

PROGRAMMAZIONE INFORMATICA

8. MATLAB, PARTE 3: FUNZIONI PER L'ALGEBRA LINEARE E L'ANALISI

RICCARDO ZAMOLO
rzamolo@units.it

UNIVERSITÀ DEGLI STUDI TRIESTE
INGEGNERIA CIVILE E AMBIENTALE



A.A. 2020-21

- Il sistema di m equazioni lineari in n incognite x_1, \dots, x_n :

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n = b_2 \\ \vdots \\ a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n = b_m \end{cases}$$

può essere scritto in forma matriciale come:

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{Bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{Bmatrix} = \begin{Bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{Bmatrix}$$

$$\mathbf{Ax} = \mathbf{b}$$

- $m = n$ e $\det(\mathbf{A}) \neq 0$, ossia $\text{rank}(\mathbf{A}) = m = n$, \Rightarrow sistema determinato;
 - $m > n \Rightarrow$ sistema sovradeterminato;
 - $m < n \Rightarrow$ sistema sottodeterminato.
- In MATLAB il precedente sistema si risolve con il comando *backslash*:

$$\mathbf{x} = \mathbf{A} \backslash \mathbf{b}$$

che sceglie automaticamente che tipo di soluzione cercare in funzione delle dimensioni e delle proprietà della matrice \mathbf{A} .

- Calcolare la distanza tra un punto \mathbf{P}_0 nello spazio ed una retta r nella forma esplicita

$$\mathbf{r}(\lambda) = \mathbf{r}_0 + \lambda \mathbf{v}$$

- Metodo A (minimizzazione esplicita): si determina esplicitamente il valore di λ che minimizza la distanza punto-retta, equivalente ad imporre $\mathbf{P}_0 - \mathbf{r}(\lambda)$ ortogonale a \mathbf{v} :

$$\min_{\lambda \in \mathbb{R}} \|\mathbf{P}_0 - \mathbf{r}(\lambda)\|^2 \Rightarrow \bar{\lambda} = \frac{(\mathbf{P}_0 - \mathbf{r}_0)^T \mathbf{v}}{\mathbf{v}^T \mathbf{v}}$$

$$dist = \|\mathbf{P}_0 - \mathbf{r}(\bar{\lambda})\|$$

```
% Valori particolari
P0 = [ 1 ; 1 ; 0 ] ; % punto

r0 = [ 0 ; 0 ; 0 ] ; % retta, punto di partenza
v = [ 1 ; -1 ; 0 ] ; % retta, vettore direzione
r = @(l) r0 + l*v ; % funzione anonima della retta

% Metodo A
lambda = ( (P0-r0)' * v ) / ( v'*v ) ;
dist_A = norm( P0 - r(lambda) )
```

- Che fornisce il risultato:

```
dist_A =
    1.4142
```

- Metodo B (sottrazione componente parallela): si scompone la differenza $\mathbf{d} = \mathbf{P}_0 - \mathbf{r}_0$ in una componente parallela e in una ortogonale a \mathbf{v} :

$$\mathbf{d} = \mathbf{d}_v + \mathbf{d}_\perp$$

Fissato $\mathbf{e}_v = \mathbf{v}/\|\mathbf{v}\|$ il vettore di lunghezza unitaria in direzione \mathbf{v} , la componente parallela si ottiene direttamente dal prodotto scalare lungo \mathbf{e}_v in quanto $\mathbf{d}_\perp \cdot \mathbf{e}_v = 0$:

$$\mathbf{d} \cdot \mathbf{e}_v = \mathbf{d}_v \cdot \mathbf{e}_v \quad \Rightarrow \quad \mathbf{d}_v = (\mathbf{d} \cdot \mathbf{e}_v)\mathbf{e}_v$$

Si sottrae quindi a \mathbf{d} la sua componente parallela a \mathbf{v} , lasciando quindi la componente ortogonale \mathbf{d}_\perp la cui lunghezza è proprio la distanza cercata:

$$dist = \|\mathbf{d} - \mathbf{d}_v\| = \|\mathbf{d}_\perp\|$$

```
% Metodo B
e_v = v / norm(v) ;           % vettore unitario lungo v
d   = P0 - r0 ;               % differenza punto-partenza retta
d_v = dot(d, e_v) * e_v ;     % componente parallela
d_ortho = d - d_v ;           % componente ortogonale
dist_B = norm(d_ortho)        % distanza
```

- Che fornisce il risultato:

```
dist_B =
    1.4142
```

- Metodo C (matriciale: ingiustificato e sconsigliato): si scrivono le condizioni di ortogonalità in forma esplicita:

$$\mathbf{d} = \mathbf{d}_v + \mathbf{d}_\perp = \lambda \mathbf{v} + I_3 \mathbf{d}_\perp$$

$$\mathbf{d}_\perp \cdot \mathbf{v} = 0$$

Si assumono come incognite il vettore ortogonale \mathbf{d}_\perp ed il coefficiente λ , per un totale di 3+1 scalari, da esprimere mediante il seguente sistema lineare:

$$\begin{bmatrix} I_3 & \mathbf{v} \\ \mathbf{v}^T & 0 \end{bmatrix} \begin{Bmatrix} \mathbf{d}_\perp \\ \lambda \end{Bmatrix} = \begin{Bmatrix} \mathbf{d} \\ 0 \end{Bmatrix}$$

$$\mathbf{M}\mathbf{x} = \mathbf{b}$$

```
% Metodo C (esagerato)
I3 = eye(3,3) ;
M = [ I3 v ; ...
      v' 0 ] ; % costruzione matrice
d = P0 - r0 ; % differenza punto-partenza retta
b = [ d ; 0 ] ; % vettore termini noti
x = M\b ; % soluzione mediante backslash
d_ortho = x(1:3) ; % vettore ortogonale = prime 3 incognite
dist_C = norm( d_ortho )
```

- Che fornisce il risultato corretto `dist_C = 1.4142`.

- Metodo D (Pitagora): si fa il prodotto scalare

$$\mathbf{d} \cdot \mathbf{d} = \mathbf{d} \cdot \mathbf{d}_v + \mathbf{d} \cdot \mathbf{d}_\perp = (\mathbf{d} \cdot \mathbf{e}_v)^2 + dist^2$$

$$dist = \sqrt{\mathbf{d} \cdot \mathbf{d} - (\mathbf{d} \cdot \mathbf{e}_v)^2}$$

con $\mathbf{e}_v = \mathbf{v} / \|\mathbf{v}\|$ il vettore di lunghezza unitaria in direzione \mathbf{v} .

```
% Metodo D
d  = P0 - r0 ;           % differenza punto-partenza retta
dd = dot(d, d) ;        % ipotenusa^2
e_v = v / norm(v) ;     % vettore unitario lungo v
L_v = dot(d, e_v)^2 ;   % cateto^2 lungo v
dist_D = sqrt( dd - L_v )
```

- Che fornisce il risultato:

```
dist_D =
    1.4142
```

- Calcolare la distanza tra un punto \mathbf{P}_0 nello spazio ed un piano q nella forma esplicita

$$\mathbf{q}(\lambda_1, \lambda_2) = \mathbf{q}_0 + \lambda_1 \mathbf{v}_1 + \lambda_2 \mathbf{v}_2 = \mathbf{q}_0 + [\mathbf{v}_1 \ \mathbf{v}_2] \begin{Bmatrix} \lambda_1 \\ \lambda_2 \end{Bmatrix} = \mathbf{q}_0 + \mathbf{V}\boldsymbol{\lambda}$$

- Metodo A (ortogonalità): si impone $\mathbf{P}_0 - \mathbf{q}(\lambda_1, \lambda_2)$ ortogonale al piano:

$$\mathbf{v}_i \cdot (\mathbf{P}_0 - \mathbf{q}(\lambda_1, \lambda_2)) = 0 \quad i = 1, 2$$

$$\mathbf{V}^T (\mathbf{P}_0 - \mathbf{q}_0 - \mathbf{V}\boldsymbol{\lambda}) = \begin{Bmatrix} 0 \\ 0 \end{Bmatrix} \Rightarrow \mathbf{V}^T \mathbf{V}\boldsymbol{\lambda} = \mathbf{V}^T (\mathbf{P}_0 - \mathbf{q}_0)$$

$$\text{dist} = \|\mathbf{P}_0 - \mathbf{q}(\lambda_1, \lambda_2)\|$$

```
% Valori particolari
P0 = [ 3 ; 3 ; 5 ] ; % punto

q0 = [ 0 ; 4 ; 2 ] ; % piano, punto di partenza
v1 = [ 1 ; 3 ; 0 ] ; % piano, primo vettore direzione
v2 = [ 0 ; 2 ; 5 ] ; % piano, secondo vettore direzione

V = [ v1 v2 ] ; % matrice V = [ v1 v2 ]
q = @(lambda) q0 + V*lambda ; % funzione piano

% Metodo A
A = V' * V ; % matrice quadrata dei coefficienti
b = V' * (P0-q0) ; % vettore dei termini noti
lambda = A\b ; % soluzione, lambda
dist_A = norm( P0 - q(lambda) )
```

- Metodo B (prodotto vettoriale): si calcola il prodotto vettoriale

$$\mathbf{c} = \mathbf{v}_1 \times \mathbf{v}_2 = \det \begin{bmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ v_{1,x} & v_{1,y} & v_{1,z} \\ v_{2,x} & v_{2,y} & v_{2,z} \end{bmatrix}$$

e lo si normalizza mediante $\mathbf{e}_c = \mathbf{c}/\|\mathbf{c}\|$. Il prodotto vettoriale \mathbf{c} è quindi ortogonale al piano, essendo ortogonale sia a \mathbf{v}_1 che a \mathbf{v}_2 .

