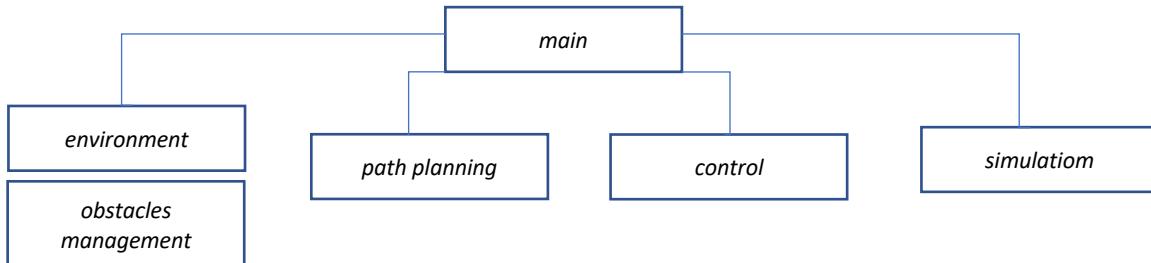


PROGETTO ROBOTICA MOBILE

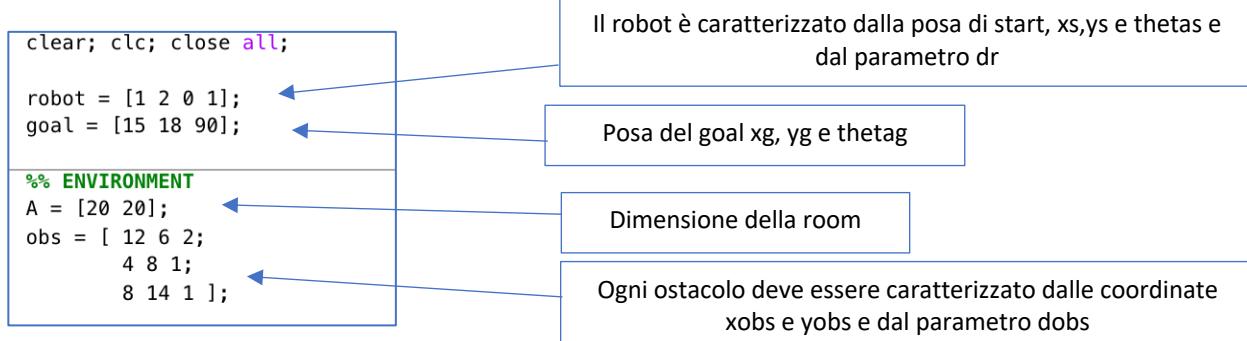
Il sistema può essere suddiviso in moduli, ognuno dei quali eseguirà un compito preciso dalla definizione dell'ambiente, al path planning, alla parte di controllo e infine con la simulazione.

Il sistema può essere sintetizzato con il seguente schema:



Si passa alla discussione dello script principale *main.m* con una configurazione di esempio, per poi andare a dettagliare nello specifico tutti i moduli che sono parte del sistema.

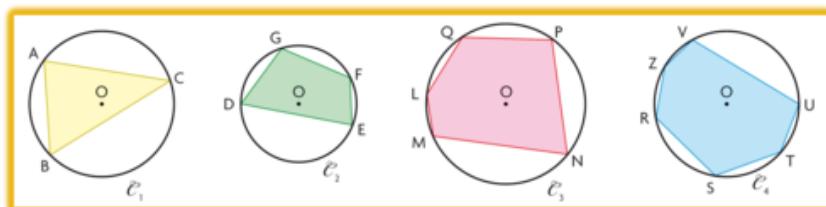
1. Definizione dello scenario di lavoro



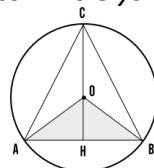
Qui c'è la possibilità da parte dell'utente di poter settare configurazioni a piacere.

Specifiche dei parametri:

- *dobs* indica il raggio della circonferenza centrato in *xobs* e *yobs* che inscrive l'ostacolo, ciò vale per ogni singolo ostacolo



- *dr* indica il raggio della circonferenza centrato in *xs* e *ys* che inscrive il robot(unicycle)



Le fasi successive con i vari metodi e plot del sistema risponderanno alla configurazione settata in precedenza.

