# Algebra lineare numerica in Matlab

Alvise Sommariva

Università degli Studi di Padova
Dipartimento di Matematica Pura e Applicata

17 gennaio 2019

# Fattorizzazione LU

Il comando `lu` calcola la corrispettiva fattorizzazione di una matrice A con $n$ righe e colonne, ottenuta mediante eliminazione di Gauss con pivoting.

Il comando è

```
[L,U,P]=lu(A)
```

e determina le matrici

- $L = (l_{i,j}) \in \mathbb{R}^{n\times n}$ triangolare inferiore, ovvero tale che $l_{i,j} = 0$ se $j < i$ con $l_{i,i} = 1$,
- $U = (u_{i,j}) \in \mathbb{R}^{n\times n}$ triangolare superiore, ovvero tale che $u_{i,j} = 0$ se $j > i$,
- $P = (p_{i,j}) \in \mathbb{R}^{n\times n}$ di permutazione, ovvero con esclusivamente un valore non nullo e pari a 1 per ogni riga e colonna,

cosicchè $PA = LU$.

# Fattorizzazione LU

Vediamo un esempio.

```
>> A=[4 -2 -1 0; -2 4 1 0.5; -1 1 4 1; 0 0.5 1 4];
>> [L,U,P]=lu(A);
>> % L triang. inf. con elementi diagonali uguali a 1
>> L
L =
    1.0000         0         0         0
   -0.5000    1.0000         0         0
   -0.2500    0.1667    1.0000         0
         0    0.1667    0.2500    1.0000
>> % L triangolare sup.
>> U
U =
    4.0000   -2.0000   -1.0000         0
         0    3.0000    0.5000    0.5000
         0         0    3.6667    0.9167
         0         0         0    3.6875
>> % P di permutazione
>> P
P =
     1     0     0     0
     0     1     0     0
     0     0     1     0
     0     0     0     1
>>
```

# Soluzione di sistemi lineari

Supponiamo sia

- $A \in \mathbb{R}^{n\times n}$, con $\det(A) \neq 0$,
- $b \in \mathbb{R}^n$,

In tali ipotesi, esiste un unico $x^* \in \mathbb{R}^n$ che risolva il sistema lineare $Ax = b$.
Sotto queste ipotesi si può provare che se $PA = LU$, allora necessariamente

- $\det(P) = \pm 1$, $P^{-1} = P^T$,
- $\det(L) = 1$
- $\det(U) \neq 0$.

Quindi, posto $Pb = c$, abbiamo

$$Ax = b \Leftrightarrow PAx = Pb \Leftrightarrow LUx = Pb = c$$

Di conseguenza

- posto $y = Ux$, $y$ è soluzione del sistema triangolare inferiore $Ly = c$;
- calcolato $y$, la soluzione $x$ del sistema $Ax = b$ è pure soluzione del sistema triangolare superiore $Ux = y$;

I due sistemi lineari

$$Ly = c, \quad Ux = y$$

possono essere convenientemente essere risolti rispettivamente mediante sostituzione in avanti e all'indietro in $O(n^2)$ operazioni moltiplicative.

# Soluzione di sistemi lineari

L'ambiente Matlab utilizza questa e altre strategie per risolvere i sistemi lineari. A tal proposito, digitando "`help \`" ricaviamo

```
>> help \
 \    Backslash or left matrix divide.
    A\B is the matrix division of A into B, which is roughly the
    same as INV(A)*B , except it is computed in a different way.
    If A is an N-by-N matrix and B is a column vector with N
    components, or a matrix with several such columns, then
    X = A\B is the solution to the equation A*X = B. A warning
    message is printed if A is badly scaled or nearly singular.
    A\EYE(SIZE(A)) produces the inverse of A.
    ...
    See also ldivide, rdivide, mrdivide.

    Reference page for mldivide
    Other functions named mldivide
>>
```

In pratica tale comando serve per poter risolvere sistemi lineari del tipo $Ax = b$.