La distanza cercata è quindi data dalla componente di $\mathbf{P}_0 - \mathbf{q}_0$ lungo \mathbf{e}_c , senza segno:

$$dist = |(\mathbf{P}_0 - \mathbf{q}_0) \cdot \mathbf{e}_c|$$

```
% Metodo B
c = cross( v1 , v2 ) ; % prodotto vettoriale, vettore ortogonale al piano
e_c = c / norm( c ) ; % normalizzazione (lunghezza unitaria)
dist_B = abs( dot(P0-q0, e_c) )
```

- Dall'esecuzione dei precedenti script si ottiene:

```
dist_A =
    3.5138

dist_B =
    3.5138
```

- Scrivere una funzione che, dato il piano q (nello spazio) nella forma

$$\mathbf{q}(\lambda_1, \lambda_2) = \mathbf{q}_0 + \lambda_1 \mathbf{v}_1 + \lambda_2 \mathbf{v}_2,$$

ne determini i coefficienti della forma canonica

$$c_1x + c_2y + c_3z + d = \mathbf{c} \cdot \mathbf{q} + d = 0$$

- Il piano nella forma canonica può essere scritto come prodotto scalare

$$\mathbf{c} \cdot (\mathbf{q} - \mathbf{q}_0) = 0 \quad \Rightarrow \quad d = -\mathbf{c} \cdot \mathbf{q}_0$$

- Il vettore \mathbf{c} è ortogonale a \mathbf{v}_1 e \mathbf{v}_2 :

$$\mathbf{c} = \mathbf{v}_1 \times \mathbf{v}_2$$

piano_canonico.m

```
% Input: q0, vettore colonna 3x1 di un punto nel piano
%       v1 e v2, vettori colonna 3x1 delle direzioni lungo il piano
% Output: c, vettore colonna 3x1 dei coefficienti della forma canonica
%        d, costante della forma canonica
function [ c , d ] = piano_canonico(q0, v1, v2)
    c = cross(v1, v2) ;
    d = -dot(c, q0) ;
end
```

- Scrivere una funzione che, dato il piano q (nello spazio) in forma canonica

$$c_1x + c_2y + c_3z + d = \mathbf{c} \cdot \mathbf{q} + d = 0$$

ne determini una particolare forma esplicita del tipo

$$\mathbf{q}(\lambda_1, \lambda_2) = \mathbf{q}_0 + \lambda_1 \mathbf{v}_1 + \lambda_2 \mathbf{v}_2,$$

- Si determineranno i due vettori \mathbf{v}_1 e \mathbf{v}_2 in maniera che siano entrambi ortogonali a \mathbf{c} :

$$\mathbf{c}^T \mathbf{v}_1 = 0, \quad \mathbf{c}^T \mathbf{v}_2 = 0$$

- Pensando a \mathbf{c}^T come ad una matrice dei coefficienti di dimensione 1×3 , $\mathbf{N} = \ker(\mathbf{c}^T)$ sarà una matrice 3×2 le cui 2 colonne \mathbf{n}_j saranno tali che $\mathbf{c}^T \mathbf{n}_j = \mathbf{0}$, ossia proprio i vettori \mathbf{v}_1 e \mathbf{v}_2 cercati.
- Imponiamo poi che \mathbf{v}_1 e \mathbf{v}_2 siano ortogonali tra di loro, cioè $\mathbf{v}_1 \cdot \mathbf{v}_2 = 0$, e che siano *normalizzati*, cioè di lunghezza unitaria $\|\mathbf{v}_1\| = \|\mathbf{v}_2\| = 1$.
- Dalla forma del termine costante $d = -\mathbf{c}^T \mathbf{q}_0$ possiamo ricavare un particolare vettore \mathbf{q}_0 pensando ancora \mathbf{c}^T come una matrice 1×3 dei coefficienti.

```

% Input: c, vettore colonna 3x1 dei coefficienti della forma canonica
%         d, costante della forma canonica
% Output: q0, vettore colonna 3x1 di un punto nel piano
%         v1 e v2, vettori colonna 3x1 delle direzioni lungo il piano
function [ q0 , v1 , v2 ] = piano_esplicito(c, d)
    N = null( c' ) ; % ker( c^T )
    v1 = N(:,1) ;    % prima colonna di N
    v2 = N(:,2) ;    % seconda colonna di N
    q0 = -c' \ d ;   % un generico punto del piano
end

```

- Si può verificare la correttezza delle due precedenti funzioni per la conversione della forma del piano:

```

>> c = [ 1 ; 1 ; 1 ] ;
>> d = 1 ;
>> [q0,v1,v2] = piano_esplicito(c,d);
>> [c,d] = piano_canonico(q0,v1,v2)
c =
    0.5774
    0.5774
    0.5774

d =
    0.5774

```

```

>> q0 = [ 1 ; 2 ; 3 ] ;
>> v1 = [ 1 ; 0 ; 0 ] ;
>> v2 = [ 0 ; 1 ; 0 ] ;
>> [c,d] = piano_canonico(q0,v1,v2);
>> [q0,v1,v2] = piano_esplicito(c,d)
q0 =
     0
     0
     3
v1 =
     0
     1
     0
v2 =
    -1
     0
     0

```

- Dato un piano q (nello spazio) in forma canonica

$$\mathbf{c} \cdot \mathbf{q} + d = 0$$

la distanza di un generico punto \mathbf{P}_0 da esso è data da

$$dist = \left| \frac{\mathbf{c}}{\|\mathbf{c}\|} \cdot (\mathbf{P}_0 - \mathbf{q}_0) \right| \Rightarrow dist = \frac{|\mathbf{c} \cdot \mathbf{P}_0 + d|}{\|\mathbf{c}\|}$$

dove \mathbf{q}_0 è un qualsiasi punto del piano.

- Scrivere una funzione per il calcolo della distanza di un punto da un piano espresso in forma canonica, e verificare che la distanza calcolata con la forma esplicita (Parte3_Es2.m) del piano coincida per un caso particolare.

distanza_punto_piano_canonico.m

```
% Input: P0, vettore colonna 3x1 del punto nello spazio
%        c, vettore colonna 3x1 dei coefficienti della forma canonica
%        d, costante della forma canonica
% Output: dist, distanza di P0 dal piano
function dist = distanza_punto_piano_canonico(P0, c, d)
    norm_c = norm( c );
    piano  = dot( c , P0 ) + d ;
    dist   = abs( piano ) / norm_c ;
end
```

distanza_punto_piano_esplicito.m

```

% Input: P0, vettore colonna 3x1 del punto nello spazio
%        q0, vettore colonna 3x1 di un punto nel piano
%        v1 e v2, vettori colonna 3x1 delle direzioni lungo il piano
% Output: dist, distanza di P0 dal piano, (Metodo A)
function dist = distanza_punto_piano_esplicito(P0, q0, v1, v2)
    V = [ v1 v2 ] ;
    q = @(lambda) q0 + V*lambda ; % funzione piano
    A = V' * V ;                % matrice quadrata dei coefficienti
    b = V' * (P0-q0) ;          % vettore dei termini noti
    lambda = A\b ;              % soluzione, lambda
    dist = norm( P0 - q(lambda) ) ;
end

```

- Verifica in un caso particolare:

```

P0 = [ 1 ; 2 ; 3 ] ; % punto nello spazio
c = [ 1 ; 1 ; 1 ] ; % vettore coefficienti della forma canonica
d = 1 ;              % vettore termine costante della forma canonica

% Distanza punto-piano in forma canonica
dist_canonico = distanza_punto_piano_canonico(P0, c, d)

% Distanza punto-piano in forma esplicita
[ q0 , v1 , v2 ] = piano_esplicito(c, d) ;
dist_esplicito = distanza_punto_piano_esplicito(P0, q0, v1, v2)

