# 1 Lezione del 21-03-25

## 1.1 Valutazione della riducibilità

Riprendiamo il discorso delle matrici riducibili, soffermandoci sul come capire quando una matrice è riducibile, e come ricavare, in caso affermativo, la marice di permutazione $\Pi$ corrispondente.

Vale il seguente risultato (abbastanza banale guardando a quanto detto riguardo alle matrici irriducibili):

**Teorema 1.1: Caratterizzazione di matrice riducibile**

Una matrice $A$ è riducibile quando non è irriducibile, cioè quando il suo grafo associato $G(A)$ non è fortemente connesso.

La dimostrazione avviene osservando innanzitutto questo **lemma**: se esiste una certa permutazione $\pi$ degli elementi in riga, si può ricavare una matrice di permutazione $\Pi$ e definire la matrice:

$$A' = \Pi A \Pi^T$$

Avremo allora che il grafo associato ad $A'$ sarà lo stesso associato a $\Pi A \Pi^T$, solamente cambiando i nomi dei vertici, cioè:

$$G(A) \text{ fortemente connesso } \Leftrightarrow \ G(A') \text{ fortemente connesso}$$

La **dimostrazione** vera e propria dovrà quindi affermaredovrà quindi affermare che se una matrice è rducibile, allora il suo grafo non è fortemente connesso, cioè:

$$A \text{ riducibile } \Leftrightarrow \ G(A) \text{ non fortemente connesso}$$

$\Rightarrow$) Abbiamo che se $A$ è riducibile allora $\exists \Pi$:

$$B = \Pi A \Pi^T = \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix}$$

con $A_{11} \in \mathbb{C}^{k \times k}$ e $A_{22} \in \mathbb{C}^{(n-k)\times(n-k)}$. Che $G(B)$ non è fortemente connesso è chiaro dal blocco di 0 in basso a sinistra: significherà che non ci sono archi che collegano il blocco $A_{22}$ al blocco $A_{11}$, cioè non esistono archi che vanno da $\{k+1, ..., n\}$ a $\{1, ..., n\}$.

$\Leftarrow$) Se $G(A)$ non è fortemente connesso allora $\exists (j, h)$ per cui da $j$ non si raggiunge $h$. Dividamo allra $\{1, ..., n\}$ in due sottoinsiemi:

$$\begin{cases} \mathcal{P} = \{\text{vertici raggiungibili da } j\} \\ \mathcal{Q} = \{\text{vertici non raggiungibili da } j\} \end{cases}$$

con $\mathcal{P} \cup \mathcal{Q} = \{1, ..., n\}$ e $\mathcal{P} \cap \mathcal{Q} = \emptyset$. Non ci sarà quindi nessun arco che collega un elemento di $\mathcal{P}$ a un elemento di $\mathcal{Q}$. Se si considera una permutazione $\Pi$ che manda in testa tutti gli elementi di $\mathcal{Q}$, si ritrova esattamente la forma della definizione 7.7, cioè quella di una matrice riducibile, notando $k = |\mathcal{Q}|$ e $n - k = \mathcal{P}$.

□

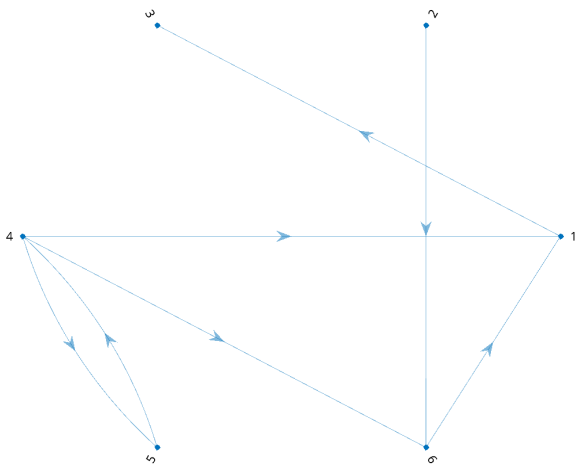
Prendiamo ad esempio la matrice:

$$A = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 3 \\ 0 & 0 & -1 & 0 & 0 & 0 \\ 3 & 0 & 0 & 2 & -1 & -2 \\ 0 & 0 & 0 & -1 & 2 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Disegnamo il grafo $G(A)$ associato alla matrice $A$. In MATLAB, questo si potrà fare come:

```
>> A = [ 1 0 1 0 0 0; ... 1 0 0 0 0 1 ]
>> A(eye(size(A)) == 1) = 0 % rimuovi gli elementi sulla diagonale
>> G = digraph(A)
>> p = plot(G)
>> layout(p, "circle") % usa il layout circolare
```

che dalla $A$ dell'esempio dà:




Vediamo quindi che le adiacenze fra nodi (pensando alla matrice, le dipendenze riga-colonna) sono:

| Nodo | Raggiungibile | Non raggiungibile |
|---|---|---|
| 1 | 1 3 | 2 4 5 6 |
| 2 | 1 2 3 6 | 4 5 |
| 3 | 3 | 1 2 4 5 6 |
| 4 | 1 3 4 5 6 | 2 |
| 5 | 1 3 4 5 6 | 2 |
| 6 | 1 3 6 | 2 4 5 |

Notiamo che il nodo 6 ci fornisce la partizione migliore $3 \times 3$ e $3 \times 3$. Decidiamo quindi di prendere:

$$\mathcal{P} = \{1, 3, 6\}, \quad \mathcal{Q} = \{2, 4, 5\}$$

per cui la permutazione che manda $\mathcal{Q}$ in testa è:

$$\Pi = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \begin{matrix} 2 \to 1 \\ 4 \to 2 \\ 5 \to 3 \\ 1 \to 4 \\ 3 \to 5 \\ 6 \to 6 \end{matrix}$$

Applicando $PiA\Pi^T$ si ottiene quindi:

$$\Pi A \Pi^T = \begin{pmatrix} A_{11} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & -1 & 0 \end{pmatrix} & A_{12} = \begin{pmatrix} 0 & 0 & 3 \\ 3 & 0 & -2 \\ 0 & 0 & 0 \end{pmatrix} \\ 0 & A_{21} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \end{pmatrix}$$

che è una forma più agile per la risoluzione della matrice originale $A$.

Nella directory `/matlab` si rende disponibile uno script `block_decomp.m` per la decomposizione in matrici a blocchi come nell'esempio. Un'esecuzione tipica dello script potrebbe avere l'aspetto:

```
p = block_decomp(A) % ottieni una permutazione
    Node       Reachable       Not reachable
    ----    -------------    -------------

     1      {[       1 3]}    {[  2 4 5 6]}
     2      {[  1 2 3 6]}    {[      4 5]}
     3      {[        3]}    {[1 2 4 5 6]}
     4      {[1 3 4 5 6]}    {[        2]}
     5      {[1 3 4 5 6]}    {[        2]}
     6      {[    1 3 6]}    {[    2 4 5]}

   Choose an index: 6 % chiesto dallo script
>> A(p, p) % permuta A
```

da cui si ottiene la stessa matrice a blocchi riportata sopra.

### 1.1.1 Riduzione iterata

Ricordiamo di poter iterare ricorsivamente il processo di riduzione, cioè di poter trovare per ogni blocco $A_{ii}$ con $i \in \{1, 2\}$ un $\Pi_i$ tale che:

$$\Pi_i A_{ii} \Pi_i^T = \begin{pmatrix} A_{11}^{(i)} & A_{12}^{(i)} \\ 0 & A_{22}^{(i)} \end{pmatrix}$$

così che valga:

$$\begin{pmatrix} \Pi_1 & 0 \\ 0 & \Pi_2 \end{pmatrix} \begin{pmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{pmatrix} \begin{pmatrix} \Pi_1^T & 0 \\ 0 & \Pi_2^T \end{pmatrix} = \begin{pmatrix} \Pi_1 A_{11} \Pi_1^t & \Pi_1 A_{12} \Pi_2^T \\ 0 & \Pi_2 A_{22} \Pi_2^t \end{pmatrix}$$

$$= \begin{pmatrix} \begin{pmatrix} A_{11}^{(1)} & A_{12}^{(1)} \\ 0 & A_{22}^{(1)} \end{pmatrix} & * \\ 0 & \begin{pmatrix} A_{11}^{(2)} & A_{12}^{(2)} \\ 0 & A_{22}^{(2)} \end{pmatrix} \end{pmatrix}$$

e via dicendo, dove in $(*)$ comparrà qualcosa che al momento non ci interessa.