# Soluzione di sistemi lineari

Vediamo un esempio.

```
>> A=[1 3 5; 2 4 5; 1 1 1];
>> % verifichiamo che det(A) non e' nullo
>> % (ovvero la matrice A e' invertibile)
>> det(A)
ans =
    -2
>> % definiamo il termine noto "b"
>> b=[1 1 1]';
>> % calcoliamo la soluzione di A*x=b;
>> % attenzione che e' "\" e non "/".
>> x=A\b
x =
     2
    -2
     1
>> A*x
ans =
     1
     1
     1
>> % Quindi visto che "b" e' il vettore colonna [1 1 1]'
>> % abbiamo che "x" e' la soluzione richiesta.
```

# Metodi iterativi

Si supponga di dover risolvere il sistema lineare $Ax = b$ con $A \in \mathbb{R}^{n\times n}$, $\det(A) \neq 0$, $b \in \mathbb{R}^n$.

Per una matrice $A$ in generale il metodo di eliminazione gaussiana richiede $O(n^3/3)$ operazioni, calcolando la soluzione esatta.

Qualora

- si sia interessati a una approssimazione della soluzione esatta, ad esempio con un certo numero di cifre decimali esatte, e magari $n$ sia molto grande,
- e/o $A$ abbia molte componenti nulle,

tipicamente si utilizzano metodi iterativi, che spesso raggiungono questo risultato con complessità dell'ordine $O(n^2)$.

# Metodi iterativi: Jacobi

Un primo esempio di metodo iterativo è quello di Jacobi, che calcola una sequenza di vettori $x^{(k)}$ che in certe ipotesi si dimostra convergono alla soluzione $x^*$.

In dettaglio, se $A = (a_{i,j})$ e $a_{k,k} \neq 0$ per $k = 1, \ldots, n$,

$$x_i^{(k+1)} = (b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)})/a_{ii}, \qquad (1)$$

con $i = 1, \ldots, n$.

# Metodi iterativi: Gauss-Seidel

Un altro metodo iterativo, è quello di Gauss-Seidel, che calcola una sequenza di vettori $x^{(k)}$ che in certe ipotesi si dimostra convergono alla soluzione $x^*$.

In dettaglio, se $A = (a_{i,j})$ e $a_{k,k} \neq 0$ per $k = 1, \ldots, n$,

$$x_i^{(k+1)} = \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} \right) / a_{ii}, \qquad (2)$$

con $i = 1, \ldots, n$.

# Metodi iterativi

Risulta possibile riscrivere questi metodi in forma matriciale.
Sia

- $A \in \mathbb{R}^{n\times n}$ una matrice quadrata,
- $A = P - N$ un cosidetto splitting della matrice $A$, con $\det(P) \neq 0$

Allora consideriamo metodi le cui iterazioni siano fornite dalle iterazioni successive

$$x^{(k+1)} = P^{-1}Nx^{(k)} + P^{-1}b. \tag{3}$$

Sia $A = D - E - F$ con

1. $D$ matrice diagonale,
2. $E$ triangolare inferiore,
3. $F$ triangolare superiore.

Allora

- il metodo di Jacobi corrisponde a scegliere, se $D$ è invertibile,

$$P = D, \qquad N = E + F;$$

- il metodo di Gauss-Seidel corrisponde a scegliere, se $D$ è invertibile,

$$P = D - E, \qquad N = F.$$

# Implementazione di alcuni metodi iterativi

Implementiamo in Matlab tali routines, con la caratterizzazione matriciale.

```
function [x,errs,iter,flag]=metodo_jacobi(A,x,b,maxit,tol)

% oggetto:
% la routine risolve il sistema lineare Ax=b mediante
% il metodo di Jacobi
%
% input:
% A: matrice quadrata non singolare
% x: approssimazione iniziale della soluzione
%     (vettore colonna)
% b: termine noto (vettore colonna)
% maxit: numer massimo di iterazioni
% tol: tolleranza del metodo di Jacobi
%
% output:
% x: approssimazione della soluzione, fornita dal
%      metodo di Jacobi.
% errs: norme dell'errore
%                   norm(x_new-x_old)/norm(x_new)
%     al variare delle iterazioni, ovvero lo step
%     relativo.
% iter: numero di iterazioni del metodo
% flag: 0: si e' raggiunta l'approssimazione
%             desiderata della soluzione
%       1: non si e' raggiunta l'approssimazione
%             desiderata della soluzione
```