Parte dello script dedicata alla fase di gestione degli ostacoli

```
%% OBSTACLES MANAGEMENT
abs = obstaclesManagement(robot,goal,obs);

Xs = abs(1,:);
Xg = abs(2,:);
Xobs(:, :) = abs(3:end,:);

figure("Name","Environment")
    plotEnvironment(robot,obs,Xs,Xg,Xobs,A)
hold off

fprintf('\n- Plot environment \n')
```

Plot dello scenario iniziale e della sua astrazione

L'attenzione ora si focalizzerà sul modulo *obstaclesManagement.m* che ha il compito di astrarre lo scenario iniziale, in modo da semplificare il problema e per evitare collisioni tra robot e ostacoli.

```
function ret = obstaclesManagement(robot,goal,obs)
    Xs = [robot(1) robot(2) robot(3)]; % start
    Xg = [goal(1) goal(2) goal(3)]; % goal

    dimRobot = robot(4);

    Xobs = zeros(size(obs,1),3); % obstacles
    for i=1:size(obs,1)
        Xobs(i,:) = [obs(i,1) obs(i,2) obs(i,3)+dimRobot]; % xo,yo,do
    end

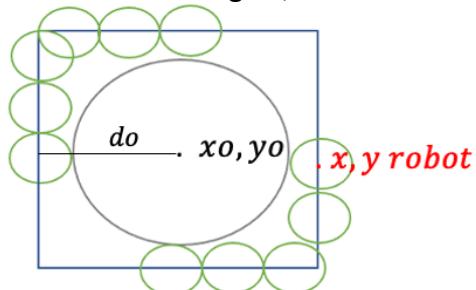
    ret = [Xs; Xg; Xobs];
end
```

Xstart = [xs,ys,thetas]

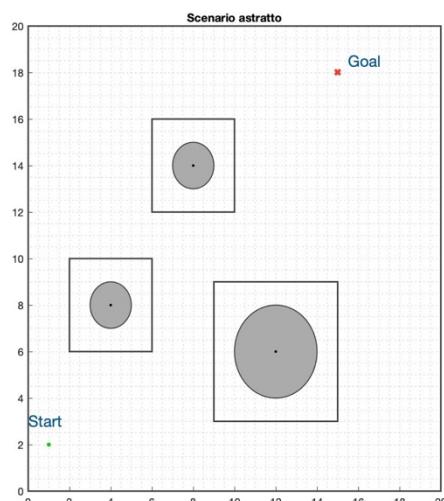
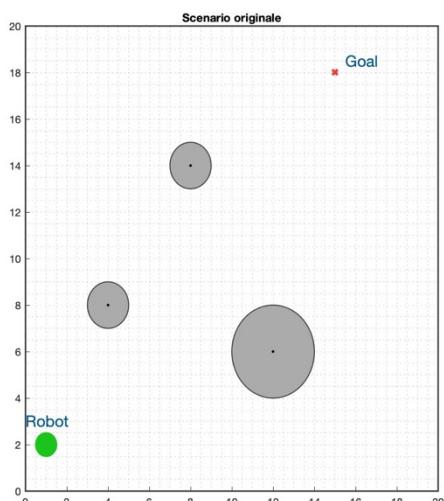
Xgoal = [xg,yg,thetag]

Xobs = [xo,yo,do]
Dove do = dobs + dr

L'idea è quella di allargare gli ostacoli, in modo tale che il robot possa essere considerato come un punto materiale e per assicurare che durante il moto verso il goal, il robot non tocchi nessun ostacolo.



Confronto tra *original environment* e *abstract environment*



2. Pianificazione della traiettoria

```
%% PATH PLANNING
which = input("\n1. Quale path planning? ");
X = pathPlanner(Xs,Xg,Xobs,A,which);

x = mySpline(X(:,1));
y = mySpline(X(:,2));

figure("Name","Path Planning")
plotPath(Xs,Xg,Xobs,x,y,X,A)
hold off

fprintf('- Plot path planning \n')
```

I°fase

II°fase

Nella I°fase si può digitare da input quale tecnica di *Path Planning* utilizzare.