### 1.1.2 Problemi agli autovalori per riduzione

Notiamo che questo procedimento semplifica anche la risoluzione dei **problemi agli autovalori**: infatti iterando abbastanza, il problema si ridurrà a trovare i singoli autovalori di matrici sulla diagonale sempre più piccole, e quindi dal polinomio caratteristico di più facile risoluzione.

## 1.2 Sistemi lineari

Veniamo quindi alla trattazione dei sistemi lineari, che avevamo definito come forme $Ax = b$ con $A \in \mathbb{C}^{n \times n}, b \in \mathbb{C}^n$.

Studieremo 2 tipi di metodi risolutivi:

- **Metodi diretti:** esatti ma dispendiosi, se eseguiti in aritmetica esatta (cioè senza arrotondamenti) poterebbero in un numero $n$ finito di passaggi alla soluzione esatta. Esempi di metodi diretti sono il **metodo di Cramer** (visto in 4.7.2) e l'**eliminazione di Gauss** (che vedremo fra poco);
- **Metodi iterativi:** meno accurati ma più efficienti computazionalmente, portano ad una successione $\{x_k\}_{k \in \mathbb{N}}$ di approssimazioni tali che $\lim_{k \to +\infty} x_k = x$ soluzione esatta. Notiamo però che, in generale, è impossibile trovare il valore esatto di $x$ in un numero esatto di iterazioni. Per contropartita, risultano spesso molto più efficienti dei metodi diretti (esistono esempi di sistemi addirittura non risolvibili, nella pratica, con metodi diretti).

### 1.2.1 Sistemi triangolari

Diamo la definizione parallela a quella di matrice triangolare:

**Definizione 1.1: Sistema triangolare**

Si dice sistema triangolare un sistema della forma $Ux = c$ con $U$ matrice triangolare.

Per un **sistema triangolare superiore**, si avrà la forma:

$$\begin{cases} u_{11}x_1 + u_{12}x_2 + ... + u_{1,n-1}x_{n-1} + u_{1n}x_n = c_1 \\ u_{22}x_2 + ... + u_{2,n-1}x_{n-1} + u_{2n}x_n = c_2 \\ \vdots \\ u_{n-1,n-1}x_{n-1} + u_{n-1,n}x_n = c_{n-1} \\ u_{nn}x_n = c_n \end{cases}$$

Il **metodo risolutivo** sarà allora la *sostituzione all'indietro*, definita ricorsivamente come:

$$\begin{cases} x_n = \frac{c_n}{u_{nn}} \\ x_i = \frac{c_i - \sum_{j=i+1}^{n} u_{ij}x_j}{u_{ii}} \end{cases}$$

che equivale all'algoritmo, in MATLAB:

```
function x = bck_subst(U, b)
    n = height(U);
    x = zeros(n, 1);

    for i = n:-1:1
        p = b(i);
        for j = (i + 1):n
            p = p - U(i, j) * x(j);
        end
        x(i) = p / U(i, i);
    end
end
```

Riguardo alla complessità, si potra dire che al passo $i$ si eseguono $n-i+n-i+2 = 2(n-1)+2$ passaggi, cioè 2 per la divisione per $u_{jj}$ e la somma fra $c_i$ e il termine accumulato a destra, $n-i$ per i prodotti nella sommatoria e di nuovo $n-i$ per la sommatoria stessa. Sarà allora che:

$$\sum_{i=1}^{n} (2(n-i)+2) \sim O(n^2)$$

cioè si ha complessità quadratica.

