Laboratorio
di
Calcolo Numerico

Luca De Franceschi

Università degli studi di Padova

Indice

Elenco dei listati

Elenco delle figure

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.
Se facciamo eseguire questi algoritmi da un calcolatore per il quale la libreria BLAS è stata ottimizzata, otteniamo risultati di questo tipo:
e dunque, formulazioni identiche dal punto di vista matematico possono portare a performance molto differenti dal punto di vista delle prestazioni di calcolo. Notare che, in questo esempio, il numero di operazioni in virgola mobile è sempre uguale a $2 * n^3$ in tutti e tre i casi.

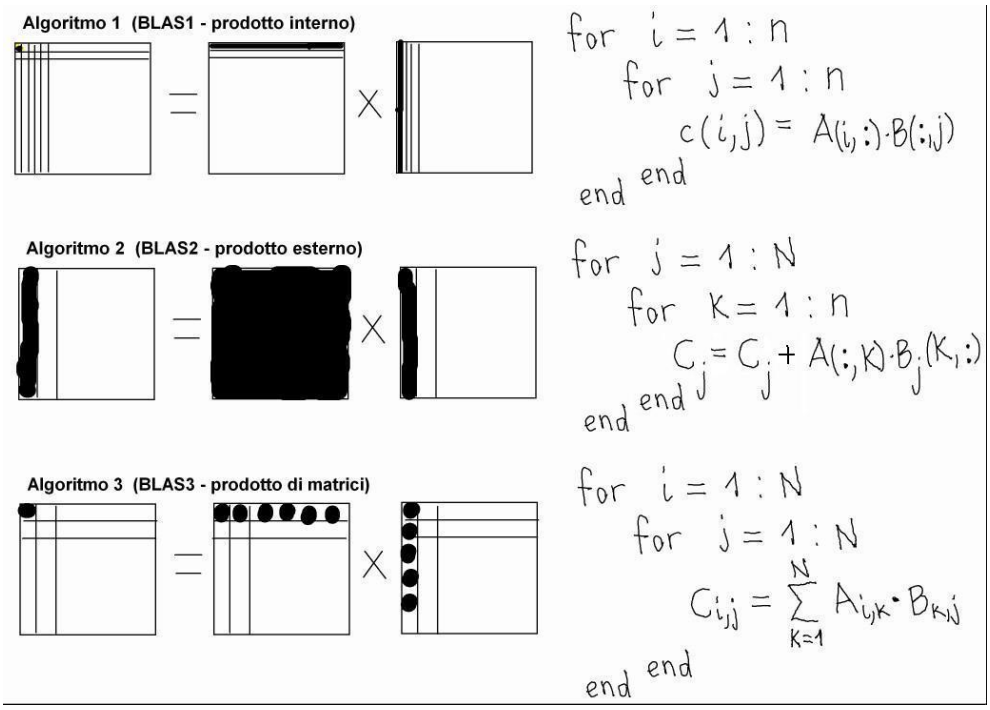


Figura 1.1: Algoritmi BLAS

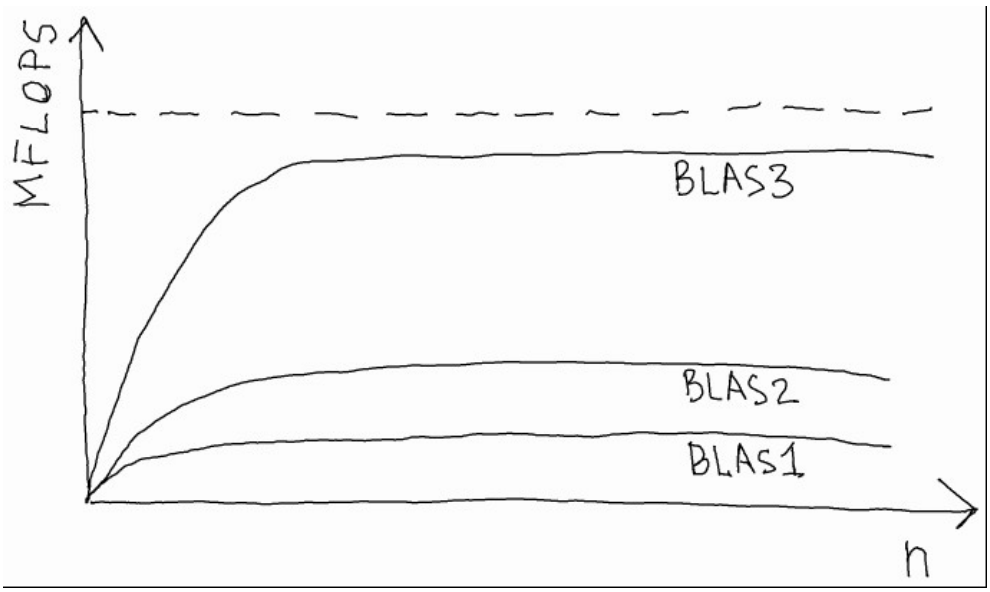


Figura 1.2: Prestazioni BLAS

Questo risultato ha validità generale: le operazioni BLAS3 sono molto più veloci delle BLAS2 e BLAS1 nelle macchine di calcolo di ultima generazione, e dunque vanno utilizzate il più possibile negli algoritmi di ALN per avere buone

prestazioni dal calcolatore.

