

# 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 matrice 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 affermare che se una matrice è riducibile, 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  $\{k+1, \dots, n\}$  a  $\{1, \dots, k\}$ .

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

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

con  $\mathcal{P} \cup \mathcal{Q} = \{1, \dots, n\}$  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  $n - k = |\mathcal{P}|$ .

□

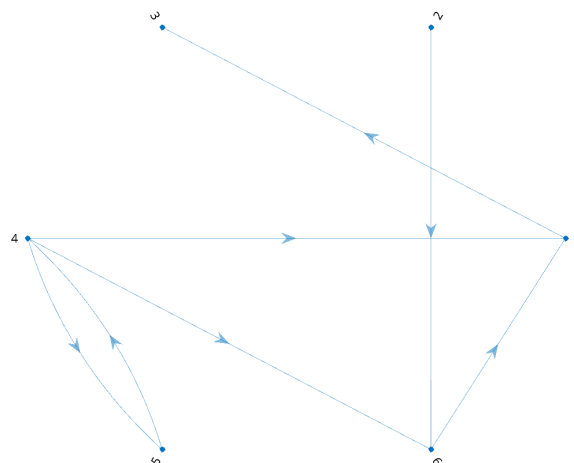
Prendiamo ad esempio la matrice:

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

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

```
1 >> A = [ 1 0 1 0 0 0; ... 1 0 0 0 0 1 ]
2 >> A(eye(size(A)) == 1) = 0 % rimuovi gli elementi sulla diagonale
3 >> G = digraph(A)
4 >> p = plot(G)
5 >> 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:

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

Applicando  $\Pi A \Pi^T$  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:

```
1 p = block_decomp(A) % ottieni una permutazione
2   Node      Reachable      Not reachable
3   ----      -
4
5   1      {[      1 3]}      {[ 2 4 5 6]}
6   2      {[ 1 2 3 6]}      {[      4 5]}
7   3      {[      3]}      {[1 2 4 5 6]}
8   4      {[1 3 4 5 6]}      {[      2]}
9   5      {[1 3 4 5 6]}      {[      2]}
10  6      {[ 1 3 6]}      {[ 2 4 5]}
11
12   Choose an index: 6 % chiesto dallo script
13 >> 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) potrebbero 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  $\lim_{k \rightarrow +\infty} x_k = x$  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:

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

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:

```

1 function x = bck_subst(U, b)
2     n = height(U);
3     x = zeros(n, 1);
4
5     for i = n:-1:1
6         p = b(i);
7         for j = (i + 1):n
8             p = p - U(i, j) * x(j);
9         end
10        x(i) = p / U(i, i);
11    end
12 end

```

Riguardo alla complessità, si potrà 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:

```

1 function x = fwd_subst(L, b)
2     n = height(L);
3     x = zeros(n, 1);
4
5     for i = 1:n
6         p = b(i);
7         for j = 1:(i - 1)
8             p = p - L(i, j) * x(j);
9         end
10        x(i) = p / L(i, i);
11    end
12 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**

---

```

1 function [A, b] = gauss_decomp(A, b)
2     n = height(A);
3
4     for i = 1:n % i itera sulle diagonali
5         den = A(i, i);
6
7         for j = (i + 1):n % j itera sulle righe
8             mul = A(j, i) / den; % moltiplicatore
9
10            A(j, :) = A(j, :) - A(i, :) * mul;
11            b(j) = b(j) + b(i) * mul;
12        end
13    end
14 end

```

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

```

1 >> [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 fattibilità 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  $a_{jj^{j-1}} \neq 0 \forall j = 1, \dots, n - 1$ . 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:

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

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:

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

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 che il metodo di Gauss sarà equivalente a considerare:

$$H_{n-1}H_{n-2} \dots H_1 Ax = H_{n-1}H_{n-2} \dots H_1 b$$

e potremo quindi dire:

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

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:

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

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

$$H_{i_1} \cdot H_{i_2} = \begin{pmatrix} 1 & \dots & \dots & 0 \\ -l_{2i_1} & 1 & \dots & 0 \\ \dots & -l_{ji_2} & \dots & \dots \\ -l_{ni_1} & -l_{ni_2} & \dots & 1 \end{pmatrix}$$

e:

$$H_{i_1}^{-1} \cdot H_{i_2}^{-1} = \begin{pmatrix} 1 & \dots & \dots & 0 \\ l_{2i_1} & 1 & \dots & 0 \\ \dots & l_{ji_2} & \dots & \dots \\ l_{ni_1} & l_{ni_2} & \dots & 1 \end{pmatrix}$$

cioè semplicemente si somma sotto la diagonale.

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

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

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

```

1 function [A, b, L] = gauss_decomp(A, b)
2     n = height(A);
3
4     L = eye(n); % prepara L
5
6     for i = 1:n % i itera sulle diagonali
7         den = A(i, i);
8
9         for j = (i + 1):n % j itera sulle righe
10             mul = A(j, i) / den; % moltiplicatore
11             L(j, i) = mul;
12
13             A(j, :) = A(j, :) - A(i, :) * mul;
14             b(j) = b(j) + b(i) * mul;
15         end
16     end
17 end

```

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

```

1 >> [U, ~, L] = gauss_decomp(A, b)
2 >> 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:

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

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:

```

1 >> [U, ~, L] = gauss_decomp(A, b)
2 >> y = fwd_subst(L, b)
3 >> 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  $(A|B)$ . Alla fine troveremo una matrice  $(U|B^{(n-1)})$ , dove gli apici  $(n-1)$



rappresentano che è la  $B$  che si ottiene all' $n - 1$ -esimo passaggio, che risolverà i sistemi triangolari superiori:

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

Facciamo un esempio, assistito da MATLAB, per comprendere a pieno questo risultato. Vogliamo calcolare  $X$  come la soluzione del sistema  $AX = B$ , e capiamo quindi che quello che cerchiamo sono i vettori colonna  $x_1, \dots, x_n$  tali per cui:

$$Ax_1 = B_1, \quad \dots, \quad Ax_n = b_n$$

con  $b_i$  l' $i$ -esimo vettore colonna di  $B$ . 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:

```
1 AI = [A, B]
2 UB1 = gauss_decomp(AB, zeros(n, 1)) % argomento fittizio per b
3 U = UB1(1:n, 1:n)
4 B1 = UB1(1:n, (n + 1):(n + s))
```

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

```
1 >> x1 = bck_subst(U, B1(1:n, 1))
2 >> x2 = bck_subst(U, B1(1:n, 2))
3 ...
```

e infine concatenare le colonne come:

```
1 >> X = [x1, ..., xn]
```

che realizzato in un unico script risulta:

```
1 function X = gauss_solve(A, B)
2     n = height(A);
3     s = width(B);
4
5     AB = [A, B];
6
7     UB1 = gauss_decomp(AB, zeros(n, 1));
8     U = UB1(:, 1:n);
9     B1 = UB1(:, (n + 1):(n + s));
10
11     X = zeros(n, s);
12
13     for i = 1:s
14         X(:, i) = bck_subst(U, B1(:, i));
15     end
16 end
```

esempietto numerico fatto in classe?

### 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 di questo procedimento. Vogliamo calcolare  $A^{-1}$  come la soluzione del sistema  $AX = I$ , e capiamo quindi che quello che cerchiamo sono i vettori colonna  $i_1, \dots, i_n$  tali per cui:

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

con  $I_i$  il vettore di zeri con un 1 all' $i$ -esima riga. Anche allora vogliamo sfruttare 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.

Questo è esattamente quello che ci permette di fare la funzione `gauss_solve()` definita prima, fissando il secondo argomento all'identità  $I$  (in matlab `eye(n)`):

```
1 function A_inv = gauss_inv(A)
2     n = height(A);
3     A_inv = gauss_solve(A, eye(n));
4 end
```

altro esempietto numerico ???