7. Costo computazionale degli algoritmi numerici

Negli scorsi capitoli abbiamo familiarizzato con due concetti cardine del calcolo numerico:

Convergenza

La maggior parte dei metodi numerici prevede la costruzione di una successione di oggetti (numeri, vettori, funzioni) che convergono in senso opportuno ad un oggetto limite, che e' l'oggetto da approssimare.
Si tratta di un processo infinito che va fermato quando, tramite opportune stime, l'approssimazione e' entrata in un prefissato intorno del limite, determinato da una tolleranza.
Lo schema tipico e':

$$\underbrace{\text{ERRORE}(n)}_{e_n=|x_n-l|} \tilde{\leq} \text{STIMA}(n) \leq \underbrace{\text{TOLLERANZA}}_{\epsilon}$$

dove si e' dimostrato teoricamente che $x_n \to l, n \to \infty$.

Stabilita'

In tutti gli algoritmi numerici, anche quelli che fanno a priori un numero finito di passi (come alcuni algoritmi dell'algebra lineare, ad esempio il metodo di eliminazione gaussiana), vengono introdotti errori furante il processo di calcolo, a partire dagli inevitabili errorei di arrotondamento, ed errori di misura dei dati, ad errori dovuti ad un algoritmo secondario che fornisce all'algoritmo primario dei risultati approssimati da elaborare (ad esempio un algoritmo che approssima uno zero di una funzione che a sua volta viene approssimata tramite uno sviluppo in serie).

Come abbiamo visto gia' con i vari esempi, cerchiamo di evitare algoritmi che propagano male gli errori amplificandoli, cioe' cerchiamo algoritmi che oltre ad essere **convergenti** siano anche **stabili** (si pensi alla successione di Archimede per π nella versione instabile che abbiamo poi stabilizzato).

Ma accanto a questi due concetti, ce n'e' un terzo che ci sice quando un algoritmo di approssimazione e' ben utilizzabile in pratica, ed e' il concetto di:

Efficienza

Tra i vari algoritmi che risolvono un problema, siamo interessati a queleli che hanno un basso **costo computazionale** (ovviamente a parita' di errore, visto che in questo corso trattiamo algoritmi numerici cioe' algoritmi che forniscono non un risultato esatto ma un risultato approssimato a meno di una certa tolleranza).

Si pensi ad esempio, per fissare le idee, all'algoritmo di Archimede per il calcolo di π che ha chiaramente a parita' di errore un costo molto piu' basso dell'algoritmo basato sulla serie armonica, visto che la convergenza e' molto piu' rapida.

Ma come si misura il costo computazionale di un algorimo numerico?
Consideriamo sostanzialmente due parametri:

- *# DI FLOPS* (floating-point operations);
- *TEMPO DI CALCOLO*.

Ovviamente il numeri di operazioni floating-point influenza il tempo di calcolo attraverso la velocita' del porcessore, che si misura in *flops/sec*, ad esempio un processore da 1 Gflops, fa 10^9 flops al secondo, che e' l'ordine di grandezza per il processore di un PC attuale (mentre i super-computer hanno ormai raggiunto i Pflops ossia 10^{15} flops/sec) e la tecnologia sta puntando all'Eflops (10^{18} flops/sec).

Ma il tempo effettivo di calcolo, che e' il parametro piu' importante dal punto di vista pratico (si pensi in particolare agli algoritmi "real-time" che devono fornire una risposta entro un tempo prefissato, in scale di tempi dipendenti dal problema, ad esempio frazioni di secondo per il controllo numerico di un macchinario industriale oppure ore/giorni nella simulazione dei modelli estremamamente complessi per le previsioni meteo).

In effetti il tempo di calcolo, oltre che dalla velocita' del processore e' influenzato anche dalla *velocita' dei flussi di dati* tra le varie parti della memoria del computer, ed e' un parametro che dipende dal tipo di computer ("machine dependent" come si dice in inglese informatico).

Invece il *# flops* ha il vantaggio di essere "machine independent" e di dare quindi una misura parzialmente incompleta ma in un certo senso "universale" del costo computazionale di un dato algorimo numerico.

Per capire l'effetto dei flussi di dati fra le diverse zone di memoria del computer in algoritmi che lavorano su grandi masse di dati, facciamo un esempio un po' ingenuo ma indicativo con un modello di computer molto semplificato, solo per fissare le idee.

