà rappresentato con 0. Ovviamente nella rappresentazione di un numero intero tutti i valori γ_i saranno nulli da un certo indice (maggiore di uno) in poi. Essendo $\gamma_0 \neq 0$ si ha:

$$1 \leq m < N. \tag{1.3}$$

Una stessa stringa rappresenta numeri diversi a seconda della base usata. Ad esempio la stringa 123 in base quattro rappresenta il numero:

$$1 \cdot 4^2 + 2 \cdot 4 + 3 = \text{ventisette}$$

cioè il numero che in base 10 indichiamo con 27. Nel caso in cui N fosse più grande di dieci, il numero di simboli che usualmente si usano per indicare le cifre tra 0 ed $N-1$, cioè 0,1,2,...,9 non bastano più. Si fa ricorso alle lettere maiuscole dell'alfabeto per integrare i simboli mancanti. Ad esempio se N = 16, i simboli usati sono 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F. Intendendo con questo che il numero dieci viene rappresentato con A, l'undici con B, ecc.. Il numero sedici sarà ovviamente indicato con 10.

Esiste una convenienza nell'uso di una base al posto di un'altra? Diamo uno sguardo alla tabella seguente in cui il numero (non la stringa!) centoventitrè è rappresentato in diverse basi:

base	rappresentazione
16	7B
10	123
5	443
4	1323
2	1111011

È evidente che lo stesso numero richiede un numero di cifre crescente al diminuire della base. Ciò può risultare scomodo. Vi è però certamente un vantaggio nell'usare le basi più piccole. Infatti i calcoli risultano semplificati. La tavola pitagorica, che in base dieci è formata da un quadrato di 100 elementi, nella base 2 risulterebbe formata da 4 elementi.

La base due è ancora più interessante perché in tale base tutti i numeri sono rappresentati da successioni dei due simboli 0 ed 1 che possono essere messe in corrispondenza con ogni sistema che abbia solo due stati (ad esempio acceso-spento, magnetizzato-smagnetizzato ecc.). Difatti è per questo

motivo che essa risulta indispensabile per la codifica di informazioni nei calcolatori. Ogni informazione, per essere memorizzata nel calcolatore, viene opportunamente codificata nel linguaggio comprensibile alla macchina che è appunto basato sui simboli del sistema di numerazione binario, 0 ed 1. Abbiamo già avuto occasione di dire che questi due simboli rappresentano lo stato fisico dei circuiti di cui è costituita la memoria, che possono essere spenti (=0) o accesi (=1).

Un numero reale rappresentato tramite la (1.2) deve essere prima espresso in cifre binarie 0 ed 1 per poter essere memorizzato in un calcolatore.

Ovviamente la rappresentazione (1.2) necessita un numero infinito di cifre e quindi sarebbe impossibile memorizzarla in un calcolatore. Nei calcolatori ad ogni numero viene riservato uno spazio fisso di memoria, per cui sia il numero di cifre della mantissa che il numero di cifre dell'esponente non possono superare dei limiti prefigurati. I numeri di macchina detti anche numeri floating point sono del tipo:

$$\pm \gamma_0.\gamma_1\gamma_2 \dots \gamma_t \cdot N^q \tag{1.4}$$

con $\gamma_0 \neq 0$ e $q_{min} \leq q \leq q_{max}$ più, naturalmenete, lo zero.

1.2 Standard IEEE

Descriviamo più in dettaglio come viene rappresentato un numero usando lo standard floating point IEEE, che costituisce un insieme di regole definito dall'istituto degli ingegneri elettrici e elettronici per la rappresentazione ed elaborazione dei numeri floating point nei computer. Lo standard IEEE specifica esattamente cosa sono i numeri floating point e come sono rappresentati nell'hardware e ha 4 scopi principali:

1. Rendere l'aritmetica floating point il più accurata possibile;
2. Produrre risultati sensati in situazioni eccezionali;
3. Standardizzare le operazioni floating point fra i computer;
4. Dare al programmatore un controllo sulla manipolazione delle eccezioni;