```

```

dist_canonico =
    4.0415

dist_esplicito =
    4.0415

```

- Scrivere una funzione che, data una retta r nello spazio espressa nella forma esplicita

$$\mathbf{r}(\lambda) = \mathbf{r}_0 + \lambda \mathbf{v},$$

ne determini una forma implicita come intersezione di due piani espressi in forma canonica:

$$\begin{cases} \mathbf{c}_1 \cdot \mathbf{r} + d_1 = 0 \\ \mathbf{c}_2 \cdot \mathbf{r} + d_2 = 0 \end{cases}$$

- Si avrà sempre

$$\mathbf{c}_1 \cdot \mathbf{v} = 0, \quad \mathbf{c}_2 \cdot \mathbf{v} = 0$$

e quindi \mathbf{c}_1 e \mathbf{c}_2 si potranno nuovamente ottenere dal $\ker(\mathbf{v}^T)$. Le costanti d_1 e d_2 si possono poi ottenere sostituendo la forma esplicita della retta nel sistema in forma canonica, ottenendo:

$$d_1 = -\mathbf{c}_1 \cdot \mathbf{r}_0, \quad d_2 = -\mathbf{c}_2 \cdot \mathbf{r}_0$$

retta_canonica.m

```
% Input: r0, vettore colonna 3x1 di un punto della retta
%       v, vettore colonna 3x1 della direzione della retta
% Output: c1,c2,d1,d2 coefficienti e termini costanti della forma canonica
function [ c1 , c2 , d1 , d2 ] = retta_canonica(r0, v)
    N = null( v' );
    c1 = N(:,1) ;
    c2 = N(:,2) ;
    d1 = -dot(c1, r0) ;
    d2 = -dot(c2, r0) ;
end
```

- Scrivere una funzione che, data una retta r nello spazio espressa in forma implicita come intersezione di due piani espressi in forma canonica:

$$\begin{cases} \mathbf{c}_1 \cdot \mathbf{r} + d_1 = 0 \\ \mathbf{c}_2 \cdot \mathbf{r} + d_2 = 0, \end{cases}$$

ne determini una forma esplicita del tipo

$$\mathbf{r}(\lambda) = \mathbf{r}_0 + \lambda \mathbf{v},$$

- Metodo A (prodotto vettoriale): si calcola direttamente il vettore \mathbf{v} mediante prodotto vettoriale $\mathbf{v} = \mathbf{c}_1 \times \mathbf{c}_2$, mentre un punto \mathbf{r}_0 della retta è dato da una soluzione particolare del sistema canonico

$$\begin{bmatrix} \mathbf{c}_1^T \\ \mathbf{c}_2^T \end{bmatrix} \mathbf{r}_0 = - \begin{Bmatrix} d_1 \\ d_2 \end{Bmatrix}$$

poichè $\mathbf{c}_1 \cdot \mathbf{v} = 0$ e $\mathbf{c}_2 \cdot \mathbf{v} = 0$:

`retta_esplicita_A.m`

```
% Input: c1,c2,d1,d2 coefficienti e termini costanti della forma canonica
% Output: r0, vettore colonna 3x1 di un punto della retta
%         v, vettore colonna 3x1 della direzione della retta
function [ r0 , v ] = retta_esplicita_A(c1, c2, d1, d2)
    v = cross(c1, c2) ;
    r0 = -[ c1' ; c2' ] \ [ d1 ; d2 ] ;
end
```

- Metodo B (soluzione diretta): si calcola nuovamente \mathbf{r}_0 come soluzione particolare del sistema canonico, mentre il vettore \mathbf{v} sarà dato dal ker della matrice 2×3 dei coefficienti:

$$\begin{cases} \mathbf{c}_1 \cdot \mathbf{r} + d_1 = 0 \\ \mathbf{c}_2 \cdot \mathbf{r} + d_2 = 0 \end{cases} \Leftrightarrow \begin{bmatrix} \mathbf{c}_1^T \\ \mathbf{c}_2^T \end{bmatrix} \mathbf{r} = - \begin{bmatrix} d_1 \\ d_2 \end{bmatrix}$$

retta_esplicita_B.m

```
% Input: c1,c2,d1,d2 coefficienti e termini costanti della forma canonica
% Output: r0, vettore colonna 3x1 di un punto della retta
%          v, vettore colonna 3x1 della direzione della retta
function [ r0 , v ] = retta_esplicita_B(c1, c2, d1, d2)
    A = [ c1' ; c2' ] ;
    b = [ d1 ; d2 ] ;
    r0 = -A \ b ;
    v = null( A ) ;
end
```

- Utilizzo:

```
r0 = [ 1 ; 2 ; 3 ] ;
v = [ 0 ; 0 ; 5 ] ;
[c1,c2,d1,d2]=retta_canonica(r0,v);
[r0,v]=retta_esplicita_A(c1,c2,d1,d2)
```

```
r0 =
    1 ; 2 ; 0
v =
    0 ; 0 ; 1
```

```
[r0,v]=retta_esplicita_B(c1,c2,d1,d2)
```

```
r0 =
    1 ; 2 ; 0
v =
    0 ; 0 ; 1
```

- Scrivere una funzione che, dato un piano q ed una retta r ed nello spazio espressi in forma esplicita:

$$\mathbf{q}(\lambda_1, \lambda_2) = \mathbf{q}_0 + \lambda_1 \mathbf{v}_1 + \lambda_2 \mathbf{v}_2$$

$$\mathbf{r}(\lambda) = \mathbf{r}_0 + \lambda \mathbf{v},$$

determini il punto d'intersezione, nell'ipotesi che esista ($\mathbf{v} \neq \alpha \mathbf{v}_1 + \beta \mathbf{v}_2$).

- Metodo A (diretto): si eguaglia direttamente la retta r con il piano q , assumendo come incognite i coefficienti $\lambda_1, \lambda_2, \lambda$:

$$\mathbf{q}_0 + \lambda_1 \mathbf{v}_1 + \lambda_2 \mathbf{v}_2 = \mathbf{r}_0 + \lambda \mathbf{v} \quad \Rightarrow \quad \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & -\mathbf{v} \end{bmatrix} \begin{Bmatrix} \lambda_1 \\ \lambda_2 \\ \lambda \end{Bmatrix} = \mathbf{r}_0 - \mathbf{q}_0$$

che una volta risolto permette di calcolare indirettamente il punto d'intersezione utilizzando l'equazione esplicita della retta, per esempio:

intersezione_piano_retta_espliciti_A.m

```
% Input: q0, v1, v2, vettori 3x1 del piano
%         r0, v, vettori 3x1 della retta
% Output: x, vettore 3x1 dell'intersezione
function x = intersezione_piano_retta_espliciti_A(q0, v1, v2, r0, v)
    A = [ v1 v2 -v ] ;
    z = A \ ( r0 - q0 ) ;
    x = r0 + z(3) * v ;
end
```

- Metodo B (forma canonica): si trasformano in forma canonica sia il piano che la retta, pervenendo al seguente sistema:

$$\begin{cases} \mathbf{c} \cdot \mathbf{x} + d = 0 \\ \mathbf{c}_1 \cdot \mathbf{x} + d_1 = 0 \\ \mathbf{c}_2 \cdot \mathbf{x} + d_2 = 0 \end{cases} \Rightarrow \begin{bmatrix} \mathbf{c}^T \\ \mathbf{c}_1^T \\ \mathbf{c}_2^T \end{bmatrix} \mathbf{x} = - \begin{bmatrix} d \\ d_1 \\ d_2 \end{bmatrix}$$

che una volta risolto fornisce direttamente il punto d'intersezione:

`intersezione_piano_retta_espliciti_B.m`

```
% Input: q0, v1, v2, vettori 3x1 del piano
%       r0, v, vettori 3x1 della retta
% Output: x, vettore 3x1 dell'intersezione
function x = intersezione_piano_retta_espliciti_B(q0, v1, v2, r0, v)
    [ c , d ] = piano_canonico(q0, v1, v2) ;
    [ c1 , c2 , d1 , d2 ] = retta_canonica(r0, v) ;
    A = [ c' ; c1' ; c2' ] ;
    b = [ d ; d1 ; d2 ] ;
    x = -A \ b ;
end
```

- *Regressione lineare.* Data una serie di n valori x_i della variabile indipendente in corrispondenza dei quali sono dati n valori \bar{y}_i della variabile dipendente, ad esempio delle misurazioni, ci si pone il problema di determinare la retta $y = c_1 + c_2x$ che “meglio” approssima i dati forniti.
- Soluzione ai minimi quadrati: c_1 e c_2 si determinano minimizzando la somma dei quadrati degli scarti:

$$\min_{c_1, c_2 \in \mathbb{R}} \sum_{i=1}^n \delta_i^2 = \min_{c_1, c_2 \in \mathbb{R}} \sum_{i=1}^n \left((c_1 + c_2x_i) - \bar{y}_i \right)^2$$

- Organizzando i dati nei vettori colonna $\mathbf{x} = (x_i)$ e $\bar{\mathbf{y}} = (\bar{y}_i)$, il vettore colonna degli scarti è

$$\boldsymbol{\delta} = (c_1 + c_2\mathbf{x}) - \bar{\mathbf{y}} = \mathbf{A}\mathbf{c} - \bar{\mathbf{y}}$$

dove

$$\mathbf{A} = \begin{bmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix} = [\mathbf{1} \ \mathbf{x}] \ , \quad \mathbf{c} = \begin{Bmatrix} c_1 \\ c_2 \end{Bmatrix}$$

- Il problema si può quindi risolvere calcolando la soluzione ai minimi quadrati del sistema lineare sovradeterminato $\mathbf{A}\mathbf{c} = \bar{\mathbf{y}}$, dove la matrice dei coefficienti \mathbf{A} ha dimensione $n \times 2$.

