Relazione esercitazione sugli errori

Andres Coronado MAT 2761046

3 luglio 2019

1 Errori di somme tra numeri distanti tra loro

1.1 Il tipo double

In c++ il tipo double è a 64 bit suddivisi come in tabella:

segno	esponente	mantissa	totale
1	11	52	64

Il sistema di memorizzazione dei numeri in virgola mobile nello standard IEEE 754 utilizzato da C++ per rappresentare i numeri float, double, long double ed in generale i numeri in virgola mobile si usano tre campi: segno s(1 bit, 2 possibili valori),mantissa M(52 bit) ed esponente e(11 bit 2048 possibili valori), mediante la relazione :$(-1)^s \cdot 2^E \cdot M$. Dove E rappresenta l'esponente polarizzato con un bias k ($E = \#e - k$) pari a 1023, ed M la mantissa normalizzata(siccome la prima cifra significativa della mantissa normalizzata in base 2 è sempre 1, si guadagna un bit!).Un numero macchina di tipo Double si può quindi pensare come la funzione

$$\mathfrak{F}(2, 52, 1024, 1023) = \{x \in \mathbb{R} : \pm f 2^p, -1024 \leq p \leq 1023, f = \sum_{i=1}^{52} d_i 2^{-i}\}$$

Dove $B = 2$ è la Base, $t = 52$ è il numero di cifre di cui è composta la mantissa, $-m = -1024$ è l'esponente minimo, sotto al quale si ha un Underflow ed $M = 1023$ è invece l'upper bound al di sopra del quale si andrebbe in Overflow.
L'errore relativo assoluto (precisione macchina) dato un numero macchina è minorato da

$$u = \frac{1}{2} \cdot 2^{-52+1} = 2^{-52} \approx 2.2 \cdot 10^{-16}$$
$$|\varepsilon| \leq u$$

Quindi relativamente all'esercitazione posso già intuire che considerando la somma di un numero α ed un numero β piccolo "abbastanza" cioè più piccolo di circa 16 ordini di grandezza rispetto ad α ($\beta < \alpha \cdot 2.2 \cdot 10^{-16}$) la somma così ottenuta non risente del valore β. In particolare nell'esercitazione sono presenti somme tra un numero dell'ordine di grandezza 10^i ($i \in [0, 6]$) a numeri dell'ordine di 10^{20}. Osservo quindi da quanto considerato che finchè i non è abbastanza grande (circa 4) questo tipo di somma non viene "sentita" e viene quindi troncata/approssimata.

1.2 Risultati

l'algoritmo fornito(ed implementato in C++) si può schematizzare come segue:
$alg0 : a0 := d0 + 1\ ap0 := 10^i\ b0 := d1 + 1\ ap1 := 10^i\ a := a0 * ap0\ b := b0 * ap1\ c := (-1) * b\ f0 := a + b\ f1 := b + c\ ya := f0 + c\ yb := a + f1$

l'output del programma conferma quanto osservato nel paragrafo precente, infatti a partire dalla quarta iterazione dove la somma $f0 = a + b$ vale $5.00000000000000007e + 20$ di conseguenza y_a comincia ad avvicinarsi a y_b al crescere di i (cioè nelle iterazioni successive).

1.3 Conclusione

Si dimostra come la rappresentazione dei numeri macchina possa alterare risultati di somme di numeri distanti tra loro.

2 Approssimazione di $exp(x)$ con polinomio di Taylor

2.1 Osservo i dati

Nel punto $x_1 = 0,5$ l'algoritmo 1 calcola una approssimazione di e^x rispetto a $exp(x)$ della libreria ANSI avente errori assoluto e relativo nell'ordine di 10^{-3} per un polinomio di grado 3. Al crescere del grado del polinomio l'errore assoulto e quello relativo migliorano (si riducono) ulteriormente per gradi $10, 50, 100, 150$ arrivando fino al limite miminimo di errore raggiungibile dai numeri in doppia precisione (per via delle somme "non sentite" di cui l'esercitazione precedente $\approx 10^{-16}$) già per un polinomio di grado 50.
Per quanto riguarda il punto $x_2 = 30$ invece noto subito un comportamento diverso: l'errore relativo e assoluto sono su una scala inaccettabile per quanto riguarda i polinomi di grado 3, 10 e 50 ($\approx 10^{10}$) per poi ridursi solo a partire da polinomio di grado 100 sulla scala di 10^{-3}. Osservo che: $f(30) \approx 1.06865 \cdot 10^{13}$ è un numero molto grande ed x_2 è piuttosto distante da 0, quindi mi aspetto errori grandi per polinomi grado basso. Provo ad espandere lo sviluppo di Taylor di grado 3 per x_2: si ha $f_3(x_2) = \frac{30^0}{0!} + \frac{30^1}{1!} + \frac{30^2}{2!} + \frac{30^3}{3!} = 4981$, risultato molto lontano rispetto a quello ottenuto con $exp(x_2)$ della libreria ANSI, (credo si possa fare di meglio centrando il polinomio su x_2 anzichè su $x_0 = 0$ ottenendo un risultato simile a quello ottenuto per x_1 in termini di errori relativo ed assoluto).