Il punto (1) si ottiene in due modi. Lo standard specifica esattamente come un numero floating point dovrebbe essere rappresentato nell'hardware e

richiede che le operazioni (addizione,radice,...) siano il più accurate possibili. Per il punto (2) lo standard produce Inf (Infinito) per indicare che un risultato è più grande del massimo numero floating point rappresentabile e NaN (Not a Number) per indicare che un risultato non ha senso. Prima dello standard la maggior parte dei computer avrebbero bloccato un programma in tali circostanze. Il punto (3) ha alcune conseguenze:

(i) I numeri floating point possono essere trasferiti fra due computer che utilizzano lo standard IEEE in binario senza perdere precisione;

(ii) I dettagli dell'aritmetica floating point dovrebbero essere conosciuti dal programmatore;

(iii) Lo stesso programma su computer diversi dovrebbe produrre lo stesso risultato, anche se questo non è vero in pratica.

Ricordiamo che l'unità base di informazione immagazzinata da un computer è il bit, una variabile il cui valore è o 0 o 1. I bit sono organizzati in gruppi di 8 chiamati byte. L'unità più comune è la parola di 4 byte (32 bit). Per accuratezze più alte è raddoppiata a 8 byte o 64 bit. Vi sono 2 possibili valori per un bit, $2^8 = 256$ possibili valori per un byte e 2^{32} differenti parole di 4 byte.

I due tipi di numeri rappresentati sono interi (fixed point) e reali (floating point). Un numero intero ha tipo float in c e real in Fortran e Matlab. Nella maggior parte dei compilatori c e in Matlab un float ha per default 8 byte invece di 4.

L'aritmetica con gli interi è molto semplice. Vi sono $2^{32} \approx 4 \cdot 10^9$ interi a 32 bit che coprono l'intervallo da $-2 \cdot 10^9$ a $2 \cdot 10^9$. Addizione, sottrazione e moltiplicazione sono fatte esattamente se la risposta è compresa nell'intervallo. La maggior parte dei computer danno risultati imprevedibili se il risultato è fuori dal range (overflow). Lo svantaggio dell'aritmetica con gli interi è che non possono essere rappresentate le frazioni e l'intervallo dei numeri è piccolo.

Il formato IEEE sostituisce la base 10 con la base 2. Quando una parola a 32 bit è interpretata come un numero floating point il primo bit è il bit del segno, $s = +$ o$s = -$. I successivi 8 bit formano l'esponente e i rimanenti 23 bit determinano la mantissa. Vi sono 2 possibili segni, 256 esponenti (che variano da 0 a 255) e $2^{23} \approx 8.4$ milioni di possibili mantisse. Per avere gli esponenti negativi si usa una convenzione: lo zero è nella posizione 127, dopo ci sono i numeri positivi e prima i numeri negativi. Quindi in memoria

viene rappresentato $q^* = q + 127$. Inoltre il primo bit della mantissa, che rappresenta γ_0, è sempre uguale a 1, quindi non vi è necessità di memorizzarlo esplicitamente. Nei bit assegnati alla mantissa viene memorizzato m^* e $m = 1.m^*$. Un numero floating point positivo ha quindi il valore $x = +2^{q^*-127} \cdot (1.m^*)_2$ e la notazione $(1.m^*)_2$ indica che $1.m^*$ è interpretata in base 2. Per esempio il numero $2.752\dot{1}0^3 = 2572$ può essere scritto:

$$\begin{aligned} 2752 &= 2^{11} + 2^9 + 2^7 + 2^6 \\ &= 2^{11}(1 + 2^{-2} + 2^{-4} + 2^{-5}) = \\ &= 2^{11}(1 + (0.01)_2 + (0.0001)_2 + (0.00001)_2) \\ &= 2^{11} \cdot (1.01011)_2 \end{aligned}$$

allora la rappresentazione di questo numero avrebbe segno + esponente $q^* = q + 127 = 138 = (10001010)_2$ e $m^* = (010110\cdots0)_2$.