1.2 Calcolo delle prestazioni ottenibili dalle BLAS

Vediamo di spiegare perchè le routines BLAS3 sono più performanti ed in quali condizioni. Una caratteristica sempre più frequente dei problemi di ALN che sorgono nelle applicazioni reali è quella di avere matrici di grandi dimensioni, e dunque una criticità nasce dal punto di vista dell'occupazione di memoria. Vediamo uno schema dell'organizzazione gerarchica della memoria nei calcolatori attuali ($d_s, d_p ed_c$ indicano il tempo necessario al trasferimento di un'unità di dati):

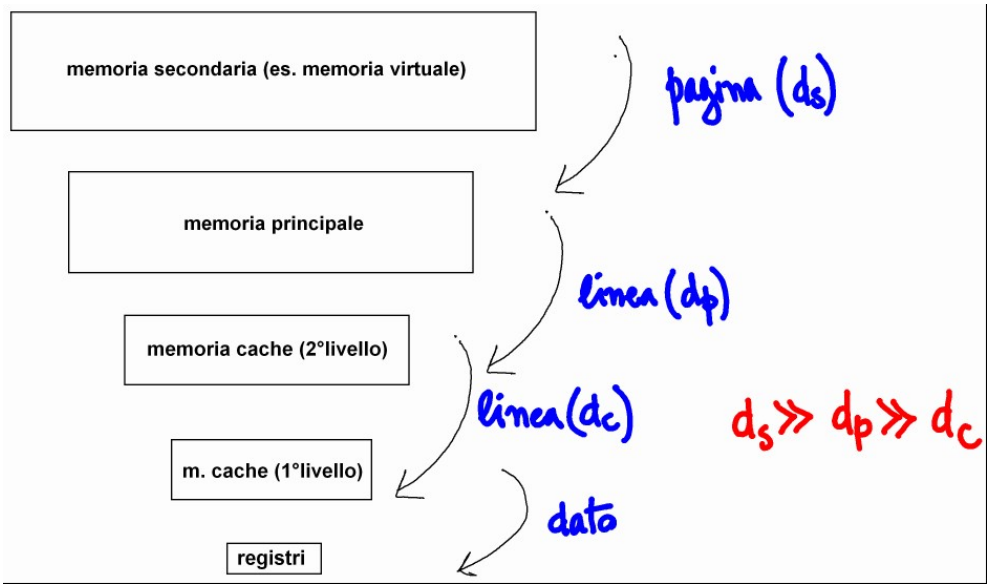


Figura 1.3: Organizzazione memorie

Lo sviluppo tecnologico delle CPU moderne ha portato a richiedere tempi ridottissimi per l'esecuzione di un'operazione in virgola mobile; non così è stato per la velocità di trasferimento dei dati tra i livelli di memoria: dato che per eseguire un'operazione in virgola mobile la CPU deve poter caricare gli operandi dalla memoria veloce (cache) nei suoi registri e che questa, per ragioni tecnologiche e di costo, ha dimensioni limitate, la criticità maggiore per le prestazioni sul tempo di calcolo è mantenere impegnata la CPU. In piccola parte il microprocessore ed il compilatore hanno dei mezzi per mascherare i tempi di trasferimento dei dati, ma per ottenere un buon risultato è necessario che l'algoritmo di calcolo e la sua implementazione siano progettati in modo che:

$$Noperazionif.p. >> Ntrasferimentididati$$

Esiste una misura precisa ed un valore massimo di velocità raggiungibile dalle prestazioni di calcolo. La velocità massima di calcolo raggiungibile da una CPU dipende dalla frequenza di clock, dal numero di cicli di clock necessari ad

eseguire un'operazione in virgola mobile (floating-point) e da quante unità hardware per il calcolo f.p. sono presenti. Quindi la velocità massima può essere calcolata dai dati di targa della macchina. Se il programma prevede una sequenza (lunga) di operazioni in virgola mobile senza istruzioni di altro tipo e se il processore trova sempre disponibili gli operandi nella memoria veloce, il calcolo procede alla massima velocità possibile.
Supponiamo $m = mA + mB + mC$, cioè scomposto nella somma degli accessi alla matrice A, alla matrice B ed alla matrice C e calcoliamone il valore. Ispezionando l'implementazione dell'algoritmo, è possibile determinare il numero di accessi effettuati (vedere figura seguente):