- Implementare la soluzione ai minimi quadrati per la regressione lineare, utilizzando la seguente funzione che fornisce i valori \bar{y}_i delle misure:

misura_con_rumore.m

```
% Input: x, vettore dei valori della variabile indipendente
%         c1, c2, coefficienti della retta
%         r, peso del rumore
% Output: y, vettore dei valori della variabile dipendente nella forma
%         y = c1 + c2*x + rumore
function y = misura_con_rumore(x, c1, c2, r)
    y = c1 + c2 * x ;
    rumore = rand(size(x)) - .5 ;
    y = y + r * rumore .* y ;
end
```

```
% Numero di valori
n = 50 ;

% Valori variabile indipendente
x = linspace(0, 1, n)';

% Valori variabile dipendente, misura
c1 = 3.0 ;
c2 = 9.0 ;
r = 0.1 ;
y = misura_con_rumore(x, c1, c2, r) ;

% Minimi quadrati
A = [ ones(n,1) x ] ;
c = A \ y ;
```

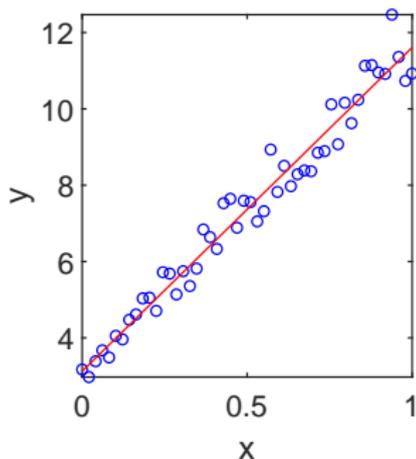
- Possiamo poi visualizzare i risultati mediante plot ed output testuale:

```

% Plot
plot( x , y , 'ob' ) ;           % dati
hold on ;
plot( x , c(1)+c(2)*x , '-r' ) ; % retta di regressione

% Output testuale dei coefficienti
testo = [ 'Regressione lineare di %d valori:\n' ...
          ' c      = (%.2f,%.2f)\n' ...
          ' c_regr = (%.2f,%.2f)\n' ] ;
fprintf(testo, n, ...
        c1 , c2 , ...
        c(1), c(2) ) ;

```



- L'output testuale che si ottiene nella command window è

```

Regressione lineare di 50 valori:
 c      = (3.00,9.00)
 c_regr = (2.96,9.06)

```

- *Eliminazione di Gauss.* Data una matrice $\mathbf{A} = (a_{ij})$ di dimensione $m \times n$, si eseguono successive operazioni lineari sulle righe, in particolare lo scambio di due righe R_i ed R_k e la somma ad una riga R_i del multiplo di un'altra riga R_k :

$$R_i \leftrightarrow R_k$$

$$R_i \leftarrow R_i - \lambda R_k$$

in maniera tale da ottenere una matrice *triangolare superiore*.

- Scrivere una funzione che, data una matrice \mathbf{A} , restituisca la relativa matrice triangolare superiore ottenuta effettuando l'eliminazione di Gauss, scrivendo come output testuale la serie di operazioni eseguite, ad esempio:

$$R2 \leftrightarrow R1$$

$$R2 \leftarrow R2 - 5 \times R1$$

seguite dal risultato dell'operazione nella matrice.

- Si deve prevedere il caso di un elemento *pivot* nullo, che verrà sostituito con un altro elemento pivot non nullo, se esiste.

- Conviene raggruppare le operazioni da effettuare sulle righe in apposite funzioni definite all'interno della funzione principale.
- Sottofunzione per lo scambio di due righe $i1$ e $i2$:

```
function M = scambia_righe( M , i1 , i2 )
M( [i1;i2] , : ) = M( [i2;i1] , : ) ; % scambio righe
fprintf('R%d ⇌ R%d\n', i1, i2) ;    % output testuale dell'operazione
disp( M ) ;                          % output della matrice
end
```

- Sottofunzione per l'operazione $R_i \leftarrow R_i - \lambda R_k$:

```
function M = somma_righe( M , i , lambda , k )
M( i , : ) = M( i , : ) + lambda * M( k , : ) ; % somma righe
fprintf('R%d ← R%d%+.2f×R%d\n', i, i, lambda, k) ; % output testuale
disp( M ) ; % output della matrice
end
```

- Sottofunzione per l'operazione di scambio del pivot:

```
function M = scambio_pivot( M , j )
possibili_pivot = M( (j+1):end , j ) ; % estrazione colonna j
[ ~ , k ] = max( abs( possibili_pivot ) ) ; % ricerca pivot "migliore"
M = scambia_righe( M , j , j+k ) ; % scambio righe pivot
end
```

- Funzione principale:

eliminazione_Gauss.m

```
function A = eliminazione_Gauss( A )
...
( sottofunzioni per le operazioni sulle righe )
...
% Funzione per il test x = 0
is_zero = @(x) abs(x) < 10*eps ;

m = size(A, 1) ;
% Ciclo sulle colonne da ridurre a 0
for j = 1 : m-1
    pivot = A( j , j ) ; % il pivot è l'elemento diagonale

    % Caso di pivot nullo => nuovo pivot (se possibile)
    if is_zero( pivot )
        A = scambio_pivot( A , j ) ;
        pivot = A( j , j ) ;
    end

    % Ciclo sugli elementi da eliminare nella colonna j
    for i = j+1 : m
        lambda = -A(i,j) / pivot ; % coefficiente scalare per l'eliminazione
        if ~is_zero( lambda )      % se lambda = 0 l'eliminazione non serve
            A = somma_righe( A , i , lambda , j ) ; % operazione somma righe
        end
    end
end
end

end
```

- *Sostituzione all'indietro.* Data la matrice triangolare superiore \mathbf{T} ed il vettore dei termini noti ridotti \mathbf{f} ottenuti operando l'eliminazione di Gauss sulla matrice \mathbf{A} e sul vettore dei termini noti \mathbf{b} , la soluzione \mathbf{x} del sistema

$$\mathbf{Ax} = \mathbf{b},$$

con \mathbf{A} matrice quadrata $m \times m$, si può ottenere operando successive sostituzioni all'indietro utilizzando \mathbf{T} ed \mathbf{f} , determinando quindi le incognite x_m, \dots, x_1 dall'ultima alla prima.

- Scrivere una funzione che, data una matrice triangolare superiore \mathbf{T} ed il vettore dei termini noti ridotto \mathbf{f} , restituisca la soluzione \mathbf{x} .

sostituzione.m

```
% Input: T, matrice m x m triangolare superiore quadrata
%         f, matrice m x k dei termini noti ridotti
% Output: f, matrice m x k, soluzione
function f = sostituzione( T , f )
    m = size(T, 1) ;
    % Ciclo sugli m valori incogniti da determinare, all'indietro
    for i = m : -1 : 1
        jx = (i+1) : m ; % indici dei termini incogniti già risolti
        v_T = T( i , jx ) ; % corrispondenti termini di T per l'incognita i
        v_f = f( jx , : ) ; % termini incogniti già risolti
        f( i , : ) = ( f( i , : ) - v_T * v_f ) / T( i , i ) ;
    end
end
```

- Utilizzo, **b** vettore:

```

% Matrice e termine noto particolari
A = [ 0 1 2 ; 4 5 6 ; 7 7 8 ] ;
b = [ 1 ; 1 ; 3 ] ;

% Eliminazione di Gauss su concatenazione di A e b
Tf = eliminazione_Gauss( [ A b ] ) ;
T = Tf(:,1:3) ; % estrazione matrice triang. sup. T
f = Tf(:, 4) ; % estrazione termini noti ridotti f

% Sostituzione all'indietro
x = sostituzione( T , f ) ;

% Verifica
A*x - b ;