Come e' noto, la velocita' di scambio dati con la memoria centrale (accesso veloce) puo' essere maggiore di vari ordini di grandezza rispetto allo scambio dati con l'hard-disk o altre memoria di massa.
Conviene quindi implementare gli algoritmi che lavorano su masse di dati e hanno bisogno delle memoria ad accesso piu' lento, minimizzando i flussi di dati, come mostriamo nel prossimo esempio.
Supponiamo di dover fare il prodotto di due matrici $A, B \in \mathbb{R}^{n \cdot m}$, con il vincolo che nella memroia centrale si puo' memorizzare solo una matrice e qualche vettore di dimensione n, ma non due matrici. Chiamiamo come spesso si fa in letteratura *FLOAT* un reale-macchina.

La situazione descrita non e' irrealistica: con una RAM di *8 Gbytes* possiamo memorizzare $\frac{8 \cdot 10^9}{8} = 10^9$ *floats*, cioe' un miliardo di floats a $64\text{bits} = 8\text{bytes}(1\text{byte} = 8\text{bit})$.
Quindi se $n = 30000$, ogni matrice occupa $n^2 = 9 \cdot 10^8$ floats e nella RAM non ci stanno entrambe le matrici A e B, una dlle due, ad esempio B deve essere memoriazzata nell'hard-disk, cosi' come la matrice prodotto $C = AB$.

Ricordiamo che $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$ cioe' $c_{i,j}$ e' il prodotto della *riga* i di A con la *colonna* j di B, $R_i(A) \cdot C_j(B)$, e' chiaro che il costo del calcolo misurato in flops e' $\approx 2 \cdot n^3$, visto che vanno calcolati n^2 prodotti riga-colonna e ciascuno costa n prodotti e $n - 1$ somme algebriche, cioe' $2n - 1$ flops.

Possiamo costruire la matrice C ad esempio per righe:

$$c_{i1} = R_i(A) \cdot C_1(B), c_{i2} = R_i(A) \cdot C_2(B), \ldots, c_{in} = R_i(A) \cdot C_n(B), \quad 1 \leq i \leq n$$

man mano che costruiamo le righe di C, le memorizziamo nell'hard disk.
Per ogni riga dobbiamo spostare dall'hard-disk alla RAM tutte le colonne di B e viceversa con la riga risultato, cioe' $n^2 + n$ floats, per un totale di $n(n^2 + n) \approx n^3$ floats.
Ma e' l'unico modo di procedere?

C'e' un altro modo, possiamo calcolare $C = AB$ per colonne, osservando che in generale il prodotto matrice-vettore e' una combinazione linare delle colonne della matrice che ha per coefficienti gli elementi del vettore; in notazione vettoriale:

$$A \cdot \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{bmatrix} = \sum_{j=1}^{n} u_j \cdot C_j(A)$$

(questa e' un'osservazione molto utile in molte dimostrazioni di algebra lineare), costruendo C per colonne $C_j(C) = A \cdot C_j(B), \quad 1 \leq j \leq n$.

In questo modo ogni colonna di B viene spostata una volta sola, mentre nella costruzione (molto inefficiente) di C per righe per calcolare ogni riga di C bisognava spostare tutta la matrice B.
Il flusso di dati si riduce quindi a $2n$ floats per colonna di C e quindi in totale $2n^2$ invece di $n^3 + n^2$ floats.

Il guadagno in termini di flussi di dati e' evidente, anche se il numeri di flops resta ovviamente lo stesso cioe' $\approx 2n^3$.
Nel seguito non ci occuperemo di algoritmi che elaborano grandi masse di dati, ma faremo esempi di confonto di algoritmi che risolvono lo stesso problema con costi computazionali diversi, usando $c_n = \#\text{ flops}$ come parametro per misurare il costo computazionale (in funzione di un parametro n che misura la "dimensione" del problema).

Esempio 1: calcolo del valore di un polinomio

Sia $p(x) = a_0 + a_1 x + \cdots + a_n x^n$ un polinomio di grado n, quanto costa calcoalre il valore di p in un punto x?
Il primo algoritmo di calcolo che viene in mente e'

$$p(x) = \underbrace{\underbrace{a_0}_{} \underbrace{+a_1 x}_{} \underbrace{+a_2 x^2}_{} + \cdots + a_n x^n}_{}$$

cioe sommare successivamente i monomi dal grado 0 al grado n cosi' ad ogni passo si tratta di fare due moltiplicazioni (una per $x^k = x \cdot x^{k-1}$ e una per $a_k \cdot x^k$) e una somma lagebrica (somma del nuovo monomio alla somma precedente
$S_k = a_k x^k + S_{k-1}, S_{k-1} = \sum_{j=0}^{k-1} a_j x^j, k = 1, 2, \ldots, n$) quindi il costo totale e' $c_n^{(1)} = 3n$ flops.

Ma questo non e' l'unico modo di procedere.