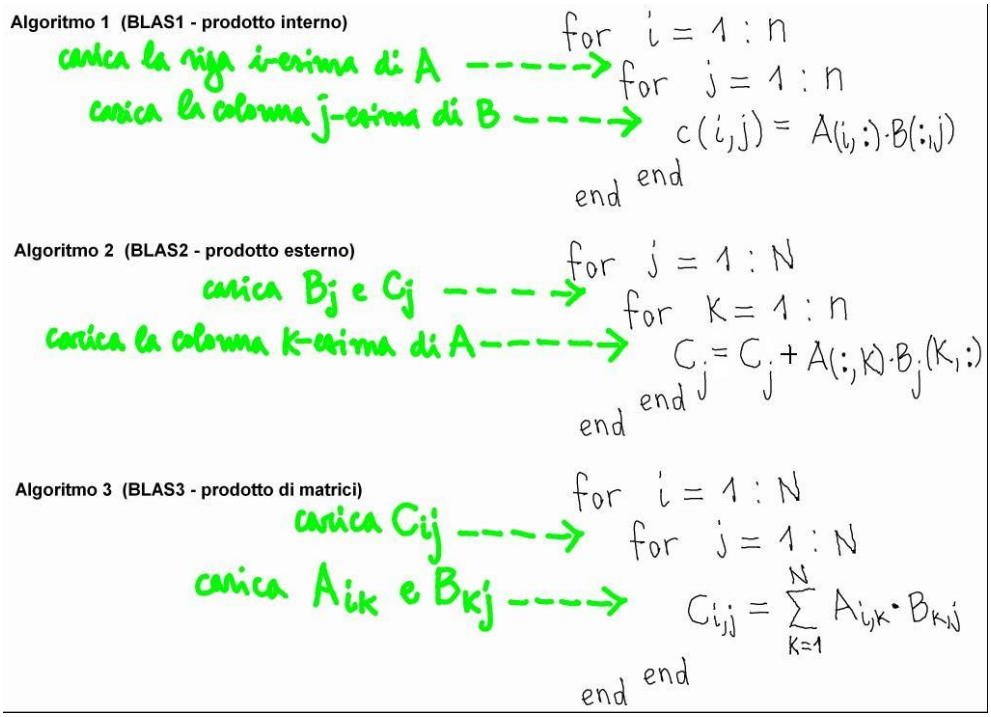


Figura 1.4: Accessi alla memoria per prodotto di matrici

1.3 Ottimizzazione empirica automatizzata della libreria software BLAS: il progetto ATLAS

Recentemente è stato portato a termine un progetto molto interessante riguardo alle BLAS: la costruzione di un software che durante la sua esecuzione, genera la libreria BLAS ottimizzata per la macchina su cui è in esecuzione. Questo software è disponibile gratuitamente al sito : www.netlib.org/atlas (attenzione: la sua esecuzione richiede in media due-tre ore su un PC, però funziona in modo completamente automatico e sicuro). Il suo successo è notevole, dato che riesce ad ottenere ottimizzazioni delle librerie tali da raggiungere le prestazioni ottenute dalle librerie commerciali, ottimizzate "a mano" da esperti. Provandolo su un PC e confrontandone le prestazioni con le BLAS "generali" scaricate da www.netlib.org/blas si può

notare che che la versione "atlas" delle BLAS è decine di volte più veloce delle BLAS "generali", già per una moltiplicazione di matrici di dimensione 1000. Questo dimostra come le prestazioni di calcolo possono essere ottenute solo sfruttando in maniera sapiente le risorse hardware a disposizione, che possono variare considerevolmente anche solo restando nel mondo del PC (es. un parametro fondamentale è la dimensione della memoria cache).

1.4 Esercizio 1

Costruire un programma in linguaggio Matlab/Octave che esegua il calcolo del prodotto di due matrici A e B di dimensione n x n, utilizzando la singola istruzione $C = A * B$, per i valori di n presi tra 10 e 2350 con passo=10, calcoli il tempo di esecuzione di ogni prodotto e faccia il grafico "$(n, tempo)$". Commentare l'andamento del grafico.
NOTA: *il tempo di esecuzione, su un PC con processore a 1.8GHz, risulta essere pari a 23 sec per l'ultima moltiplicazione (n=2350) e pari a 22 minuti per il programma complessivo; si consiglia di fare alcune prove con valori di "n" bassi ed eventualmente fermarsi ad n ¡ 2350.*

Listato 1.1: Prodotto di due matrici con singola istruzione

```
dimensioniMatrice = 10:10:1000;   % Dichiaro un vettore contenente
    tutte le dimensioni delle matrici. Si parte da 10 e si arriva a
     2350 con passo 10

tempi = zeros(length(dimensioniMatrice)); % dichiaro un vettore di
    tempi grande tanto quanto la dimensione del vettore
    dimensioniMatrice

i = 1;            % Inizializzo il contatore per l'array dei tempi

for n = dimensioniMatrice % Faccio partire il ciclo iterando sul
    vettore delle dimensioni
  A = rand(n,n);      % Inizializzo una matrice quadrate n*n con
      valori casuali compresi tra 0 e 1
  B = rand(n,n);      % Inizializzo una matrice quadrata n*n con
      valori casuali compresi tra 0 e 1
  tic;          % Faccio partire il timer
  C = A*B;        % Moltiplico le due matrici quadrate
  tempi(i) = toc;     % Fermo il timer e registro il tempo sul
      vettore dei tempi
  disp(['n =' num2str(n) ' elapsed time = ' num2str(tempi(i))]);
       % Mostro sullo schermo i tempi
  i++;          % Incremento il contatore
end

xlabel('Dimensione n della matrice');     % Imposto l'etichetta
    dell'asse x
ylabel('Tempo espresso in secondi');      % Imposto l'etichetta
    dell'asse y
plot(dimensioniMatrice, tempi, 'g');              % Mostro il
    grafico
```