```

```

R1  $\leftrightarrow$  R2
  4      5      6      1
  0      1      2      1
  7      7      8      3

R3  $\leftarrow$  R3-1.75×R1
  4.0000   5.0000   6.0000   1.0000
           0   1.0000   2.0000   1.0000
           0  -1.7500  -2.5000   1.2500

R3  $\leftarrow$  R3+1.75×R2
  4      5      6      1
  0      1      2      1
  0      0      1      3

```

- Utilizzo, **b** matrice identità per determinare l'inversa di **A**:

```
% Matrice identità
b = eye(3,3) ;

% Eliminazione di Gauss su concatenazione di A e b
Tf = eliminazione_Gauss( [ A b ] ) ;
T = Tf(:,1:3) ; % estrazione matrice triang. sup. T
f = Tf(:,4:6) ; % estrazione termini noti ridotti f

% Sostituzione all'indietro
inv_A = sostituzione( T , f ) ;

% Verifica
A * inv_A ;
```

R1 \leftrightarrow R2

4	5	6	0	1	0
0	1	2	1	0	0
7	7	8	0	0	1

R3 \leftarrow R3 - 1.75 \times R1

4.0000	5.0000	6.0000	0	1.0000	0
0	1.0000	2.0000	1.0000	0	0
0	-1.7500	-2.5000	0	-1.7500	1.0000

R3 \leftarrow R3 + 1.75 \times R2

4.0000	5.0000	6.0000	0	1.0000	0
0	1.0000	2.0000	1.0000	0	0
0	0	1.0000	1.7500	-1.7500	1.0000

- Oltre alle funzioni già viste:
 - `norm(v)` per la norma 2 di vettori (e matrici) $\|v\|$,
 - `dot(a,b)` per il prodotto scalare $a \cdot b$,
 - `cross(a,b)` per il prodotto vettoriale $a \times b$,
 - `null(A)` per il ker, o spazio nullo $\ker(A)$,

vi sono altre funzioni particolarmente utili per l'algebra lineare: vediamo alcune.

- `det(A)` per calcolare il determinante di una matrice quadrata A ;
- `inv(A)` per calcolare l'inversa di una matrice quadrata invertibile A ;
- `rank(A)` per calcolare il rango di una matrice A ;
- `R=orth(A)` per calcolare una base ortonormale del range di A : R ha lo stesso numero di righe di A ed un numero di colonne pari al rango di A , `rank(A)`; ogni colonna è un vettore della base ortonormale;
- `[V,D]=eig(A)` per calcolare autovettori (colonne di V) ed autovalori (elementi diagonali di D) di una matrice quadrata A . Se l'argomento richiesto in uscita alla funzione è unico, `d=eig(A)`, gli autovalori sono disposti in un vettore colonna, `d`.

- Verificare direttamente le seguenti proprietà del determinante:

Operazione	Funzione	Effetto sul det
$R_i \leftrightarrow R_k$	scambia_righe(A,i,k)	$\det(\mathbf{A}) \rightarrow -\det(\mathbf{A})$
$R_i \leftarrow \lambda R_i$	moltiplica_riga(A,i,l)	$\det(\mathbf{A}) \rightarrow \lambda \det(\mathbf{A})$
$R_i \leftarrow R_i - \lambda R_k$	somma_righe(A,i,l,k)	$\det(\mathbf{A}) \rightarrow \det(\mathbf{A})$

in un caso particolare. Verificare inoltre in questo caso particolare che il determinante di una matrice triangolare superiore è pari al prodotto dei termini diagonali.

```

det_int = @(A) round(det(A)) ;

n = 3 ;
A = round( 10*rand(n,n) ) ;

% Scambio righe
fprintf( 'det_A = %d\n' , det_int(A) ) ;
A_mod = scambia_righe( A , 1 , 2 ) ;
fprintf( 'det_A dopo scambio righe = %d\n\n' , det_int(A_mod) ) ;

% Moltiplicazione riga
fprintf( 'det_A = %d\n' , det_int(A) ) ;
A_mod = moltiplica_riga( A , 1 , 10 ) ;
fprintf( 'det_A dopo moltiplicazione riga = %d\n\n' , det_int(A_mod) ) ;

% Somma di riga con multiplo di altra riga
fprintf( 'det_A = %d\n' , det_int(A) ) ;
A_mod = somma_righe( A , 1 , 10 , 2 ) ;
fprintf( 'det_A dopo somma multiplo riga = %d\n\n' , det_int(A_mod) ) ;

```

```

det_A = 134
R1  $\leftrightarrow$  R2
    2    2    8
    5    8    0
    3    1    3
det_A dopo scambio righe = -134

det_A = 134
R1  $\leftarrow$  10.00  $\times$  R1
    50    80    0
    2     2    8
    3     1    3
det_A dopo moltiplicazione riga = 1340

det_A = 134
R1  $\leftarrow$  R1+10.00  $\times$  R2
    25    28    80
    2     2    8
    3     1    3
det_A dopo somma multiplo riga = 134

```

```

% Eliminazione Gauss
fprintf( 'det_A = %d\n' , det_int(A) ) ;
A_mod = eliminazione_Gauss( A ) ;
fprintf( 'det_A dopo Gauss = %d\n' , det_int(A_mod) ) ;
prodotto_diagonale = prod( diag( A_mod ) ) ;
fprintf( 'Prodotto termini diagonali = %d\n' , round(prodotto_diagonale) ) ;

```

- Verificare in un caso particolare le seguenti proprietà per una matrice quadrata \mathbf{A} di dimensione $n \times n$:

P1. $\det(\mathbf{A}) = \lambda_1 \cdot \lambda_2 \cdots \lambda_n$, dove λ_i sono gli autovalori di \mathbf{A}

P2. $\det(\mathbf{A}^{-1}) = \det(\mathbf{A})^{-1}$

P3. gli autovalori di \mathbf{A}^{-1} valgono λ_i^{-1}

generando un output testuale per ogni proprietà.

```
n = 3 ;
A = rand(n,n) ;
err = 100*eps ;

% Determinante ed autovalori di A
det_A = det( A ) ;
lambda = sort( eig( A ) ) ;

% P1
prod_autovalori = prod( lambda ) ;
P1 = abs( det_A - prod_autovalori ) < abs(det_A)*err ;
fprintf( 'P1: det(A) = prodotto autovalori? %s\n' , si_no(P1) ) ;

% P2
inv_A = inv( A ) ;
det_i = det( inv_A ) ;
P2 = abs( det_i - 1/det_A ) < abs(det_i)*err ;
fprintf( 'P2: det(inv(A)) = 1/det(A)? %s\n' , si_no(P2) ) ;

% P3
lambda_i = sort( 1 ./ eig( inv_A ) ) ;
P3 = abs( 1 - lambda ./ lambda_i ) < err ;
fprintf( 'P3: autovalori(inv(A)) = 1/autovalori(A)? %s\n' , si_no(all(P3)) ) ;
```

- Scrivere una funzione che verifichi la continuità di una funzione $f(x)$ in un punto $x = x_0$.
- Sarebbe sufficiente verificare che il limite destro sia uguale al limite sinistro

$$f^+ = \lim_{x \rightarrow x_0^+} f(x) = \lim_{x \rightarrow x_0^-} f(x) = f^-$$

utilizzando la funzione `limite` scritta precedentemente. Tuttavia bisognerebbe fare attenzione a che valore iniziale impiegare per `delta` (basti pensare a $f(x) = H(x)$, funzione scalino, per $x_0 \neq 0$ molto vicino a 0).

- Scriviamo la funzione che effettua direttamente il controllo tra limite destro e sinistro:

is_continua.m

```
% Input: f, function handle alla funzione f
%       x0, valore di x dove si vuole controllare la continuità
% Output: b, booleano: vero se f è continua in x0, falso altrimenti
function b = is_continua( f , x0 )
    h = 0.1 ;
    salto_tollerato_y = 1e-8 ; % salto funzione
    salto_tollerato_x = abs(x0) * 10 * eps ; % salto incremento in x = h
    while abs( f(x0+h)-f(x0-h) ) > salto_tollerato_y && h > salto_tollerato_x
        h = h / 2 ;
    end
    b = h > salto_tollerato_x ;
end
```

- Scrivere una funzione che verifichi la derivabilità di una funzione $f(x)$ in un punto $x = x_0$.
- In questo caso è sufficiente verificare che la derivata destra sia uguale alla derivata sinistra

$$\lim_{h \rightarrow 0^+} \frac{f(x+h) - f(x)}{h} = \lim_{h \rightarrow 0^-} \frac{f(x+h) - f(x)}{h}$$

utilizzando la funzione `limite` scritta precedentemente.