Per capirlo, riscriviamo in modo opportuno un polinomio di grado 3:

$$p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 = \big((a_3 x + a2) \cdot x + a_1\big) \cdot x + a_0$$

Vediamo che in questo modo le potenze x^k non appaiono esplicitamente, ma sono implicite nella rappresentazione.
In generale

$$p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \Bigg(\cdots \Big(\big((a_n x + a_{n-1}) x + a_{n-2} \big) x + \cdots \Big) x \Bigg) + a_0.$$

Con questa rappresentazione, non dovendo piu' calcolare le potenze di x, il conto diventa $c_n^{(2)} = 2n$ flops.

Lo "speed-up", cioe' il guadagno dell'algoritmo 2 (chiamato **schema dittormer**) rispetto all'algoritmo 1 e' $\dfrac{c_n^{(1)}}{c_n^{(2)}} = \dfrac{3}{2}$ (l'algoritmo 1 costa 1.5 volte l'algoritmo 2).
Faremo ora un esempio in cui il guadagno e' ben piu' notevole.

Esempio 2: calcolo di una potenza ad esponente intero

Il problema qui e' il calcolo di a^n, con $a \in \mathbb{R}^+$ e $n \in \mathbb{N}$ (possiamo limitarci agli interi positivi, visto che $a^{-n} = \dfrac{1}{a^n}, a \neq 0$).
Dalla definizione di potenza:

$$a^n = \underbrace{a \cdot a \cdot \ldots \cdot a}_{n-1 \text{ moltiplicazioi}}$$

Quindi banalmente il costo comptazionale e' $c_n^{(1)} = n - 1$ flops e sembra difficile fare in modo diverso.

Invece e' possibile, con un'idea molto furba che parte dalla seguente considerazione: se n e' una potenza di 2, $n = 2^m$, si puo' calcolare a^n facendo solo m moltiplicazioni.

Per caprilo prendiamo $n = 16 = 2^4$

$$4 \text{ moltiplicazioni } \begin{cases} a^2 = a \cdot a \\ a^4 = a^2 \cdot a^2 \\ a^8 = a^4 \cdot a^4 \\ a^{16} = a^8 \cdot a^8 \end{cases}$$

Quindi a^{2^m} si calcola con $m = \log_2(m)$ moltiplicazioni.

E se n non e' una potenza di 2? Qui ci viene in aiuto la rappresentazione di n in base 2 (codifica binaria):

$$n = \sum_{j=0}^{n} c_j 2^j$$

dove $c_j \in \{0, 1\}$ sono le cifre binarie e $m = [\log_2(n)]$ (dove $[z]$ rappresenta la parte intera, cioe' il piu' piccolo intero $\leq z \in \mathbb{R}$).
Ad esempio:

$$7 = 1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2 = (111)_2$$
$$12 = 0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3 = (1100)_2$$

Usando le proprieta' delle potenze

$$a^n = a^{\sum_{j=0}^{m} c_j 2^j} = \prod_{j=0}^{m} a^{c_j 2^j}$$

Ad esempio:

$$a^7 = a^{1 \cdot 2^0 + 1 \cdot 2^1 + 1 \cdot 2^2} = a \cdot a^2 \cdot a^4$$
$$a^{12} = a^{0 \cdot 2^0 + 0 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3} = a^0 \cdot a^0 \cdot a^4 \cdot a^8$$

Quante moltiplicazioni stiamo facendo? Ci sono $m = [\log_2(n)]$ moltiplicazioni per calcolare $a^2, a^4, \ldots, a^{2^m}$ e nel prodotto $\prod_{j=0}^{m}$ ci sono poi un numero di moltiplicazioni uguale al numero di cifre 1 nella codifica binaria, meno uno (le cifre 0 non contano perche' $a_0 = 1$).
Quindi il totale e' $c_n^{(2)} = m + (\#\{1\} - 1)$.
Siccome $\#\{1\}$ e' al massimo $m + 1$ e questo accade per $n = 2^k - 1$ (ad esempio $15 = 2^4 - 1 = (1111)_2$) in cui la codifica binaria di n e' una sequanza di 1, si ha che

$$\max c_n^{(2)} = m + m = 2[\log_2(n)]$$

Quindi lo speed-up minimo diventa

$$\min \text{speed-up} = \frac{c_n^{(1)}}{\max c_n^{(2)}} = \frac{n - 1}{2[\log_2(n)]}$$

Al crescenere di n si ha che $\min \text{speed-up} \sim \frac{n}{2\log_2(n)}$

Mentre $\max \text{speed-up} \sim \frac{n}{\log_2(n)} = \frac{2^m}{m}$ che avviene quando n e' una potenza di 2 come visto all'inizio