Il caso $q^* = 0$ (che corrisponde a 2^{-127}) e il caso $q^* = 255$ (che corrisponde a 2^{128} hanno una interpretazione differente e complessa che rende la IEEE diversa dagli altri standard. Tendendo conto, quindi, che il più piccolo valore di q^* è 1 e il più grande è 254 si ha che il più piccolo numero positivo rappresentabile è $Realmin = 2^{-126} \approx 10^{-38}$, mentre il più grande numero è $Realmax = (1.111\ldots1)_2 2^{127} \approx 10^{38}$ La distanza tra $Realmin$ e il numero di macchina successivo, che ha sempre esponente $q^* = 1$ e $m^* = 0\ldots01$, è 2^{23} volte più piccola della distanza fra $Realmin$ e 0. Per questo sono stati introdotti i numeri denormalizzati, che si ottengono quando $q^* = 0$. In questo caso, per convenzione, viene rappresentato in memoria il numero $x = \pm(0.m^*)_2 2^{-126}$. In questo modo si genera l'underflow graduale (l'underflow è la situazione in cui il risultato di una operazione è diversa da zero ma è più vicina a zero di qualsiasi numero floating point). L'underflow graduale ha la conseguenza che due numeri floating point sono uguali se e solo se sottraendone uno dall'altro si ha esattamente zero.

L'altro caso particolare è $q^* = 255$ che ha due sottocasi, Inf se $m^* = 0$ e NaN se $m^* \neq 0$. Sia il c che il Matlab stampano Inf o NaN quando si vuole visualizzare il contenuto di una variabile floating point che contiene questo risultato. Il computer produce Inf se il risultato di una operazione è più grande del più grande numero rappresentabile. Operazioni invalide che producono come risultato NaN sono: Inf/Inf, $0/0$, $Inf - Inf$. Operazioni con Inf hanno il significato usuale: $Inf + finito = Inf$, $Inf/Inf = NaN$, $Finito/Inf = 0$, $Inf - Inf = NaN$.

L'aritmetica IEEE in doppia precisione usa 8 byte (64 bit). Vi è un bit del segno, 11 bit per l'esponente e 52 bit per la mantissa. Vengono rappresentati

i numeri $\pm(1.m^*)_2 2^{q-1023}$, Realmin corrispnde a $2^{-1022} \approx 10^{-308}$ e Realmax a $(1.11\ldots1)_2 2^{1023} \approx 2^{1024} \approx 10^{308}$.

1.3 Errore assoluto e relativo

Consideriamo il numero reale:

$$x = \pm\gamma_0.\gamma_1\gamma_2\ldots\gamma_t\ldots\cdot N^q$$

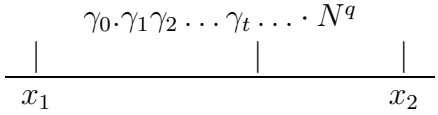
Se consideriamo i numeri di machhina con t cifre per la matinssa, x sarà compreso fra:

$$x_1 = \pm\gamma_0.\gamma_1\gamma_2\ldots\gamma_t\cdot N^q$$

e

$$x_2 = x_1 + 0.0\ldots01\cdot N^q = N^{-t}\cdot N^q$$

Graficamente si ha:



Vi sono due modi diversi per approssimare questo numero mediante i numeri floating point, in cui solo t cifre della mantissa sono rappresentabili:

1) il troncamento;

2) l'arrotondamento.

Nel primo caso il numero x viene sempre approssimato con x_1 trascurando tutte le cifre successive a γ_t; nel secondo caso il numero viene approssimato con x_1 se $x < (x_1 + x_2)/2$, cioè x si trova nella prima metà dell'intervallo $[\mathbf{x}_1 x_2]$, altrimenti viene approssimato con x_2.

Denotiamo con $fl(x)$ l'approssimazione di x. Se viene usato il troncamento qualunque sia la posizione di x nell'intervallo, viene approssimato con il numero di macchina alla sua sinistra e quindi l'errore assoluto, cioè la distanza in valore assoluto di $fl(x)$ da x, sarà sempre minore dell'ampiezza dell'intervallo $[x_1, x_2]$:

$$|fl(x) - x| < N^{-t}\cdot N^q$$

Nel secondo caso invece l'errore assoluto sarà sempre minore o uguale della metà dell'ampiezza dell'intervallo:

$$|fl(x) - x| \leq \frac{1}{2} N^{-t} N^q$$

L'errore assoluto dipende dalla grandezza del numero considerato, conviene quindi utilizzare l'errore relativo definito per un numero $x \neq 0$ da:

$$\left| \frac{fl(x) - x}{x} \right|$$

Teorema 1.3.1 *Se $x \neq 0$ e usiamo t cifre per rappresentare la mantissa, allora:*

$$\left| \frac{fl(x) - x}{x} \right| \leq u \tag{1.5}$$

dove:

$$u = \begin{cases} N^{-t} & \text{in caso di troncamento} \\ \frac{1}{2} N^{-t} & \text{in caso di arrotondamento} \end{cases}$$

Dimostrazione. Sia $x = \gamma_0.\gamma_1\gamma_2 \ldots \gamma_t \ldots \cdot N^q$, allora $|x| \geq N^q$ e, nel caso di troncamento, si ha:

$$\frac{|x - fl(x)|}{|x|} \leq \frac{N^{q-t}}{N^q} \leq N^{-t},$$

da cui segue la tesi.
Si procede analogamente nel caso di arrotondamento.

Il numero u è detto *unità di arrotondamento(o precisione di macchina)*. Secondo lo standard IEEE un numero reale in doppia precisione occupa 64 bits (= 8 bytes) di memoria di cui 52 sono per la mantissa. Essendo la base $N = 2$, la precisione di macchina risulta essere $u = 2^{-52} \approx 2.22044 \cdot 10^{-16}$.

L'accuratezza delle operazioni floating point è determinata dalla grandezza degli errori di arrotondamento e come conseguenza del Teorema 1.3.1 si ha che l'errore relativo, piuttosto che l'errore assoluto, è legato alle cifre esatte di un numero x.
Poniamo:

$$\epsilon = \frac{fl(x) - x}{x}$$

Sappiamo che $|\epsilon| \leq u$. Dalla precedente si ricava:

$$fl(x) = x(1 + \epsilon), \tag{1.6}$$

cioè per ogni x tale che $Realmin \leq |x| \leq Realmax$ esiste un numero ϵ in valore assoluto minore od uguale ad u tale che la sua approssimazione mediante un numero di macchina può essere espressa dalla (1.6). Se vale il Teorema 1.3.1 possiamo dire che il numero è rappresentato con t cifre esatte.

Esercizio 1.3.1 Tenendo conto che u è il più piccolo numero tale che $1+u > 1$, scrivere un algoritmo per determinare la precisione di macchina di un calcolatore.

1.4 Operazioni con i numeri di macchina

Indichiamo con *op* una qualunque operazione aritmetica. Non vogliamo entrare nei dettagli sul modo in cui vengono eseguite le operazioni nel calcolatore. Accenneremo solo brevemente al modo in cui si eseguono le addizioni e le moltiplicazioni. Supponiamo di voler eseguire l'addizione tra due numeri $x = \alpha_0.\alpha_1\alpha_2 \ldots \alpha_t \cdot N^p$ e $y = \beta_0.\beta_1\beta_2 \ldots \beta_t \cdot N^q$, in cui $p > q$. Dobbiamo trasfomare i due numeri in modo da avere i due esponenti uguali a p. Per fare ciò bisogna che l'area di memoria in cui si eseguono le operazioni abbia uno spazio riservato alle mantisse superiore a quello usuale. Nel nostro caso si pone $y = (\beta_0.\beta_1\beta_2 \ldots \beta_t \cdot N^{q-p}) \cdot N^p = .0 \ldots 0\beta_0\beta_1 \ldots \beta_t \cdot N^p$ e poi si sommano le mantisse. Si procede quindi al troncamento o all'arrotondamento. In generale quindi il risultato di una operazione può considerarsi come il risultato esatto dell'operazione tra x ed y seguito dall'arrotondamento o dal troncamento, in modo tale da soddisfare in genere la seguente relazione:

$$fl(x \text{ op } y) = (x \text{ op } y)(1 + \epsilon) \qquad |\epsilon| < u. \tag{1.7}$$