- La derivabilità implica continuità.
- Scriviamo la funzione che effettua direttamente il controllo tra limite destro e sinistro:

is_derivabile.m

```
% Input: f, function handle alla funzione f
%       x0, valore x dove si vuole controllare la continuità della derivata
% Output: b, booleano: vero se f' è continua in x0, falso altrimenti
function b = is_derivabile( f , x0 )
    df_dx = @(h) ( f(x0+h)-f(x0) ) / h ;
    df_dx_sx = limite( df_dx , 0 , -.1 , 1e-6 , true ) ;
    df_dx_dx = limite( df_dx , 0 , +.1 , 1e-6 , true ) ;
    salto_tollerato_df_dx = 1e-5 ;
    b = abs( df_dx_sx - df_dx_dx ) < salto_tollerato_df_dx ;
end
```

- *Ricerca delle discontinuità.* Data una funzione $f(x)$, si potrebbero determinare i valori di x per i quali f è discontinua andando a fare una verifica sul modulo della derivata approssimata:

$$\left| \frac{f(x_0 + h) - f(x_0)}{h} \right| > M \quad \Rightarrow \quad f \text{ discontinua in } x_0$$

per M sufficientemente grande. In questo modo, però, qualsiasi funzione troppo pendente, cioè con $|f'| > M$, sarebbe classificata come discontinua ovunque. Ad esempio $f(x) = 2Mx$.

- Possiamo migliorare la bontà della condizione da verificare utilizzando la derivata della derivata (la velocità con cui varia la derivata), approssimata:

$$\left| \frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2} \right| > M \quad \Rightarrow \quad f \text{ discontinua in } x_0$$

- Anche quest'ultima condizione non è definitiva e può comportare problemi per funzioni con grande curvatura $|f''| > M$, ad esempio $f(x) = Mx^2$, ma ci possiamo accontentare.

- Scrivere una funzione che determini un'approssimazione dei punti di discontinuità di una data funzione f entro un dato intervallo ab , utilizzando la condizione di verifica appena vista.
- Operiamo in maniera analoga a quanto fatto nella ricerca approssimata degli zeri di una funzione mediante la ricerca del cambio del segno:

trova_discontinuita.m

```

% Input: f, function handle alla funzione f
%        ab, vettore [ a b ] dell'intervallo entro al quale lavorare
% Output: xd, matrice 2 x N, N = numero di punti di discontinuità
%         Ogni colonna di xd è un intervallo che racchiude un punto di
%         discontinuità
function xd = trova_discontinuita( f , ab )
    dx = 1e-3 ;
    x = ab(1) : dx : ab(2) ;
    f_x = f( x ) ;
    ddf = ( f_x(1:end-2) - 2*f_x(2:end-1) + f_x(3:end) ) / (dx^2) ;
    M = 1e3 ;
    ix = find( abs(ddf) > M ) ;
    xd = x(ix+1) ;
    xd = reshape( xd , 2 , length(xd) / 2 ) ;
end

```

- Utilizzo pratico: si procede in maniera analoga a quanto fatto per la ricerca degli zeri della derivata, ossia determinando prima un'approssimazione dei punti ricercati ed andando poi ad effettuare un calcolo più preciso, punto per punto.
- L'ultima parte viene eseguita cercando gli zeri della funzione

$$g(x) = f(x) - \frac{f^+ + f^-}{2}$$

con il `metodo_bisezione`, implementato sotto forma di funzione.

Main_A.m

```
% Main A: discontinuità funzione
% funzione Heaviside
f = @heaviside_f ;

% Approssimazione iniziale
x_ab = trova_discontinuita( f , [-1 1] ) ;
n_discontinuita = size( x_ab , 2 ) ;
x_discontinuita = zeros(1, n_discontinuita) ;

% Calcolo più accurato punto per punto
for i = 1 : n_discontinuita
    ab = x_ab(:,i) ;
    f_ab = f( ab ) ;
    f_mid = sum( f_ab ) / 2 ;
    g = @(x) f(x) - f_mid ;
    tol_x = 1e-10 ;
    x_discontinuita(i) = metodo_bisezione(g, ab, tol_x) ;
end
```

- *Integrali definiti.* L'integrale di una funzione $f(x)$ su un intervallo $[a, b]$ si può approssimare in termini pratici come

$$\int_a^b f(x) dx \approx \Delta x \sum_{i=1}^n f(\bar{x}_i)$$

dove l'intervallo $[a, b]$ viene suddiviso in n intervalli $I_i = [x_{i-1}, x_i]$ di uguale lunghezza $\Delta x = x_i - x_{i-1} = (b - a)/n$, $i = 1, \dots, n$. I punti \bar{x}_i possono essere scelti in maniera arbitraria dentro ai relativi intervalli I_i , $\bar{x}_i \in I_i$. Ad esempio si può scegliere

- S1. $\bar{x}_i = x_{i-1}$ (estremo sinistro);
 - S2. $\bar{x}_i = x_i$ (estremo destro);
 - S3. $\bar{x}_i = (x_{i-1} + x_i)/2$ (punto medio);
 - S4. $\bar{x}_i = (1 - \alpha)x_{i-1} + \alpha x_i$, $\alpha \in [0, 1]$, che comprende i precedenti punti. α può essere un numero random, ad esempio;
 - S5, S6. assegnare \bar{x}_i pari ad uno dei due estremi in maniera tale da ottenere un'approssimazione della somma inferiore o superiore.
- Scrivere una funzione che prenda in input una funzione $f(x)$, un intervallo $[a, b]$, un intero n ed un intero che definisce una delle 6 possibili scelte per i punti \bar{x}_i e calcoli la relativa approssimazione dell'integrale definito.

integrale_approx.m

```

function integr = integrale_approx( f , ab , n , S )
x = linspace( ab(1) , ab(2) , n+1 ) ;
x_sx = x(1:end-1) ;
x_dx = x(2:end ) ;
switch S
case 1 % estremo sx
    x_segnato = x_sx ;
case 2 % estremo dx
    x_segnato = x_dx ;
case 3 % punto medio
    x_segnato = ( x_sx + x_dx ) / 2 ;
case 4 % punto parametrizzato
    alfa = rand( size(x_sx) ) ;
    x_segnato = alfa .* x_sx + (1-alfa) .* x_dx ;
otherwise % somme inferiori o superiori
    f_x = f( x ) ;
    f_sx = f_x(1:end-1) ;
    f_dx = f_x(2:end ) ;
    if S == 5
        f_x_segnato = min( f_sx , f_dx ) ;
    else
        f_x_segnato = max( f_sx , f_dx ) ;
    end
end
if S <= 4
    f_x_segnato = f( x_segnato ) ;
end
dx = x(2) - x(1) ;
integr = dx * sum( f_x_segnato ) ;
end

```

- *Formula dei trapezi.* L'integrale di una funzione $f(x)$ su un intervallo $[a, b]$ si può approssimare con l'area delle rette che passano per ogni coppia di punti successivi:

$$\begin{aligned} \int_a^b f(x) dx &\approx \Delta x \sum_{i=1}^n \frac{f(x_{i-1}) + f(x_i)}{2} = \\ &= \frac{\Delta x}{2} \left(f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{n-1}) + f(x_n) \right) \end{aligned}$$

- *Formula di Simpson.* Lo stesso integrale si può approssimare con l'area delle parabole che passano per ogni terna di punti successivi. I punti devono quindi essere dispari (n pari):

$$\begin{aligned} \int_a^b f(x) dx &\approx 2\Delta x \sum_{i=1}^{n/2} \frac{f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})}{6} = \\ &= \frac{\Delta x}{3} \left(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 4f(x_{n-1}) + f(x_n) \right) \end{aligned}$$

- Scrivere una funzione che prenda in input una funzione $f(x)$, un intervallo $[a, b]$, un intero n ed un intero che definisce una delle 2 precedenti formule e calcoli la relativa approssimazione dell'integrale definito.