Osserviamo poi che per **sistemi triangolari inferiori** la situazione è uguale, cioè si risolve la prima equazione, si sostituisce il risultato nella seconda, e via dicendo:

$$\begin{cases} x_1 = \frac{c_1}{u_{11}} \\ \\ x_i = \frac{c_i - \sum_{j=1}^{i-1} u_{ij} x_j}{u_{ii}} \end{cases}$$

che equivale all'algoritmo, in MATLAB:

```
function x = fwd_subst(L, b)
    n = height(L);
    x = zeros(n, 1);

    for i = 1:n
        p = b(i);
        for j = 1:(i -1)
            p = p - L(i, j) * x(j);
        end
        x(i) = p / L(i, i);
    end
end
```

Il metodo ottenuto, speculare al quello di sostituzione all'indietro, viene detto *sostituzione in avanti*, di costo identico ($O(n^2)$).

### 1.2.2 Metodo di eliminazione di Gauss

L'idea del metodo di eliminazione di Gauss è quella di partire da un sistema $Ax = b$, trasformarlo in un sistema equivalente $Ux = c$ (quindi triangolare superiore), ed applicare la sostituzione all'indietro.

Per arrivare alla forma $Ux = c$ si sostituiscono le equazioni del sistema con loro combinazioni lineari scelte in modo da annullare gli elementi inferiori alla diagonale.

L'idea è quella di eliminare, per ogni elemento $i$-esimo sulla diagonale a partire da quello in alto a destra, gli $n-i$ elementi che stanno al di sotto, cioè:

Una semplice implementazione in MATLAB del suddetto algoritmo può essere la seguente:

**Algoritmo 1** Eliminazione di Gauss

**Input:** un sistema lineare qualsiasi $Ax = b$
**Output:** un sistema lineare triangolare superiore $Ux = c$
**for** $i = 1$ to $n$ **do**
  **for** $j = i$ to $n$ **do**
    Calcola il **moltiplicatore** $l_{ji} = \frac{a_{ji}^{(i-1)}}{a_{ii}^{(i-1)}}$
    Aggiungi alla riga $j$ la riga $i$ moltiplicata per $l_{ij}$
  **end for**
**end for**

```
function [A, b] = gauss_decomp(A, b)
    n = height(A);

    for i = 1:n % i itera sulle diagonali
        den = A(i, i);

        for j = (i + 1):n % j itera sulle righe
            mul = A(j, i) / den; % moltiplicatore

            A(j, :) = A(j, :) - A(i, :) * mul;
            b(j) = b(j) + b(i) * mul;
        end
    end
end
```

da cui si potrà ottenere una riduzione di Gauss semplicemente come:

```
>> [U, c] = gauss_decomp(A, b)
```

Dal punto di vista della complessità, la riduzione in forma triangolare costa $O(\frac{2}{3}n^3)$, e chiaramente domina sul termine $O(n^2)$ della risoluzione di $Ux = c$ con la sostituzione all'indietro.

Facciamo una nota sulla fattibiltà della riduzione di Gauss, definendo:

**Definizione 1.2: Moltiplicatori di Gauss**

I termini $l_{ji} = \frac{a_{ji}^{(i-1)}}{a_{ii}^{(i-1)}}$ vengono detti moltiplicatori.

Chiaramente, per poter eseguire l'eliminazione di Gauss serve che $a_{jj^{j-1}} \neq 0 \; \forall j = 1, ..., n-1$. Inoltre, i casi $a_{jj}^{j-1} \approx 0$ possono causare problemi di instabilità numerica. Vedremo in seguito metodi per ovviare a questo problema.

### 1.2.3 Fattorizzazione LU

Il primo passo dell'algoritmo di Gauss si può vedere come equivalente a moltiplicare l'equazione $Ax = b$ a sinistra per una particolare matrice $H_1$ triangolare inferiore:

$$H_1 = \begin{pmatrix} 1 & ... & ... & 0 \\ -l_{21} & 1 & ... & 0 \\ ... & & & \\ -l_{n1} & ... & ... & 1 \end{pmatrix}$$

con la diagonale a $1$ e i moltiplicatori sulla prima colonna, così che $AH_1$ risulti esattamente quello che volevamo per Gauss, cioè la combinazione lineare di ogni riga $j$ con l'prima riga moltiplicata per il moltiplicatore $l_{j1}$ (con $j > 2$).