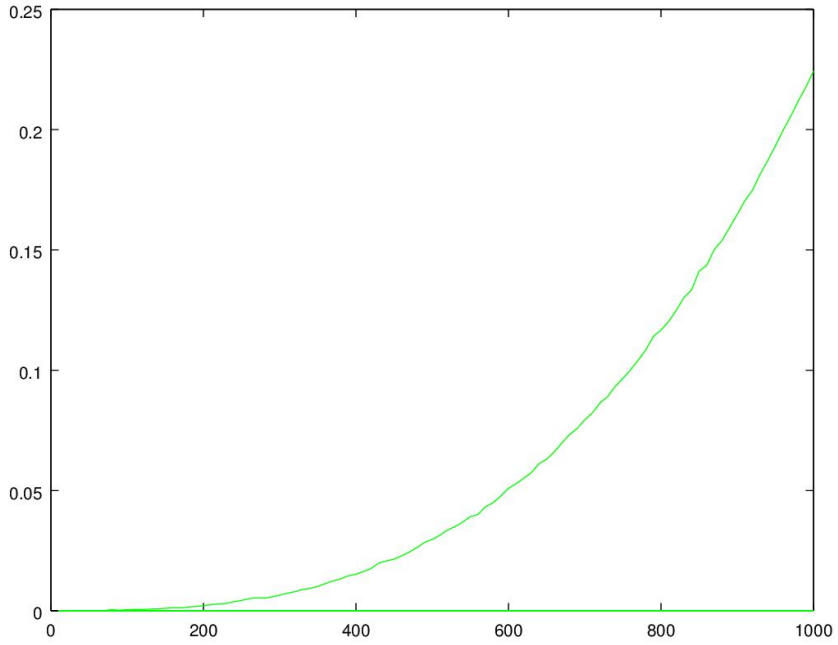


Figura 1.5: Grafico esercizio 1, dimensione matrice in funzione del tempo

1.5 Esercizio 2

Costruire un programma in linguaggio Matlab/Octave che, date due matrici A e B di dimensione nxn, nei casi $n = 200, 800, 3200, 4000, 4800$, calcoli il prodotto "per componenti" della prima riga di A e di B utilizzando una sola istruzione, ne calcoli il tempo di esecuzione t_riga, e poi faccia lo stesso con la prima colonna.
Il programma deve poi creare un grafico "$(n, t_col/t_riga)$". Spiegare con un breve commento perchè questa differenza nei tempi di esecuzione.
[suggerimento: l'estensione Numeric per Python è scritta in linguaggio C e quindi le matrici vengono memorizzate in un vettore in cui vengono inserite per righe. Notare che in Matlab, come in FORTRAN, le matrici vengono memorizzate per colonne].
NOTA: *il tempo di esecuzione, per n¿=4000 potrebbe diventare sostanzialmente lungo: fare attenzione e, a quel punto, fermarsi.*

Listato 1.2: Prodotto di due matrici per componenti

```
1 dimensioniMatrice = [200, 800, 3200, 4000, 4800];
2
3 % [200, 800, 3200, 4000, 4800];
4
5 t_riga = zeros(size(dimensioniMatrice));
6 t_col = zeros(size(dimensioniMatrice));
7
```

```
i = 1;
for n = dimensioniMatrice
  A = rand(n,n);
  B = rand(n,n);

  tic
  C = A(1,:).*B(1,:);
  t_riga(i) = toc;

  tic
  C = A(:,1).*B(:,1);
  t_col(i) = toc;
  i++;
end

plot(dimensioniMatrice, t_col./t_riga);

xlabel('Dimensione matrice');
ylabel('Rapporto tra tempi di calcolo');
```

1.6 Esercizio 3

Costruire un programma in linguaggio Matlab/Octave che esegua il calcolo del prodotto di due matrici A e B di dimensione n x n e calcoli i singoli tempi di esecuzione, utilizzando:

- L'istruzione Matlab/Octave C=A*B;
- Un algoritmo scritto in Matlab/Octave che usi operazioni BLAS3 (vedi sopra), chiamando N3 il numero di blocchi quadrati in cui viene suddivisa la matrice;
- Un algoritmo scritto in Matlab/Octave che usi operazioni BLAS2 (vedi sopra), chiamando N2 il numero di blocchi di colonne in cui viene suddivisa la matrice;
- Un algoritmo scritto in Matlab/Octave che usi operazioni BLAS1 (vedi sopra).

Provare il programma per un valore di n più elevato possibile e che sia una potenza di 2, diciamo 2^j (cominciare da $n = 2^9 = 512$) e per $N2, N3 = 2^k$ per $k = 1, 2, ..., j$.
Spiegare in qualche riga di commento le differenze nei tempi di esecuzione.