Le operazioni con i numeri di macchina (aritmetica *floating point*), non godono di tutte le proprietà di cui godono le corrispondenti operazioni con i numeri reali. In generale non vale più la proprietà associativa. Consideriamo ad esempio il seguente caso in cui $N = 10$ e $t = 2$. Eseguiamo la somma $5.24 \cdot 10^{-2} + 4.04 \cdot 10^{-2} + 1.21 \cdot 10^{-1}$ in due modi diversi:

1) $5.24 \cdot 10^{-2} + (4.04 \cdot 10^{-2} + 1.21 \cdot 10^{-1}) = 5.24 \cdot 10^{-2} + (0.404 + 1.21)10^{-1} = 5.24 \cdot 10^{-2} + 1.61 \cdot 10^{-1} = (0.524 + 1.61)10^{-1} = 2.13 \cdot 10^{-1}$;

2) $(5.24 \cdot 10^{-2} + 4.04 \cdot 10^{-2}) + 1.21 \cdot 10^{-1} = 9.28 \cdot 10^{-2} + 1.21 \cdot 10^{-1} = (0.928 + 1.21)10^{-1} = 2.14 \cdot 10^{-1}$.

Supponiamo ora di avere due numeri reali x ed y e di volere fare la somma $x + y$. Nel calcolatore x sarà rappresentato da $fl(x) = x(1 + \epsilon_x)$ e y da $fl(y) = y(1 + \epsilon_y)$. Inoltre $fl(fl(x) + fl(y)) = (fl(x) + fl(y))(1 + \epsilon_+)$. L'errore relativo sarà quindi, trascurando i termini contenenti $\epsilon_+\epsilon_x$ ed $\epsilon_+\epsilon_y$,

$$\begin{aligned}\frac{(fl(x) + fl(y))(1 + \epsilon_+) - (x + y)}{x + y} &= \frac{(x + x\epsilon_x + y + y\epsilon_x)(1 + \epsilon_+) - x - y}{x + y} \\ &\approx \epsilon_+ + \epsilon_x \frac{x}{x + y} + \epsilon_y \frac{y}{x + y}.\end{aligned}$$

Dal precedente risultato risulta evidente che se il denominatore non è piccolo, come avviene certamente nel caso in cui i due numeri siano dello stesso segno, l'errore relativo è dello stesso ordine degli errori relativi ϵ_+, ϵ_x ed ϵ_y e quindi in valore assoluto minore di $3u$. Ciò non avviene nel caso in cui il denominatore risulti molto piccolo. Ciò risulterà evidente nel seguente esempio.

Supponiamo infatti di voler effettuare la differenza tra i due numeri reali $x_1 = 0.147554326$ ed $x_2 = 0.147251742$ ed operare con $t = 5$ ed $N = 10$. Usando l'arrotondamento sarà $fl(x_1) = 1.47554 \cdot 10^{-1}$ e $fl(x_2) = 1.47252 \cdot 10^{-1}$. Si ha $fl(fl(x_1) - fl(x_2)) = 3.02000 \cdot 10^{-4}$, mentre la vera differenza è $3.02584 \cdot 10^{-4}$. Le ultime tre cifre della mantissa risultano errate. Ciò è dovuto al fatto che dopo aver eseguito la differenza $fl(x_1) - fl(x_2) = 0.000302$, la rappresentazione normalizzata $3.02000 \cdot 10^{-4}$ ha introdotto tre zeri alla fine della mantissa. Ovviamente questi tre zeri non sono esatti. L'errore relativo risultante è:

$$\frac{3.02584 - 3.02000}{3.02584} \approx 10^{-3}$$

che è piuttosto elevato. Quindi bisogna fare particolare attenzione a questo fenomeno, che viene chiamato *cancellazione numerica*. Essa si presenta quando si esegue la sottrazione tra numeri molto vicini.

L'operazione di moltiplicazione non presenta questo tipo di problema. Infatti l'errore relativo sarà, semnpre trascurando i termini contenenti contenenti $\epsilon_*\epsilon_x$ ed $\epsilon_*\epsilon_y$,

$$\frac{fl(x)fl(y)(1+\epsilon_*) - xy}{x * y} \approx \frac{(xy + xy\epsilon_x + xy\epsilon_x)(1+\epsilon_*) - xy}{xy} \approx$$
$$\approx \epsilon_* + \epsilon_x + \epsilon_y.$$

e quindi l'errore relativo risulterà in valore assoluto minore di $3u$. L'analisi degli errori appena analizzata si chiama *analisi degli arrori in avanti (forward)* .