Possiamo generalizzare questo processo a una serie di matrici $H_i$, per ogni elemento sulla diagonale, con la diagonale a $1$ e i moltiplicatori corrispondenti a $i$ sulla $i$-esima colonna:

$$H_i = \begin{pmatrix} 1 & ... & ... & 0 \\ 0 & 1 & ... & 0 \\ ... & -l_{ji} & ... & ... \\ 0 & -l_{ni} & ... & 1 \end{pmatrix}$$

Così, ancora una volta, $AH_i$ risulterà quello che volevamo per Gauss, cioè la combinazione lineare di ogni riga $j$ con l'$i$-esima riga moltiplicata per il moltiplicatore $l_{ji}$.

Varrà allora he il metodo di Gauss sarà equivalente a considerare:

$$H_{n-1}H_{n-2}...H_1Ax = H_{n-1}H_{n-2}...H_1b$$

e potremo quindi dire:

$$H_{n-1}H_{n-2}...H_1A = U, \quad L = H_1^{-1}...H_{n-1}^1$$

da cui:

$$A = LU$$

Semplifichiamo i calcoli notando alcune proprietà delle matrici $H_j$:

1. Hanno l'inversa facile, in quanto basta invertire i segni:

$$H_i^{-1} = \begin{pmatrix} 1 & ... & ... & 0 \\ 0 & 1 & ... & 0 \\ ... & l_{ji} & ... & ... \\ 0 & l_{ni} & ... & 1 \end{pmatrix}$$

2. Sono facili da moltiplicare, in quanto si può dire:

$$H_{i_1} \cdot H_{i_2} = \begin{pmatrix} 1 & ... & ... & 0 \\ -l_{2i_1} & 1 & ... & 0 \\ ... & -l_{ji_2} & ... & ... \\ -l_{ni_i} & -l_{ni_2} & ... & 1 \end{pmatrix}$$

   e:

$$H_{i_1}^{-1} \cdot H_{i_2}^{-1} = \begin{pmatrix} 1 & ... & ... & 0 \\ l_{2i_1} & 1 & ... & 0 \\ ... & l_{ji_2} & ... & ... \\ l_{ni_i} & l_{ni_2} & ... & 1 \end{pmatrix}$$

   cioè semplicemente si somma sotto la diagonale.

Questo significa che una volta svolta la prima parte dell'eliminazione di Gauss si è gia calcolata la fattorizzazione LU come la matrice dei moltiplicatori:

$$H_{i_1}^{-1} \cdot H_{i_2}^{-1} = \begin{pmatrix} 1 & ... & ... & 0 \\ l_{21} & 1 & ... & 0 \\ ... & l_{3,2} & ... & \\ l_{n1} & l_{n2} & ... & 1 \end{pmatrix}$$

Modifichiamo il codice MATLAB della scorsa sessione per calcolare, oltre all'eliminazione di Gauss (cioè la matrice $U$), la matrice dei moltiplicatori $L$:

```
function [A, b, L] = gauss_decomp(A, b)
    n = height(A);

    L = eye(n); % prepara L

    for i = 1:n % i itera sulle diagonali
        den = A(i, i);

        for j = (i + 1):n % j itera sulle righe
            mul = A(j, i) / den; % moltiplicatore
            L(j, i) = mul;

            A(j, :) = A(j, :) - A(i, :) * mul;
            b(j) = b(j) + b(i) * mul;
        end
    end
end
```

A questo punto per calcolare la fattorizzazione LU di una matrice basterà eseguire:

```
>> [U, ~, L] = gauss_decomp(A, b)
>> L * U % idealmente dara' A
```

Si osserva quindi che se già si conoscono $L$ ed $U$ (magari di una matrice che dovremo usare spesso) risolvere $Ax = b$ costa $O(n^2)$, in quanto basta dire:

$$Ax = b \Rightarrow LUx = b \implies x = U^{-1}L^{-1}b$$

dove basta risolvere a cascata:

$$\begin{cases} Ly = b \\ Ux = y \end{cases}$$

Questi sono due sistemi triangolari, uno **inferiore** (risolvibile per *sostituzione in avanti*), l'altro **superiore** (risolvibile per *sostituzione all'indietro*), da cui l'andamento complessivo $O(n^2)$.