# Implementazione di alcuni metodi iterativi

```
% inizializzazioni
flag=0;
% --- le matrici P, N per il metodo di Jacobi ---
P=diag(diag(A));
% se il determinante di P e' uguale a "1", allora
% il metodo di Jacobi non si puo' applicare in
%  quanto qualche componente A(k,k)=0;
% in effetti P e' una matrice diagonale,
% il determinante di P e' il prodotto degli elementi
% sulla diagonale per cui e' nullo sse qualche
% A(k,k)=0;
if det(P) == 0
    errs=[]; iter=0; flag=1;  return;
end
N=diag(diag(A))-A;
% --- iterazioni del metodo di Jacobi ---
for iter=1:maxit
    x_old=x;
    x=P\(N*x_old+b);   % nuova iterazione
    % calcolo step relativo
    errs(iter)=norm(x-x_old)/norm(x);
    % se error e' suff. piccolo si esce dalla routine
    if (errs(iter)<=tol), return, end
end
% se abbiamo raggiunto questo punto abbiamo fatto
%  troppe iterazioni senza successo e quindi
% poniamo "flag=1".
flag=1;
```

# Commento a `metodo_jacobi`

Cominciamo discutendo il codice `metodo_jacobi`.

- Dapprima inizializziamo le variabili. In particolare se si esce correttamente `flag=0`, come da inizializzazione, e si assegnano le altre variabili di output.
- Di seguito, come suggerito dalla versione matriciale, definiamo $M$, $N$, facendo attenzione al caso in cui $M$ abbia qualche qualche componente diagonale nulla, ovvero non sia singolare. In tal caso assegniamo `flag=1` e usciamo dalla routine per `return`.
- Entriamo nel ciclo for che fa iterazioni fino a quando il loro indice `iter` non supera `maxit`.
- Assegna a `x_old` il valore della vecchia iterazione e calcola la nuova iterazione mediante il comando Matlab "\". Si osservi che la matrice $M$ è diagonale e quindi, una volta calcolato `N*x_old+b`, Matlab risolve il sistema in $O(n)$ divisioni (pensarci su).

# Commento a `metodo_jacobi`

- Nella riga successiva, alla $k$-sima iterazione valuta la quantità

$$\frac{\|x - x_{old}\|_2}{\|x\|_2}$$

dove al solito se $u = (u_1, \ldots, u_n)$ allora

$$\|u\|_2 = \sqrt{\sum_{i=1}^{n} u_i^2}$$

e assegna tale valore a `errs(iter)`.

- Se la quantità `errs(iter)` è minore o uguale della tolleranza `tol` esce dalla routine, altrimenti prosegue a iterare il ciclo for.
- Se esce dal ciclo for dopo `maxit` iterazioni, pone `flag=1` perchè nel numero di iterazioni richieste non ha trovato la soluzione.

# La routine demo_jacobi

Per testare il metodo salviamo la seguente demo_jacobi.

```
function demo_jacobi

% demo relativa al metodo di Jacobi, per la
% soluzione di sistemi lineari Ax=b.

% problema A*x=b
A=gallery('poisson',10); % matrice
sol=ones(size(A,1),1); % soluzione
b=A*sol; % termine noto
% preferenze
maxit=100000; % numero massimo di iterazioni
tol=10^(-6); % tolleranza
x0=zeros(size(b)); % vettore iniziale
% soluzione col metodo di Jacobi
[x,errs,iter,flag]=metodo_jacobi(A,x0,b,maxit,tol);
% norma 2 residuo relativo
err=norm(b-A*x,2)/norm(b);
% statistiche
fprintf('\n \t dimensione matrice: %6.0f', size(A,1));
fprintf('\n \t numero iterazioni : %6.0f', iter);
fprintf('\n \t flag              : %6.0f', flag);
fprintf('\n \t residuo relativo  : %1.2e \n \n', err);
```

# Commento a demo_jacobi

La demo non è troppo complicata, ma ha alcuni aspetti interessanti da discutere.