`integrale_formula.m`

```
function integr = integrale_formula( f , ab , n , formula )
    n = 2 * floor( n / 2 ) ;
    x = linspace( ab(1) , ab(2) , n+1 ) ;
    dx = x(2) - x(1) ;
    f_x = f( x ) ;
    switch formula
        case 1 % Trapezi
            estremi = f_x(1) + f_x(end) ;
            interni = 2 * sum( f_x(2:end-1) ) ;
            integr = (dx/2) * ( estremi + interni ) ;
        case 2 % Simpson
            estremi = f_x(1) + f_x(end) ;
            interni_pari = 4 * sum( f_x(2:2:end-1) ) ;
            interni_dispari = 2 * sum( f_x(3:2:end-2) ) ;
            integr = (dx/3) * ( estremi + interni_pari + interni_dispari ) ;
    end
end
```

- Confrontiamo i risultati che otteniamo impiegando i metodi appena implementati con la funzione `integral` di MATLAB:

$$I = \text{integral}(f, a, b, \text{opzioni})$$

che calcola numericamente l'integrale di f da a a b mediante formule di quadratura adattive. La tolleranza sul risultato I può essere definita in maniera assoluta o relativa mediante le opzioni:

- tolleranza assoluta: `integral(f,a,b, 'AbsTol', tol)` ;
- tolleranza relativa: `integral(f,a,b, 'RelTol', tol)` ;

Main_A.m

```
% Funzione non integrabile analiticamente
f = @(x) exp(-x.^2)
ab = [ 0 1 ] ;
n = 1000 ;

I_med = integrale_approx( f , ab , n , 3 ) ; % punto medio
fprintf( 'Integrale, punto medio: %.14e\n' , I_med ) ;

I_trap = integrale_formula( f , ab , n , 1 ) ; % trapezi
fprintf( 'Integrale, formula trapezi: %.14e\n' , I_trap ) ;

I_Simpson = integrale_formula( f , ab , n , 2 ) ; % Simpson
fprintf( 'Integrale, formula Simpson: %.14e\n' , I_Simpson ) ;

I_adapt = integral( f , ab(1) , ab(2) , 'RelTol' , 1e-15 ) ; % integral
fprintf( 'Integrale, MATLAB integral: %.14e\n' , I_adapt ) ;
```

```
f = function_handle with value:
    @(x)exp(-x.^2)
```

```
Integrale, punto medio:      7.46824163469049e-01
Integrale, formula trapezi:  7.46824071499184e-01
Integrale, formula Simpson:  7.46824132812435e-01
Integrale, MATLAB integral:  7.46824132812427e-01
```

```
f = @(x) x .^ 2
ab = [ 0 3^(1/3) ] ;
n = 1000 ;

I_sx = integrale_approx( f , ab , n , 1 ) ; % estremo sx
fprintf( 'Integrale, estremi sx:      errore = %+2e\n' , I_sx-1 ) ;

I_dx = integrale_approx( f , ab , n , 2 ) ; % estremo dx
fprintf( 'Integrale, estremi dx:      errore = %+2e\n' , I_dx-1 ) ;

I_med = integrale_approx( f , ab , n , 3 ) ; % punto medio
fprintf( 'Integrale, punto medio:     errore = %+2e\n' , I_med-1 ) ;

I_inf = integrale_approx( f , ab , n , 5 ) ; % somma inferiore
fprintf( 'Integrale, somma inferiore:  errore = %+2e\n' , I_inf-1 ) ;

I_sup = integrale_approx( f , ab , n , 6 ) ; % somme superiori
fprintf( 'Integrale, somma superiore:  errore = %+2e\n' , I_sup-1 ) ;

I_trap = integrale_formula( f , ab , n , 1 ) ; % trapezi
fprintf( 'Integrale, formula trapezi:  errore = %+2e\n' , I_trap-1 ) ;

I_Simpson = integrale_formula( f , ab , n , 2 ) ; % Simpson
fprintf( 'Integrale, formula Simpson:  errore = %+2e\n' , I_Simpson-1 ) ;
```

- *Integrali impropri e cambio di variabile.* Con la funzione `integral` è possibile anche calcolare degli integrali impropri del tipo:

$$\lim_{b \rightarrow \bar{b}} \int_a^b f(x) dx$$

dove un estremo di integrazione (o anche entrambi) può essere infinito ($\bar{b} = \pm\infty$) oppure essere punto singolare per f ($\bar{b} = c^-, c^+$).

- Esempio di integrale improprio di $f(x) = e^{-x^2}$ e cambio di variabile

$$x = \frac{t}{1-t} : [0, 1] \rightarrow [0, +\infty) \quad \Rightarrow \quad \frac{dx}{dt} = \frac{1}{(1-t)^2}$$

per trasformarlo in integrale sull'intervallo finito $[0, 1]$:

$$I = \int_0^{+\infty} e^{-x^2} dx = \int_0^1 \frac{e^{-x^2}}{(1-t)^2} dt$$

Main_B.m

```
f = @(x) exp( -x .^ 2 ) ;
I_improprio = integral( f , 0 , Inf ) ;
x_t = @(t) t ./ (1-t) ;
dx_dt = @(t) (1-t) .^ (-2) ;
I_cambio = integral( @(t) f( x_t(t) ) .* dx_dt(t) , 0 , 1 ) ;
```

che forniscono entrambi il risultato corretto $0.886226925452758 \approx \sqrt{\pi}/2$.

- *Funzione integrale.* La funzione integrale

$$F(x) = \int_{x_0}^x f(t)dt$$

si può quindi implementare direttamente come funzione (anonima):

```
F = @(x0,x) integral( f , x0 , x ) ;
```

che però ha il difetto di non poter essere richiamata su ingressi x vettoriali poichè la funzione `integral` ammette solo valori scalari.

- Scriviamo una funzione per calcolare in maniera accurata la funzione integrale $F(x)$ usando `integral` per un vettore riga x di valori di x , sommando in maniera cumulativa gli integrali tra punti x successivi:

F_integrale.m

```
% Input: f, function handle della funzione da integrare
%         x0, primo estremo dell'integrazione, scalare
%         x, vettore riga del secondo estremo d'integrazione
% Output: F, funzione integrale valutata sui valori di x
function F = F_integrale( f , x0 , x )
    I = 0*x ;
    x = [ x0 sort( x ) ] ;
    for i = 1 : length(x)-1
        I(i) = integral( f , x(i) , x(i+1) ) ;
    end
    F = cumsum( I ) ;
end
```

- Scriviamo una funzione per calcolare la funzione integrale $F(x)$ usando però la formula di Simpson invece di `integral`, sommando in maniera cumulativa gli integrali tra i punti successivi x_0, x_1, \dots, x_n del vettore colonna x di dimensione $n + 1$, con n pari.
- L'integrale tra punti successivi dovrà sempre prendere in considerazione le parabole passanti per terne di punti successivi $x_{2i-2}, x_{2i-1}, x_{2i}$, $i = 1, \dots, n/2$, ma ci servirà calcolarne separatamente l'integrale tra i primi due punti della terna x_{2i-2}, x_{2i-1} , che si ottiene utilizzando i pesi $\frac{5}{12}, \frac{8}{12}, \frac{-1}{12}$, e l'integrale tra gli ultimi due punti della terna x_{2i-1}, x_{2i} , che si ottiene utilizzando i pesi $\frac{-1}{12}, \frac{8}{12}, \frac{5}{12}$:

F_integrale_Simpson.m

```
function [ x , F ] = F_integrale_Simpson( f , x0 , x_max , n )
    n = 2 * floor( n / 2 ) ;
    x = linspace( x0 , x_max , n+1 ) ;
    dx = x(2) - x(1) ;
    f_x = f( x ) ;           % vettore di f in tutti i punti
    p = [ 5 8 -1 ] / 12 ; % pesi
    f1 = f_x(1:2:end-2) ; % vettore di f nel primo punto della terna
    f2 = f_x(2:2:end-1) ; % vettore di f nel secondo punto della terna
    f3 = f_x(3:2:end ) ; % vettore di f nel terzo punto della terna
    I_sx = p(1)*f1 + p(2)*f2 + p(3)*f3 ; % integrale prima metà
    I_dx = p(3)*f1 + p(2)*f2 + p(1)*f3 ; % integrale seconda metà
    I = zeros( 1 , n ) ;
    I(1:2:end-1) = I_sx ;
    I(2:2:end ) = I_dx ;
    F = [ 0 dx * cumsum( I ) ] ;
end
```

- Possiamo confrontare i due metodi, utilizzando anche il metodo dei trapezi che è già definito in MATLAB mediante la funzione `cumtrapz`:

$$F = \text{cumtrapz}(x, f_x)$$

che valuta $F(x)$ sul vettore x a partire dal vettore $f_x = f(x)$.