In MATLAB, la soluzione si può quindi avere usando le funzioni definite finora:

```
>> [U, ~, L] = gauss_decomp(A, b)
>> y = fwd_subst(L, b)
>> x = bck_subst(U, y) % x e' la soluzione del sistema
```

Notiamo che questo vale se $L$ ed $U$ sono note (o come nell'esempio vengono calcolate), quindi tolto il prezzo dato dal doverle calcolare (come avevamo notato, conviene per matrici che magari dobbiamo usare spesso).

### 1.2.4 Metodo di Gauss per variabili matriciali

Se si vuole risolvere $AX = B$ con $X$ e $B$ matrici di vettori colonna di $s$ colonne:

$$X = \begin{pmatrix} x_1 & \dots & x_s \end{pmatrix}, \quad B = \begin{pmatrix} b_1 & \dots & b_s \end{pmatrix}$$

cioè se si vogliono risolvere $s$ sistemi lineari con la stessa matrice $A$:

$$Ax_1 = b_1, \quad \dots, \quad Ax_s = b_s$$

si può modificare l'algoritmo di Gauss, effettuando le mosse di Gauss sulla matrice aumentata $\left(A \,|\, B\right)$. Alla fine troveremo una matrice $\left(U \,|\, B^{(n-1)}\right)$, dove gli apici $(n-1)$ rappresentano che è la $B$ che si ottiene all'$n-1$-esimo passaggio, che risolverà i sistemi triangolari superiori:

$$Ux_1 = b_1^{(n-1)}, \quad \dots, \quad Ux_s = b_s^{(n-1)}$$

### 1.2.5 Metodo di Gauss per il calcolo dell'inversa

Un caso particolare è il **calcolo dell'inversa** con l'algoritmo di **Gauss-Jordan**. Infatti, scegliendo:

$$AX = I$$

con $n = s$, le colonne in $X$ diventeranno l'inversa di $A$ (basti vedere che $AA^{-1} = I$ per definizione).

Facciamo un ultimo esempio, assistito da MATLAB, per comprendere a pieno questo risultato. Vogliamo calcolare $A^{-1}$ come la soluzione del sistema $AX = I$, e capiamo quindi che quello che cerchiamo sono i vettori colonna $i_1, ..., i_n$ tali per cui:

$$Ai_1 = I_1, \quad ..., \quad Ai_n = I_n$$

con $I_i$ il vettore di zeri con un 1 all'$i$-esima riga. Sfruttiamo allora l'eliminazione di Gauss per ricavare due matrici, $U$ e $B^{(n-1)}$, tali che $Ux = B^{(n-1)}$, con il vantaggio che $U$ è triangolare superiore e quindi risolvibile per sostituzione all'indietro. Possiamo fare questo in MATLAB come:

```
AI = [A, eye(n)]
UB = gauss_decomp(AI, zeros(n, 1)) % argomento fittizio per b
U = UB(1:n, 1:n)
B = UB(1:n, (n + 1):(2 * n)
```

A questo punto potremo trovare le colonne $i_i$ come:

```
>> i1 = bck_subst(U, B(1:n, 1))
>> i2 = bck_subst(U, B(1:n, 2))
...
```

e infine concatenare le colonne come:

```
>> inv_A = [i1, ..., in]
```

In un unico script MATLAB, questo procedimento si realizza come:

```
function A_inv = gauss_inv(A)
    n = height(A);

    AI = [A, eye(n)];

    UB = gauss_decomp(AI, zeros(n, 1));
    U = UB(:, 1:n);
    B = UB(:, (n + 1):(2 * n));

    A_inv = zeros(n, n);

    for i = 1:n
        A_inv(:, i) = bck_subst(U, B(:, i));
    end
end
```