Spesso è comodo vedere la (1.7) da un punto di vista leggermente diverso. Consideriamo per esempio il caso in cui *op* sia l'addizione. Sarà:

$$fl(x+y) = (x + \epsilon_+ x) + (y + \epsilon_+ y),$$

cioè il risultato dell'operazione di macchina può essere visto come l'operazione esatta su dati perturbati $(x + \delta x)$ e $(y + \delta y)$. Ovviamente nel caso considerato sarà $\delta x = \epsilon x$ e $\delta y = \epsilon y$. Nel caso in cui *op* sia la moltiplicazione si ha:

$$fl(xy) = xy(1+\epsilon_*) = (x+\delta x)(y+\delta y) \text{ con } \delta x = 0, \delta y = \epsilon_* y.$$

La tecnica di considerare la perturbazione nei dati che porterebbe allo stesso risultato finale se le operazioni fossero eseguite con precisione infinita è molto utile in Analisi Numerica e viene chiamata *analisi degli errori all'indietro (backward)*.

1.5 Propagazione degli errori

Consideriamo una funzione $y = f(x)$ e supponiamo di volerla calcolare in un punto particolare $\overline{x}$ il quale è stato misurato con un errore Δx. Questo errore si ripercuote sulla variabile y con un errore Δy che, nel caso in cui f ammetta derivata prima in un intorno di $\overline{x}$ si può stimare facilmente. Si ha:

$$\overline{y} + \Delta y = f(\overline{x} + \Delta x) \simeq f(\overline{x}) + f'(\overline{x})\Delta x$$

da cui:

$$\frac{\Delta y}{\overline{y}} \simeq \overline{x}\frac{f'(\overline{x})}{f(\overline{x})}\frac{\Delta x}{\overline{x}}.$$

Si deduce che un errore relativo $\Delta x/\overline{x}$ sulla variabile indipendente produce un errore relativo sulla variabile dipendente che risulterà notevolmente amplificato se la quantità:

$$K(\overline{x}, f) = \left| \overline{x} \frac{f'(\overline{x})}{f(\overline{x})} \right|$$

è molto grande. Questa quantità è detta indice di condizionamento del problema. Se $K(\overline{x}, f)$ è grande il problema si dice mal condizionato, altrimenti si dice ben condizionato.

Esempio 1.5.1 Sia $f(x) = \log x$. Si ha $K(x, f) = 1/\log(x)$ da cui risulta che il calcolo del logaritmo è un problema mal condizionato se x è vicino ad 1.

Esempio 1.5.2 Consideriamo il problema di calcolare le radici di un polinomio. In questo caso i dati del problema sono i coefficienti del polinomio e il risultato finale le sue radici. Se consideriamo il seguente polinomio:

$$(x-1)^4 = x^4 - 4x^3 + 6x^2 - 4x + 1$$

sappiamo che la radice è 1 con molteplicità 4. Perturbiamo il termine noto ottenendo il polinomio:

$$(x-1)^4 - 1e - 8 = x^4 - 4x^3 + 6x^2 - 4x + 1 - 1e - 8$$

il quale ha le radici:

$$\begin{aligned} x_1 &= 1.01 \\ x_2 &= 0.99 \\ x_3 &= 1 + i0.01 \\ x_4 &= 1 - i0.01 \end{aligned}$$

quindi una pertubazione relativa di 1e-8 sul termine noto produce una perturbazione relativa di 1e-2 sul risultato. Possiamo concludere che il problema è mal condizionato.

Quando il risultato di un problema non si ottiene soltanto calcolando il valore di una funzione regolare in un punto, lo studio della propagazione degli errori e del condizionamento del problema può essere molto laboriosa.

Studiare il condizionamento del problema è molto importante nel caso in cui si voglia risolvere il problema usando il calcolatore. In questo caso infatti una perturbazione sui dati viene sempre provocata quando gli stessi vengono memorizzati.