Laboratorio
di
Calcolo Numerico

Luca De Franceschi

Università degli studi di Padova

Indice

1 Esercitazione 1

1.1 Librerie di base per il calcolo seriale in Algebra Lineare Numerica (BLAS)

Gran parte dei problemi del calcolo scientifico ed ingegneristico richiede di risolvere uno o più problemi dell'algebra lineare numerica (ALN):

- Risoluzione di sistemi lineari;
- Risoluzione di problemi ai minimi quadrati;
- Ricerca di autovalori e/o autovettori;
- Calcolo della SVD (valori e vettori singolari);

Inoltre, la risoluzione di questi problemi è generalmente una percentuale considerevole del costo computazionale totale richiesto per la risoluzione del problema globale; questo costo si traduce spesso in ore o giorni di tempo-macchina impiegato. Dunque, implementare in modo efficiente gli algoritmi che risolvono i problemi dell'algebra lineare numerica è estremamente importante dal punto di vista applicativo ed è anche per questo che trattiamo questo caso più approfonditamente. Un secondo motivo è dovuto al fatto che esistono in rete delle librerie che rappresentano lo stato dell'arte per questi problemi e sono molto ben costruite anche dal punto di vista della implementazione e distribuzione del software numerico: BLAS, LAPACKe ATLAS (disponibili al sito www.netlib.org/blas , /lapack e /atlas).
Gli algoritmi di algebra lineare numerica hanno in comune un insieme relativamente piccolo e stabile di operazioni di base, che svolge la quasi totalità dei calcoli necessari negli algoritmi di ALN. Questo fatto giustifica lo sforzo di creazione di una libreria che implementi queste funzione di base. La libreria BLAS è organizzata in tre livelli: operazioni che lavorano su vettori e producono uno scalare (es. il prodotto interno, o scalare, tra due vettori colonna: v' * w), operazioni tra vettori e matrici o che comunque producono una matrice (es. il prodotto esterno tra due vettori colonna: v * w'), operazioni tra matrici (es. il prodotto di due matrici).
Un vantaggio rilevante di aver creato la BLAS è che questa può essere ottimizzata per ogni singola macchina ed in questo modo gli algoritmi di ALN possono diventare "portabili" anche dal punto di vista delle prestazioni di calcolo e non solo da quello della sintassi, semplicemente chiamando le routines della BLAS ove possibile.
Ora, ha senso chiedersi se le operazioni di livello 1, 2 o 3 raggiungono le stesse prestazioni di calcolo. Lo vediamo implementando lo stesso identico algoritmo, il prodotto di due matrici C = A * B, in tre modi diversi, corrispondenti all'utilizzo esclusivo di operazioni di livello 1, di livello 2 e di livello 3.