2.2 I termini negativi e algoritmo 2

I punti negativi ($x_3 = -0.5$ e $x_4 = -30$) vengono meglio approssimati dal secondo algoritmo poichè considerando il primo algoritmo si crea il problema per il quale i termini dispari della serie di Taylor danno luogo ad addendi di segno opposto rispetto agli addendi con n pari (e quando i numeri diventano grandi si introducono errori). Con l'algoritmo 2 questo non succede poichè gli addendi della serie sono tutti dello stesso segno, rendendolo preferibile.

3 Calcolo della precisione macchina ε_m

ε_m è il più piccolo numero positivo di macchina tale che $float(1.0 + \varepsilon_m) > 1.0$. per stimare ε_m Parto da un seme $seed = 0.5$ e finchè $1 + seed <> 1$ ne dimezzo il valore. a questo punto il valore ottenuto è per definizione ε_m. Per calcolare la precisione macchina sia a precisione singola $float$ che a precisione doppia *double* ho creato una classe *eps* con due costruttori diversi per ogni tipo.

4 Sistemi Lineari

4.1 Osservazioni

Il calcolo della norma matriciale (indotta) $||A||_\infty$, dove $A \in \mathbb{R}^{mxn}$ consiste nella ricerca della massima somma dei valori assoluti per riga e fornisce la massima distorsione positiva della lunghezza che la matrice (intesa come trasformazione lineare) produce.
Nel caso delle prime due matrici e della matrice tridiagonale 56x56 le norme sono numeri relativamente bassi e facilmente rappresentabili con numeri macchina a singola precisione (rispettivamente $14, 8$ e 4). Di fatti eseguendo il programma si nota che la soluzione del sistema $Ax = b$ calcolato mediante il metodo di eliminazione gaussiano (con pivotizzazione parziale) coincide con quella che ci aspettiamo (cioe' il vettore colonna $(1, ..., 1)^t$). Perturbando il sistema sommando al vettore dei termini noti i coefficienti proposti $||b||_\infty \cdot (-0.01, 0.01, ..., 0.01)^t$ le soluzioni si spostano di "poco", addirittura meno di quanto si "sposti" il vettore dei termini noti perturbati. Nel caso di A matrice di Pascal 56x56 invece i numeri diventano presto grandi: considerando la precisione macchina dei numeri a singola precisione calcolata nell'esercizio precedente $\varepsilon = 1.19209 \cdot 10^{-7}$ e notando che $||A||_\infty = 7.66628 \cdot 10^9$ posso già intuire che alcune somme tra valori che distano tra loro circa 10^7 non verranno rappresentate correttamente e verranno troncate / non sentite / approssimate. Infatti la soluzione di $Ax = b$ risente già di queste approssimazioni durante l'eliminazione gaussiana, restituendo alcuni risultati molto "lontani".
Introducendo una piccola perturbazione l'errore della soluzione di $Ax = b$ aumenta notevolmente al punto da essere pressochè inutilizzabile.

4.2 Il programma

Per implementare le matrici ho utilizzato una classe matrix che definisce al suo interno una struttura dati apropriata (uso un array bidimensionale allocato dinamicamente tramite la libreria vector di c++) e metodi

che ho utilizzato per semplificare e rendere più leggibile il codice. L'overloading degli operatori '+' e '*' mi ha permesso di scrivere i prodotti matriciali e scalari in forma A*Matrice e A*scalare. Inoltre ho definito la funzione swap che viene utilizzata dentro la classe MEG (metodo eliminazione gaussiana) per effettuare la riduzione con pivotizzazione parziale. La classe Meg effettua inoltre la sostituzione all'indietro e stampa i risultati. Siccome alcune matrici sono molto grandi c'è la possibilità di avere l'output su file 'output.txt'.Per compilare basta usare il comando '$ make' dalla directory principale del progetto, il file seguibile si può eseguire sulla shell di linux '$./err'. Per compilare è necessario GCC o Clang.