Listato 1.3: Prodotto di matrici con 4 algoritmi

```
dimensioneMatrice = 10;

A = rand(dimensioneMatrice,dimensioneMatrice);
B = rand(dimensioneMatrice,dimensioneMatrice);

A
B
A*B

tempi = [0 0 0 0];
```

```
11
12 tic
13 C = A*B;
14 tempi(1) = toc;
15
16 N3 = dimensioneMatrice*dimensioneMatrice;
17
18 tic
19 for i = 1:dimensioneMatrice
20   for j = 1:dimensioneMatrice
21     C(i,j) = A(i,:)*B(:,j);
22   end
23 end
24 tempi(2) = toc;
25
26 tempi
```

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:

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

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:

2.4 Esempio 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 2.4: 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
```

2.5 Esempio 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 2.5: 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;

semilogy([3:101],errorez(3:101),'g-')

hold off
```

2.6 Esempio 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 2.6: Successioni calcolabili

```
clear y;

% questa successione esplode in avanti ed è stabile all'indietro

y(1)=1/exp(1)*(exp(1)-1);

for n=1:100
  y(n+1)=1-(n+1)*y(n);
end;

```

```
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-');
```

3 Esercitazione 3

Spesso ci sono situazioni in cui guardando certe proprietà del problema numerico da risolvere, si possono trovare degli accorgimenti che risultano avere un impatto non trascurabile sul costo computazionale dell'algoritmo di calcolo. Vediamo di seguito un paio di esempi di questo tipo:

3.1 Prodotto matrice-vettore

Supponiamo che sia necessario compiere il prodotto matrice-vettore $A * b$, in cui la matrice "A", di dimensione n x n, è un campionamento della funzione:

$$f(x, y) = cos(x) * cos(y)$$

in una griglia cartesiana di punti nell'intervallo $[0, pi]x[0, pi]$, e "b" è un vettore colonna di lunghezza n. Scelti:

- Un vettore riga di coordinate "x" avente n elementi scelti nell'intervallo $[0, pi]$, e calcolato il vettore "vx" tale che $vx(i) = cos(x(i))$;
- Un vettore riga di coordinate "y" avente n elementi scelti nell'intervallo $[0, pi]$, e calcolato il vettore "vy" tale che $vy(i) = cos(y(i))$;

allora la matrice "A" può essere espressa nella forma:

$$A = vx' * vy$$

dove l'apice significa "trasposto". In generale, essendo A una matrice densa, il prodotto $A * b$ per un generico vettore "b" ha un costo dell'ordine di $O(n^2)$. Se, però, utilizziamo l'espressione con cui viene costruita "A" ed invertiamo la precedenza nei calcoli:

$$vx' * (vy * b)$$

otteniamo una complessità dell'ordine di $O(n)$.

Esercizio: costruire un programma Matlab/Octave che esegua il prodotto matrice-vettore $A * b$ nelle due modalità qui prospettate e verifichi automaticamente che il risultato del prodotto nei due casi è il medesimo. Inoltre, verificare sperimentalmente, per alcuni valori di n, se l'andamento dei tempi di esecuzione segue l'andamento della complessità qui prospettato e costuire un grafico $(n, tempi)$.

Listato 3.7: Prodotto matrice-vettore

```
1 n = 1:1:1000;
2
3 tempi1 = zeros(1,n);
4 tempi2 = zeros(1,n);
5
6 j = 1;
7
8 for i = n
9
```

```
10    b = rand(i,1);       % dichiaro un vettore colonna di dimensione n
11
12    x = (pi-0).*rand(1,i);     % vettore riga di n elementi con valori
          compresi tra 0 e pi
13    y = (pi-0).*rand(1,i);     % vettore colonna
14
15    vx = cos(x);  % creo i due vettori
16    vy = cos(y);
17
18    A = vx' * vy; % riempo la matrice A
19
20    tic
21
22    ris1 = A*b;   % eseguo il prodotto tra la matrice A e il vettore
          b nel primo modo
23
24    tempi1(j) = toc;
25
26    tic
27
28    ris2 = vx' .* (vy*b); % eseguo il prodotto tra la matrice A e il
          vettore b nel secondo modo
29
30    tempi2(j) = toc;
31
32    j++;
33
34  end
35
36  plot(n,tempi2, 'r', n, tempi1, 'b');
37  grid on;
38  xlabel('n');
39  ylabel('t');
40  title('Andamento del grafico con il secondo algoritmo');
41  legend('Algoritmo 1', 'Algoritmo 2');
```