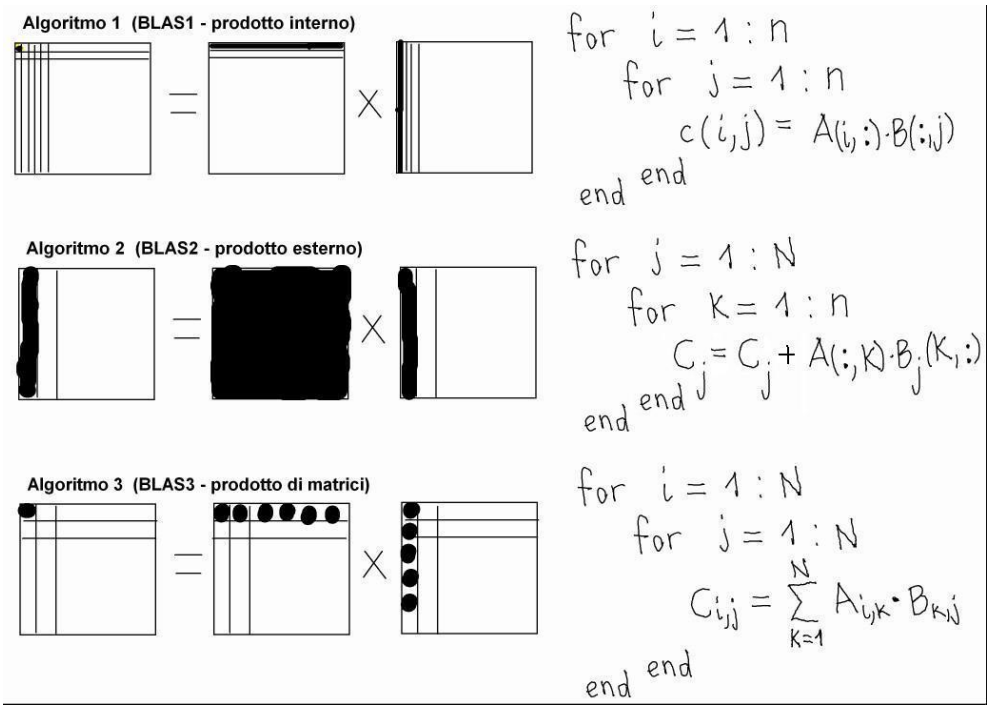

Figura 1: Algoritmi BLAS

2 Esercitazione 2

2.1 Ambienti software dedicati al calcolo scientifico

È ben noto che per creare programmi al "calcolatore" sia necessario disporre di un "ambiente di programmazione". Tale ambiente, in parte inserito nel sistema operativo ed in parte costituito da software aggiuntivo, fornisce strumenti indispensabili alla creazione e verifica dei programmi, come: un editor, un compilatore, un linker, un debugger, ...
Un ambiente software dedicato al calcolo scientifico aggiunge a questi strumenti degli altri strumenti dedicati allo sviluppo di applicazioni che utilizzano principalmente le capacità di calcolo del calcolatore. Da non confondere l'ambiente software dedicato al calcolo scientifico con la libreria di calcolo scientifico, che fornisce "solamente" un insieme di metodi numerici, richiamabili da un programma.
Per applicazioni del calcolo scientifico vengono tipicamente utilizzate assieme più d'una libreria. Ambienti molto diffusi di questo tipo sono: **Matlab** (il capostipite, 1983 ca., commerciale), **Octave** (open-source), **Scilab** (open-source). Recentemente, si sta affermando l'utilizzo di **Python**, grazie alla sua shell interattiva ed alle estensioni numeriche disponibili.

2.1.1 Il workspace

L'ambiente di calcolo, che si presenta con una linea di comando, mantiene disponibili in memoria tutte le variabili che sono state inizializzate dal momento dell'ingresso nell'ambiente, sia tramite singoli comandi che tramite istruzioni contenute nei programmi eseguiti, entrambi impartiti dalla linea di comando dell'ambiente. Questa è una sostanziale differenza rispetto alla programmazione in C o in FORTRAN, dove le variabili vengono allocate all'inizio dell'esecuzione del programma e poi de-allocate (cioè cancellate) al termine dell'esecuzione stessa. Una conseguenza immediata di questo fatto è che, ad esempio, in un ambiente come il Matlab non è necessario salvare esplicitamente su file i valori delle variabili che si vogliono osservare (es. graficare) dopo l'esecuzione del programma.

2.1.2 La grafica

Non è sufficiente produrre dei (buoni) risultati numerici, bensì è anche necessario poterli osservare a valutare in modo adeguato. Solitamente questa operazione richiede la produzione di qualche tipo di grafico, per cui l'esistenza di capacità grafiche è un elemento fondamentale di un ambiente software dedicato al calcolo scientifico. Matlab possiede capacità grafiche evolute ed Octave si appoggia a Gnuplot (applicativo anch'esso open-source).

2.1.3 L'esecuzione dei programmi

Per mandare in esecuzione un programma scritto per Matlab o per Octave (e quindi un file di testo con il suffisso ".m" contenente istruzioni che rispettano la sintassi del linguaggio di Matlab, ovvero un "m-file") è sufficiente scrivere il nome del m-file sulla riga di comando e premere il tasto invio. Se il programma non è nella directory corrente, è necessario prima mettersi nella directory che lo contiene. Il calcolo dei tempi di esecuzione è semplice, e quindi è possibile misurare la velocità di esecuzione dell'algoritmo numerico: conoscendo il numero di operazioni floating-point (flops) richieste per il calcolo della soluzione, si può calcolare la velocità:

$$MFLOPS = numeroDiLops/tempoTrascorso.$$

Notare che il tempo di esecuzione di un algoritmo numerico è oggi il principale parametro per misurarne l'efficienza. Infatti, mentre fino a qualche anno fa le operazioni in virgola mobile venivano fatte via software/firmware, e dunque una moltiplicazione costava più di un'addizione, ora sono implementate via hardware, e dunque richiedono sostanzialmente lo stesso numero di cicli di clock del processore. Pertanto, mentre fino a qualche tempo fa il parametro più usato era il numero di operazioni floating-point richieste, ora lo è il tempo di esecuzione dove, attenzione, pesano anche gli aspetti non numerici dell'algoritmo (esistenza di istruzioni di scelta, movimento dei dati da e verso la memoria, ...).
Per quanto riguarda l'ambiente di calcolo, è necessario tenere presente che Matlab (ed Octave) eseguono il programma utente in maniera interpretata, e cioè viene letta, compilata ed eseguita un'istruzione alla volta. Questo incide moltissimo sulla velocità di esecuzione di un algoritmo. In particolar modo è bene evitare di creare troppi cicli innestati.