Main_C.m

```
f      = @cos ; % funzione f
F      = @sin ; % funzione integrale F, analitica
x0     = 0 ;
x_max  = 2*pi ;
err    = @(x,y) sqrt( mean( (x-y) .^ 2 ) ) ;

% MATLAB integral
n = 100 ;
x = linspace(x0, x_max, n) ;
F_int = F_integrale( f , x0 , x ) ;
err_int = err( F_int , F(x) ) ;
fprintf( 'MATLAB integral, n = %-8d, errore = %.2e\n' , n , err_int ) ;

% Metodo Simpson
n = 1000 ;
[ x , F_Simpson ] = F_integrale_Simpson( f , x0 , x_max , n ) ;
err_Simpson = err( F_Simpson , F(x) ) ;
fprintf( 'Metodo Simpson, n = %-8d, errore = %.2e\n' , n , err_Simpson ) ;

% Metodo trapezi
n = 1000 ;
x = linspace(x0, x_max, n) ;
F_trapz = cumtrapz( x , f(x) ) ;
err_trapz = err( F_trapz , F(x) ) ;
fprintf( 'Metodo trapezi, n = %-8d, errore = %.2e\n' , n , err_trapz ) ;
```

- *Equazioni differenziali ordinarie.* Un'equazione differenziale ordinaria (EDO) di ordine n è del tipo

$$F\left(x, y(x), y'(x), \dots, y^{(n)}(x)\right) = 0$$

e assumiamo che essa si può esplicitare rispetto alla derivata di ordine massimo $y^{(n)}(x)$ nella seguente forma normale:

$$y^{(n)}(x) = G\left(x, \underbrace{y(x)}_{y_1}, \underbrace{y'(x)}_{y_2}, \dots, \underbrace{y^{(n-1)}(x)}_{y_n}\right)$$

La precedente forma normale si può a sua volta esprimere come un sistema di n equazioni del prim'ordine, poste in forma normale, utilizzando delle nuove n variabili $y_i = y^{(i-1)} \Rightarrow y'_i = y_{i+1}$, $i = 1, \dots, n-1$:

$$\begin{cases} y'_1 = y_2 \\ \vdots \\ y'_{n-1} = y_n \\ y'_n = G(x, y_1, y_2, \dots, y_n) \end{cases} \Rightarrow \mathbf{y}' = \begin{Bmatrix} y_2 \\ \vdots \\ y_n \\ G(x, \mathbf{y}) \end{Bmatrix} = \mathbf{g}(x, \mathbf{y})$$

dove $\mathbf{y} = (y_i)$ è un vettore (colonna).

- La precedente forma vettoriale

$$\mathbf{y}' = \mathbf{g}(x, \mathbf{y})$$

è quella richiesta da MATLAB per la risoluzione numerica di EDO, ossia un sistema di EDO del prim'ordine posto in forma normale.

- La funzione più comune per svolgere tale calcolo è `ode45`:

$$[\mathbf{x} , \mathbf{y}] = \text{ode45}(\mathbf{g} , \mathbf{x0_xE} , \mathbf{y0})$$

dove \mathbf{g} è la funzione vettoriale $\mathbf{g}(x, \mathbf{y})$ che definisce il precedente sistema di EDO, $\mathbf{x0_xE}$ è il vettore di due componenti degli estremi dell'intervallo d'integrazione $[x_0, x_E]$ dove è richiesta la soluzione e $\mathbf{y0}$ è il vettore dei valori iniziali, ossia $\mathbf{y}(x_0)$. `ode45` restituisce in uscita la soluzione \mathbf{y} valutata per gli m valori $x_1, \dots, x_m \in [x_0, x_E]$ riportati nel vettore (colonna) \mathbf{x} ; gli estremi sono compresi: $x_1 = x_0$ e $x_m = x_E$. La soluzione \mathbf{y} è una matrice $m \times n$ dove ogni riga $\mathbf{y}(j, :)$ è il vettore delle variabili incognite valutate per $x = x_j$, ossia $\mathbf{y}(x_j)$.

- La funzione vettoriale (colonna) \mathbf{g} da implementare in MATLAB dovrà sempre essere funzione di due variabili, la prima variabile scalare x e la seconda variabile vettoriale \mathbf{y} :

Funzione anonima

```
g = @(x,y) istruzione
```

`g.m`

```
function y_primo = g(x,y)
    istruzioni
end
```

- Caso

$$y'(x) = f(x) \quad \Rightarrow \quad y(x) - y_0 = \int_{x_0}^x f(t) dt$$

ossia $\mathbf{g}(x, \mathbf{y}) = f(x)$.

- In tal caso, essendo \mathbf{g} una funzione scalare ($n = 1$) di una sola variabile scalare x , essa si può implementare direttamente mediante funzione anonima.
- Esempio con $f(x) = e^{-x^2}$, $x_0 = 0$, $x_E = 1$ e $y_0 = y(x_0) = 0$, cioè

$$y(x) = \int_0^x e^{-t^2} dt, \quad x \in [0, 1]$$

```
f = @(x) exp(-x.^2);
g = @(x,y) f(x);
x0_xE = [ 0 1 ]; % estremi d'integrazione
y0 = 0; % valore iniziale
[ x , y ] = ode45( g , x0_xE , y0 );
```

ottenendo il vettore colonna \mathbf{y} della funzione integrale $y(x)$ calcolata per i valori di x riportati nel vettore colonna \mathbf{x} .

- In questo caso specifico si può fare una verifica della soluzione appena ottenuta mediante `ode45` con la funzione integrale implementata in `F_integrale.m` dell'esercizio `Parte3_Es14`:

```
y_verifica = F_integrale( f , 0 , x );
```

- Caso

$$y'(x) = a(x)y \quad \Rightarrow \quad \log\left(\frac{y(x)}{y_0}\right) = \int_{x_0}^x a(t) dt$$

ossia $\mathbf{g}(x, \mathbf{y}) = a(x)y$.

- \mathbf{g} si può ancora implementare direttamente mediante funzione anonima.
- Esempio con $a(x) = -2x$, $x_0 = 0$, $x_E = 3$ e $y_0 = y(x_0) = 1$, cioè

$$y(x) = e^{-x^2}, \quad x \in [0, 3]$$

```
a = @(x) -2*x ;
g = @(x,y) a(x) * y ;
x0_xE = [ 0 3 ] ; % estremi d'integrazione
y0 = 1 ; % valore iniziale
[ x , y ] = ode45( g , x0_xE , y0 ) ;
```

ottenendo quindi nel vettore colonna y un'approssimazione del vettore $\exp(-x.^2)$.

- Si può quindi fare una verifica calcolando esplicitamente $y(x) = e^{-x^2}$:

```
y_verifica = exp(-x .^ 2) ;
```

- Caso

$$my''(x) + cy'(x) + ky(x) = f(x)$$

- Si pone quindi $y_1 = y$ e $y_2 = y' = y'_1$; ponendo l'equazione differenziale in forma normale si ha quindi

$$y''(x) = y'_2 = \frac{f(x) - cy'(x) - ky(x)}{m} = \frac{f(x) - cy_2 - ky_1}{m} = G(x, y_1, y_2)$$

$$\Rightarrow \mathbf{g}(x, \mathbf{y}) = \left\{ \begin{array}{c} y_2 \\ G(x, y_1, y_2) \end{array} \right\}$$

- Caso con $m = 1$, $c = 10$, $k = 10000$ e la funzione onda quadra di periodo $T = 2$ per $f(x)$:

onda_quadra.m

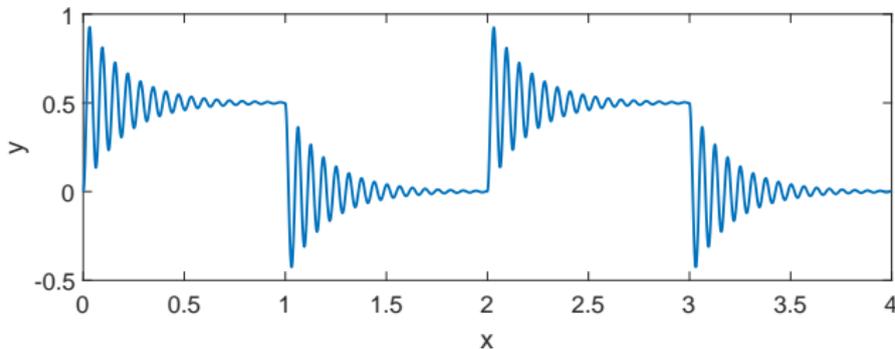
```
function y = onda_quadra(x, T)
    y = 0*x ;
    ix = mod(x, T) < T/2 ;
    y(ix) = 1 ;
end
```

```
f = @(x) 5000 * onda_quadra(x, 2) ;
m = 1 ;      % massa
c = 10 ;    % dissipazione
k = 10000 ; % costante elastica
G = @(x,y) ( f(x) - c*y(2) - k*y(1) ) / m ;
g = @(x,y) [ y(2) ; G(x,y) ] ;
x0_xE = [ 0 4 ] ; % estremi d'integrazione
y0     = [ 0 0 ] ; % valori iniziali
[ x , y ] = ode45( g , x0_xE , y0 ) ;
```

- Mediante un plot della variabile $y = y_1$ (prima colonna di y)

```
plot( x , y(:,1) ) ;
```

si ottiene:



- Nel caso di $f(x) = A \sin(\omega x)$ e $c > 0$, si può verificare che la soluzione a regime è nuovamente armonica ma con semiampiezza B

$$B = \frac{A}{|k - m\omega^2 + j\omega c|}$$

```
w = 2*pi*10 ;
f = @(x) 10000 * sin(w*x) ;
...
B = 10000 / abs( k - m*w^2 + 1i*w*c ) ;
```