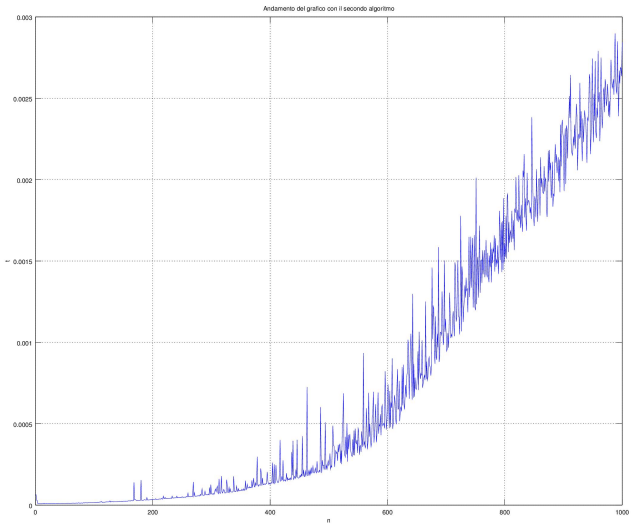


Figura 3.1: Andamento dei tempi nel caso 1

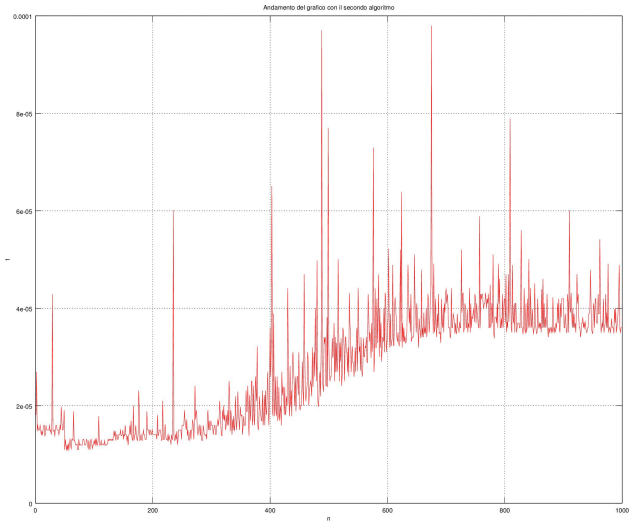


Figura 3.2: Andamento dei tempi nel caso 2

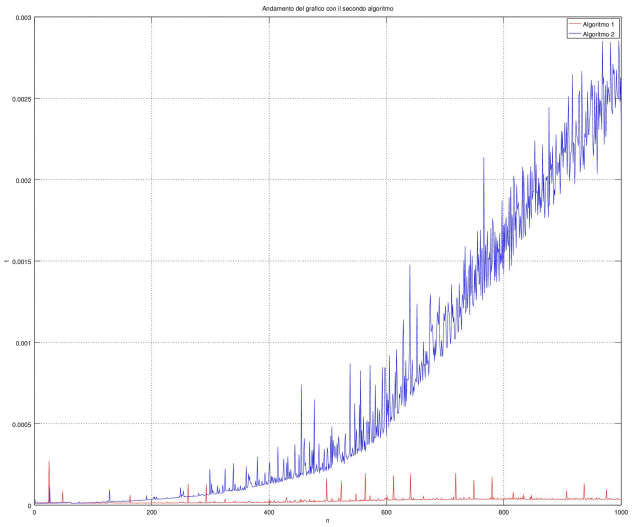


Figura 3.3: Confronto degli andamenti dei due algoritmi

3.2 Potenze di matrici

Per calcolare la potenza q-esima, $q \geq 1$, di una matrice "A", si può pensare di moltiplicare "A" q-1 volte per se stessa.
In modo più efficiente, possiamo esprimere "q" in base-2 (cioè in formato binario), calcolarci i termini $A^{(2^i)}$, $i = 1, ..., \log 2(q) - 1 = \log q / \log 2 - 1$ mediante la:

$$A^{(2^i)} = A^{(2^{(i-1)})} * A^{(2^{(i-1)})}$$

e sfruttare il fatto che:

$$A^{(c1+c2+c3)} = A^{c1} * A^{c2} * A^{c3}$$

Esercizio: costruire un programma Matlab/Octave che, data una generica matrice "A" ed assegnato un valore di "q", esegua il calcolo di A^q nelle due modalità qui esposte e verifichi automaticamente che il risultato nei due casi è il medesimo. Inoltre, valutare dal programma qual'è la complessità dei due algoritmi e verificare sperimentalmente, con una matrice "A" a scelta e valori di $q = 4, 5, ...15$, se l'andamento dei tempi di esecuzione segue l'andamento della complessità atteso. Costuire un grafico $(n, tempi)$.

Listato 3.8: Potenze di matrici

```
1 q = 4:1:15;    % valore dell'esponente
2
3 % primo metodo
4
5 n = 100;  % definisco la dimensione della matrice
6 tempi1 = zeros(size(q));
```

```
7  j = 1;
8
9  for i = q
10   A = rand(n);  % creo una matrice con valori casuali
11
12   tic
13   ris = A*A;
14
15   for k = i-1
16     ris = ris*A;
17   end
18
19   tempi1(j) = toc;
20
21   tic
22
23   toc
24
25   j++;
26
27 end
28
29 tempi1
```

4 Esercitazione 4

4.1 Implementazione dei metodi diretti per sistemi lineari: fattorizzazione LU; pivoting

Il metodo di eliminazione gaussiana, per la risoluzione di un sistema di equazioni lineari, si basa sull'idea di ridurre il sistema $A * x = b$ in un sistema equivalente (cioè avente soluzione identica) della forma $U * x = y$, dove U è una matrice triangolare superiore, e di risolvere questo sistema mediante sostituzioni all'indietro. Per rendersene conto, leggere ed eseguire il seguente m-file:

Listato 4.9: Eliminazione Gaussiana

```
A = [4 2 0.1 1.1; 1.2 5 3 0.3; 1.1 0.8 4 2.1; 0.3 1.2 3 5];
b = [1; 1; 1; 1];
A, b, disp('A e B iniziali (premi un tasto per continuare)'), pause

n = size(A,1);
for k=1:n-1
  for j=k+1:n
    m = A(j,k)/A(k,k);  % elemento pivotale
    for i=k:n
      A(j,i) = A(j,i) - m*A(k,i);
    end
    b(j) = b(j) - m*b(k);
  end
  A, b, disp('(premi un tasto per continuare)'), pause
end

% metodo delle sostituzioni all'indietro
x(n,1) = b(n)/A(n,n);
for j=n-1:-1:1
  x(j,1) = ( b(j) - A(j,j+1:n)*x(j+1:n,1) ) / A(j,j);
end;
x
A*x-b  % verifica
```