2.1.4 Il linguaggio di programmazione "domain specific"

L'ambiente Matlab (Octave) mette a disposizione un linguaggio di programmazione che da un lato contiene tutti i costrutti tipici della programmazione strutturata (if-then-else, ciclo for, ciclo while, ecc...) e dall'altro permette di manipolare in modo molto semplice e sintetico le strutture dati fondamentali per il calcolo matriciale, e cioè vettori e matrici, in modo analogo a quanto viene fatto dal linguaggio tradizionale più evoluto per il calcolo, il Fortran90 e successive varianti, ed a quanto può essere fatto a partire dai linguaggi orientati agli oggetti, come il C++ , Java, ...
Il linguaggio di Matlab (Octave) offre il vantaggio, rispetto ai linguaggi di programmazione tradizionali, di semplificare molto l'uso del linguaggio per la costruzione di un programma, lasciando quindi l'attenzione del programmatore libera il più possibile dagli aspetti puramente informatici e concentrata sugli aspetti algoritmici e numerici. Anche per questo è molto diffuso, sia come strumento di studio che come strumento di lavoro, in particolare per la prototipazione dei codici di calcolo. Per un utilizzo in "produzione", i compilatori dei linguaggi tradizionali sono però preferibili perchè permettono di ottimizzare maggiormente il codice eseguito dalla macchina e dunque, generalmente, l'algoritmo implementato viene eseguito più velocemente (che, nella realtà delle applicazioni, è molto rilevante).

2.2 Rappresentazione dei numeri al calcolatore

I calcolatori oggi in commercio rispettano in genere il medesimo standard per la rappresentazione dei numeri in virgola mobile (floating-point): lo standard **IEEE 754**.
Nello standard IEEE la precisione singola occupa una parola da 32 bit, mentre la precisione doppia occupa due parole consecutive da 32 bit. Un numero non-nullo normalizzato "X" ha la seguente rappresentazione binaria:

$$\begin{array}{c} \mathbf{S\ E\ F} \\ X = (-1)^S * 2^(E - bias) * (1.F) \end{array}$$

dove bias = 127 (single) o 1023 (double).
Lo standard richiede che il risultato delle operazioni di addizione, sottrazione, moltiplicazione e divisione sia arrotondato esattamente, cioè il risultato deve essere calcolato esattamente e poi arrotondato al numero in virgola mobile più vicino.
Problemi in tal senso provengono dalla sottrazione. Per questo motivo, nei microprocessori moderni la sottrazione viene effettuata utilizzando la tecnica bit di guardia: l'operando più piccolo viene troncato a p+1 cifre binarie, mentre il risultato della sottrazione viene troncato a p cifre (binarie). In realtà, il calcolo con un solo bit di guardia non sempre produce lo stesso risultato che si avrebbe calcolando il risultato esatto e poi arrotondando a p cifre binarie. Introducendo un secondo bit di guardia ed un terzo bit sticky, si ottiene il pieno rispetto dello standard con un extra-costo computazionale relativamente piccolo.

2.3 Propagazione degli errori di arrotondamento

Ogni calcolatore ha a disposizione un numero finito M di cifre per rappresentare un numero reale. Se il numero da rappresentare possiede più di M cifre significative, il calcolatore lo arrotonda alle M cifre più significative. A causa di questo fatto, le operazioni aritmetiche di base sono in generale soggette ad un errore nel risultato, ed è dunque necessario conoscere l'entità di questo errore e, considerando un intero algoritmo, l'effetto potenziale della propagazione di questo tipo di errore sul risultato finale. In generale, la somma, la divisione e la moltiplicazione producono un errore relativo piccolo, mentre la sottrazione può anche produrre un errore relativo grande, rispetto al risultato (ciò avviene quando i due operandi sono molto vicini tra di loro e si ha dunque una notevole perdita di cifre significative nel risultato).
Il fatto che l'errore relativo nel risultato sia piccolo non è comunque una garanzia di accuratezza in generale: la somma tra due operandi troppo distanti in magnitudo può addirittura far scomparire dal risultato il più piccolo dei due ed avere comunque un errore relativo piccolo. Per questo motivo, ad esempio, quando si deve eseguire una sommatoria, è buona regola eseguirla mettendo gli operandi in ordine crescente, in modo che la somma parziale ed il prossimo numero da sommare siano il più vicini possibile in magnitudo.
Vediamo di seguito alcuni esempi relativi a questi fenomeni:

2.3.1 Calcolo delle radici di un'equazione di secondo grado

Vediamo ora un esempio concreto di calcolo in cui introdurre una sottrazione potenzialmente pericolosa conduca effettivamente a problemi di instabilità della soluzione, e come rimediare.
Data: $a * x^2 + b * x + c$, calcolare le sue radici.
La formula classica, $x = (-b \pm sqrt(b^2–4 * a * c))/(2 * a)$, è potenzialmente instabile a causa della sottrazione tra b/2 e sqrt(...) . Provare ad implementarla e verificarne la perdita di accuratezza per opportune scelte dei coefficienti a, b, e c.
Ripetere poi lo stesso tipo di indagine su una formula alternativa, stabile, che si ottiene calcolando prima la radice positiva (in cui non si effettua la sottrazione) e poi calcolando l'altra sapendo che vale :

$$c = a * x_1 * x_2.$$

Listato 1: Equazioni di secondo grado

```
figure(1); clf, hold on

for a=1
  for b=1:100:2000   % b positivo !!!!
    for c=0.000001
      if (b^2-4*a*c)>=0
        xn=(-b - sqrt(b^2-4*a*c))/(2*a);
        xp1=(-b + sqrt(b^2-4*a*c))/(2*a);
        xp2=(c/a)/xn;            % calcolo l'errore relativo
        disp(['xn=' num2str(xn) '  xp1=' num2str(xp1) '  xp2='
            num2str(xp2) '  (xp2-xp1)/xp2=' num2str((xp2-xp1)/xp2)
            ]);
        plot(b,(xp2-xp1)/xp2,'b*');
```

```
        end
      end
    end
end
```

2.3.2 Approssimazione del pigrego

È un altro esempio di sottrazione pericolosa, questa volta in un metodo numerico iterativo per l'approssimazione di pi-greco. Provare ad implementare i seguenti tre metodi iterativi ed a graficarne l'errore relativo, in scala logaritmica, per le prime 100 iterate (un valore molto accurato di pi-greco in Matlab/Octave è contenuto nella variabile di sistema "pi"). Per n = 2,3,4,.... calcolare le tre approssimazioni di pi-greco, y(n), w(n) e z(n):

1. $s(2) = 1 + 1/4;$
   $s(n+1) = s(n) + 1/((n+1)^2);$
   $y(n+1) = sqrt(6 * s(n+1));$
2. $w(2) = 2;$
   $w(n+1) = 2^(n - 1/2) * sqrt(1 - sqrt(1 - 4^(1 - n) * w(n)^2));$
3. $z(2) = 2;$
   $z(n+1) = (sqrt(2) * z(n))/(sqrt(1 + sqrt(1 - 4^(1 - n) * z(n)^2)));$

Listato 2: Approssimazione del pigreco

```
s(2)=1+1/4;

for n=2:100
  s(n+1)=s(n)+1/((n+1)^2);
  y(n+1)=sqrt(6*s(n+1));
  errorey(n+1)=abs(y(n+1)-pi)/pi;
end;

semilogy([3:101],errorey(3:101),'b-')
hold on

w(2)=2;

for n=2:100
  w(n+1)=2^(n-1/2)*sqrt(1-sqrt(1-4^(1-n)*w(n)^2));
  errorew(n+1)=abs(w(n+1)-pi)/pi;
end;

semilogy([3:101],errorew(3:101),'r-')


z(2)=2;

for n=2:100
  z(n+1)=(sqrt(2)*z(n))/(sqrt(1+sqrt(1-4^(1-n)*z(n)^2)));
  errorez(n+1)=abs(z(n+1)-pi)/pi;
end;

```

```
29 semilogy([3:101],errorez(3:101),'g-')
30
31 hold off
```

2.3.3 Successioni calcolabili praticamente

Si consideri la seguente successione:

$$y(0) = 1/e * (e-1);$$
$$y(n+1) = 1-(n+1) * y(n);$$

Essa converge (in aritmetica con infinite cifre) a zero per n-¿inf. Implementarla e verificare cosa accade nel calcolatore, cercando di giustificare i risultati. Inoltre, sapendo che y(n) è circa =0 per n sufficientemente grande, provare ad implementarla all'indietro e vedere cosa risulta per y(0). Perchè?

Listato 3: Successioni calcolabili

```
1  clear y;
2
3  % questa successione esplode in avanti ed è stabile all'indietro
4
5  y(1)=1/exp(1)*(exp(1)-1);
6
7  for n=1:100
8    y(n+1)=1-(n+1)*y(n);
9  end;
10
11 plot(y,'b-');
12 pause
13
14 % si sa che per n->inf y->0 (cioè y->1/n), però non la si riesce a
       calcolare perché già per n<200 l'algoritmo
15 % va in overflow e non si può procedere oltre.
16
17
18 % Se si procede all'indietro, cioè supponendo y(n+1)=0 per n grande
       e calcolando da questo y(n), si
19 % riesce a calcolare tutta la successione fino a y(1) :
20
21 y(1000)=0;
22
23 for n=998:-1:0
24   y(n+1)=(1-y(n+2))/(n+1);
25 end;
26
27 plot(y,'r-');
```