- La prima riga è

  ```
  A=gallery('poisson',10);
  ```

  calcola la matrice di Poisson $P_{10}$ che è una matrice con 100 righe e colonne. Digitando `help gallery` nella command-window, si capisce che tale gallery permette di definire molte matrici di vario interesse matematico.
- Si impone che la soluzione `sol` sia un vettore colonna che abbia lo stesso numero di righe dell'ordine della matrice, e tutte le componenti uguali a 1.
- Se `sol` è la soluzione del sistema lineare $Ax = b$, necessariamente $b = A * \text{ttsol}$.
- Quale vettore iniziale, si utilizza il vettore colonna `x`, che abbia la stessa dimensione di `b`, ma con componenti tutte nulle.
- Il metodo di Jacobi fornisce l'approssimazione `x` della soluzione `sol`.
- Di seguito si calcola la norma 2 del residuo dovuto a `x` (relativo rispetto al termine noto `b`), ovvero

$$\frac{\|b - A * \texttt{sol}\|_2}{\|b\|_2}.$$

- Alla fine si salvano alcune statistiche.

# Commento a demo_jacobi

Alcuni teoremi asseriscono che il metodo di Jacobi, genera una sequenza di vettori $\{x^{(k)}\}$ che converge alla soluzione esatta, nel nostro esempio `sol`.
In effetti, da command-window otteniamo

```
>> demo_jacobi

   dimensione matrice:     100
   numero iterazioni :     255
   flag              :       0
   residuo relativo  : 5.42e-06

>>
```

**Nota.**

*Si noti che la statistica della demo è il residuo relativo e non lo step relativo proprio delle iterazioni di*

`metodo_jacobi`

*Per questo, pure essendo lo step relativo inferiore alla tolleranza, nel nostro caso* $10^{-6}$*, il residuo relativo è maggiore (ovvero* $5.42 \cdot 10^{-6}$*).*

# Esercizi. La routine `fattorizzazione_LU`

## Esercizio (1)

*Il seguente pseudocodice*

**function** [L,A]=fattorizzazione_LU(A)

**for** *k=1,...,n-1* **do**
  $l_{k,k} = 1$ **for** *i=k+1,...,n* **do**
    $l_{i,k} = \frac{a_{i,k}}{a_{k,k}}$
    **for** *j=k,...,n* **do**
      $a_{i,j} = a_{i,j} - l_{i,k} a_{k,j}$
    **end**
  **end**
**end**
$l_{n,n} = 1$

- *calcola la fattorizzazione LU di una matrice* $A = (a_{i,j})$ *data in input,*
- *in output offre le matrici triangolari* $L = (l_{i,j})$ *e* $U = (u_{i,j})$ *che corrisponde alla matrice A alla fine del processo.*

*Lo si implementi in Matlab mediante la function* `fattorizzazione_LU` *ricordando che "n" è il numero di righe e colonne di A.*

# Esercizi. La routine `fattorizzazione_LU` e la soluzione di sistemi lineari

## Esercizio (2)

*Si scriva uno script* `demo_eliminazione_gaussiana` *che definita la matrice*

$$A = \begin{pmatrix} 2 & 1 & 0 \\ 1 & 2 & 1 \\ 0 & 1 & 2 \end{pmatrix}$$

*e il termine noto*

$$b = \begin{pmatrix} 3 \\ 4 \\ 3 \end{pmatrix}$$

*calcoli la soluzione del sistema lineare $Ax = b$, risolvendo i due sistemi triangolari citati in precedenza (con $P$ uguale alla matrice identica), mediante il comando $\backslash$ di Matlab.*
*La soluzione corretta è*

$$x = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix}.$$

# Esercizi. Le routines `metodo_GS` e `demo_GS`

**Esercizio**

*Si modifichino i files* `metodo_jacobi` *e* `demo_jacobi`, *così da definire* `metodo_GS` *e* `demo_GS`, *che al posto di Jacobi, implementino e testino il metodo di Gauss-Seidel.*

- *A tal proposito, si usi la sua descrizione matriciale, coinvolgente le matrici* $M$ *e* $N$.
- *Nello svolgimento della routine, si utilizzino i comandi Matlab* `tril` *e* `triu` *(aiutarsi con l'help di Matlab).*