Vediamo con un esempio come l'algoritmo appena mostrato sia equivalente ad una fattorizzazione $A = L * U$, seguita dalla soluzione dei due sistemi triangolari (rispettivamente inferiore e superiore) $L * y = b$ e $U * x = y$. Infatti, consideriamo il sistema $A * x = b$, e:

- Sostituiamo ad A la sua fattorizzazione $L * U$, ovvero vale $A = L * U$, e quindi il sistema diventa: $L * U * x = b$;
- Poniamo $U * x = y$, che è un sistema lineare a matrice triangolare superiore, da cui risulta $L * y = b$, che è un sistema lineare a matrice triangolare superiore;
- Ci ricaviamo y, conoscendo L e b, risolvendo il sistema lineare $L * y = b$ mediante semplici sostituzioni in avanti;
- Ci ricaviamo x, conoscendo U ed y, risolvendo il sistema lineare $U * x = y$ mediante semplici sostituzioni all'indietro.

Leggere ed eseguire il seguente m-file:

Listato 4.10: Esempio di fattorizzazione LU

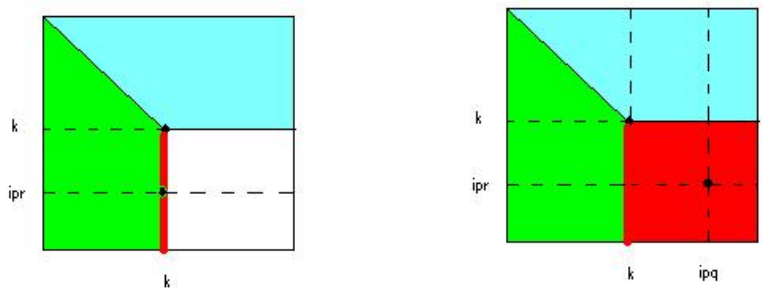
```
% consideriamo la stessa matrici dell'esempio precedente:

A1 = [4 2 0.1 1.1; 1.2 5 3 0.3; 1.1 0.8 4 2.1; 0.3 1.2 3 5];
A2 = [4 2 0.1 1.1; 1.2 5 3 0.3; 1.1 0.8 4 2.1; 0.3 1.2 3 5];
b = [1; 1; 1; 1];
A1, b, disp('A e b iniziali (premi un tasto per continuare)'),
    pause

n = size(A1,1);
prodM=eye(n);
for k=1:n-1
  for j=k+1:n
    m(j,1) = A1(j,k)/A1(k,k);  % elemento pivotale
    for i=k:n
      A1(j,i) = A1(j,i) - m(j,1)*A1(k,i);
    end
  end
  M=eye(n); M(k+1:n,k) = - m(k+1:n,1);
  A2 = M * A2;
  A1, A2, disp('(premi un tasto per continuare)'), pause
  prodM = M * prodM;
end

U = A1;
U
disp('U  il fattore triangolare superiore'), pause

% se prendiamo il prodotto di tutte le matrici "M" cos calcolate e
    lo
% invertiamo, otteniamo il fattore triangolare "L" . Infatti,
    chiamando
% M_i la matrice "M" creata alla i-esima iterazione, l'algoritmo
% pre-moltiplica la matrice "A" per "M_i" fino ad ottenere una
    matrice triangolare
% superiore "U", cio:  M_(n-1)*...*M_1*A = prodM*A = U e quindi,
    pre-moltiplicando
% entrambi i menbri di questa espressione per l'inversa di prodM,
% inv(prodM), si ha  inv(prodM)*prodM*A = I*A = A = inv(prodM)*U,
    da cui
% "L = inv(prodM)".
L = inv(prodM);
L
disp('L  il fattore triangolare infriore (notare che ha tutti "1"
    sulla diagonale)'), pause

A3=L*U;
A3
disp('verifica: L*U=A '), pause

% soluzione del sistema L*y=b
y(1,1) = b(1)/L(1,1);
for i=2:n
  y(i,1) = ( b(i) - L(i,1:i-1)*y(1:i-1,1) ) / L(i,i);
end;

% soluzione del sistema U*x=y
x(n,1) = y(n)/U(n,n);
for j=n-1:-1:1
```

```
52    x(j,1) = ( y(j) - U(j,j+1:n)*x(j+1:n,1) ) / U(j,j);
53  end;
54  x
55  A3*x-b  % verifica
```

Esistono varie implementazioni possibili della fattorizzazione LU. Ad esempio le implementazioni degli algoritmi “kji” e “ijk” sono contenute nel codice seguente:

Listato 4.11: Esempio di fattorizzazione LU

```
1   %  Fattorizzazione LU "sul posto" della matrice A (A=L*U) con la
        sequenza "kji"
2   %
3   %  "sul posto" significa che i fattori L ed U sono ancora contenuti
        in A .
4   %
5   function [A] = fatt_LU_kji(A)
6   n=size(A,1);
7   for k=1:n-1
8     A(k+1:n,k)=A(k+1:n,k)/A(k,k);
9     for j=k+1:n,
10      for i=k+1:n
11        A(i,j)=A(i,j)-A(i,k)*A(k,j) ;
12      end,
13    end
14  end
15
16
17
18  %  Fattorizzazione LU "sul posto" della matrice A (A=L*U) con la
        sequenza "ijk"
19  %
20  %  "sul posto" significa che i fattori L ed U sono ancora contenuti
        in A .
21  %
22  function [A] = fatt_LU_ijk(A)
23  n=size(A,1);
24  for i=1:n
25    for j=2:i
26      A(i,j-1) = A(i,j-1)/A(j-1,j-1);
27      for k=1:j-1
28        A(i,j)=A(i,j)-A(i,k)*A(k,j) ;
29      end
30    end
31    for j=i+1:n
32      for k=1:i-1
33        A(i,j)=A(i,j)-A(i,k)*A(k,j) ;
34      end
35    end
36  end
37
38
39
40  %  Fattorizzazione di Cholesky "sul posto" della matrice A: A = L *
        L'
41  %
42  %  "sul posto" significa che il fattori L e' ancora contenuto in A
        .
43  %
44  function [A] = fatt_Cholesky(A)
```



(a) Pivotint parziale sulle righe (b) Pivoting totale

Figura 4.1: Tecnica del Pivoting

```
n=size(A,1);
for k=1:n-1
  A(k,k)=sqrt(A(k,k));
  A(k+1:n,k)=A(k+1:n,k)/A(k,k);
   for j=k+1:n
     A(j:n,j)=A(j:n,j)-A(j:n,k)*A(j,k);
   end
end
A(n,n)=sqrt(A(n,n));
A=tril(A);
```

.

4.2 Esercizio 1

Guardare il codice sorgente delle due fattorizzazioni “kji” e “ijk” nel codice appena citato. Sapreste dire quale dei due accede per righe e quale per colonne?
Risposta:

4.3 La tecnica del pivoting

Per evitare di trovarsi un elemento pivotale nullo o molto piccolo, entrambe situazioni da evitare, in certe matrici è necessario applicare la tecnica del “pivoting”, che vediamo spiegata nella figura seguente, rispettivamente per il pivoting parziale sulle righe (sinistra) e per il pivoting totale (destra):
dove l’elemento (k, k) è il pivot “naturale”, mentre (ipr, k) e (ipr, ipq) sono quelli determinati per ricerca del massimo valore nelle rispettive zone colorate di rosso:

- Per il pivoting parziale per righe la zona in rosso è la porzione di colonna di indici $(k : n, k)$;
- Per il pivoting totale la zona in rosso è la sotto-matrice di indici $(k : n, k : n)$.

Una volta determinato l'elemento pivotale si procede allo scambio di righe/colonne:

- Si scambiano le righe "k" ed "ipr" nel pivoting parziale per righe;
- Si scambiano le righe "k" ed "ipr" e le colonne "k" ed "ipq" nel pivoting totale;

Leggere ed eseguire il seguente m-file che implementa il pivoting parziale (sulle righe):

Listato 4.12: Pivoting parziale

```
% Fattorizzazione LU "sul posto" della matrice A (A=L*U) con la sequenza "kji"
% e pivoting su righe
%
Aini = [1 2 0.1 1.1; 1.2 5 3 0.3; 1.1 0.8 0.5 2.1; 0.3 1.2 3 5];
Aini, disp('A iniziale (premi un tasto per continuare)'), pause
A = Aini;
n=size(A,1);
p = 1:n; % "p" il vettore degli indici permutati
for k=1:n-1
  [Y I]=max(abs(A(k:n,k))); % cerco il massimo pivot sulla k-esima colonna
  ip = k + I(1)-1; % "ip" l'indice-riga del massimo pivot
  if ip>k % il massimo pivot non sta sulla diagonale, quindi faccio la permutazione
    temp_subriga=A(k,:); A(k,:)=A(ip,:); A(ip,:)=temp_subriga;
    temp_i=p(k);         p(k)=p(ip);     p(ip)=temp_i;
  end
  A(k+1:n,k)=A(k+1:n,k)/A(k,k); % calcolo una sub-colonna in un' unica operazione
  for j=k+1:n,
    for i=k+1:n
      A(i,j)=A(i,j)-A(i,k)*A(k,j) ;
    end,
  end
end

P = zeros(n); % "P" la matrice di permutazione, che andiamo a costruire tramite il vettore delle permutazioni "p"
for i=1:n  P(i,p(i))=1; end;
disp('P = ');
P
disp(' ');
disp(' ');
disp('verifica: P*A = L*U');
disp(' ');
disp('P*A = ');
P*Aini
disp('L*U = ');
L = eye(n)+tril(A,-1);
U = triu(A,0);
L*U
```

4.4 Esercizio 2

Scrivere un programma Matlab/Octave che esegua la fattorizzazione LU con il pivoting totale e provarlo su una matrice piccola (ad es. 5x5, in modo da

poterla visualizzare facilmente). Come verifica della correttezza dell'algoritmo implementato nel programma, si può verificare che $sum(sum(abs(A - L * U)))$ dia un valore molto piccolo.

Listato 4.13: Fattorizzazione LU con pivoting totale