Le possibili alternative sono le seguenti:

- 'APF' – *Artificial Potential Fileds*
- 'DPF' – *Discrete artificial Potential Fields*
- 'Voronoi' – *Voronoi diagram*
- 'Visibility' – *Visibility graph*

Lo script *pathPlanner.m* funge da interfaccia tra il *main* e le varie tecniche di planning

```
function X = pathPlanner(Xs,Xg,Xobs,A,which)
switch which
    case 'APF'
        X = apf(Xs,Xg,Xobs,A);
    case 'DPF'
        X = dpf(Xs,Xg,Xobs,A);
    case 'Voronoi'
        X = voronoiDiagram(Xs,Xg,Xobs,A);
    case 'Visibility'
        X = visibilityGraph(Xs,Xg,Xobs,A);
    otherwise
        exception = MException('ThisComponent:notFound','Path Planner %s not found',which);
        throw(exception);
end
X = addData(X);
```

In caso di pochi punti nella traiettoria, la arricchisco per avere un corretto funzionamento delle *spline*.

```
function X = addData(X)
lambda = 0.25;
while size(X,1) < 4
    Xnew = X(1,:) + lambda.* (X(2,:)-X(1,:));
    lambda = lambda + 0.15;
    X=[X(1,:); Xnew; X(2:end,:)];
end
end
```

Di seguito vi è la descrizione e il risultato delle singole tecniche di planning, ognuna implementata in moduli differenti ma con in comune il fine di costruire un percorso che guida il robot dal punto di *start* al *goal*.

- Artificial potential fields

```

function X = apf(Xs,Xg,Xobs,A)
Ja = @(x,y,xg,yg) 0.5*((x-xg).^2+(y-yg).^2);
dxJa = @(x,y,xg,yg) x-xg;
dyJa = @(x,y,xg,yg) y-yg;

Jr = @(x,y,xo,yo,d) 1 ./ ((x-xo).^2 + (y-yo).^2 - d.^2);
dxJr = @(x,y,xo,yo,d) -4*(x-xo) ./ ((x-xo).^2 + (y-yo).^2 - d.^2).^3;
dyJr = @(x,y,xo,yo,d) -4*(y-yo) ./ ((x-xo).^2 + (y-yo).^2 - d.^2).^3;

xx = 0:0.5:A(1);
yy = 0:0.5:A(2);
[XX,YY]=meshgrid(xx,yy);

JA = Ja(XX,YY,Xg(1),Xg(2));
dxJA = dxJa(XX,YY,Xg(1),Xg(2));
dyJA = dyJa(XX,YY,Xg(1),Xg(2));

JR = zeros(size(JA));
dxJR = zeros(size(dxJA));
dyJR = zeros(size(dyJA));

for i = 1:size(Xobs,1)
    JR = JR + Jr(XX,YY,Xobs(i,1),Xobs(i,2),Xobs(i,3));
    dxJR = dxJR + dxJr(XX,YY,Xobs(i,1),Xobs(i,2),Xobs(i,3));
    dyJR = dyJR + dyJr(XX,YY,Xobs(i,1),Xobs(i,2),Xobs(i,3));
end

wa = input("\nPeso potenziale attrattivo, wa = ");
wr = input("\nPeso potenziale repulsivo, wr = ");

J = wa*JA + wr*JR;
dxJ = wa*dxJA + wr*dyJR;
dyJ = wa*dyJA - wr*dxJR;

plotAPF(Xs,Xg,Xobs,XX,YY,J,dxJ,dyJ)

X = optimization(Xs,Xg,Xobs,dxJa,dyJa,dxJr,dyJr,wa,wr);
end

```

$$\text{Potenziale attrattivo: } J_a(X) = \frac{1}{2} \cdot \|X - X_g\|^2$$

$$\text{Potenziale repulsivo: } J_r(X, X_{obs}) = \frac{1}{(\|X - X_{obs}\|^2 - r^2)^2}$$

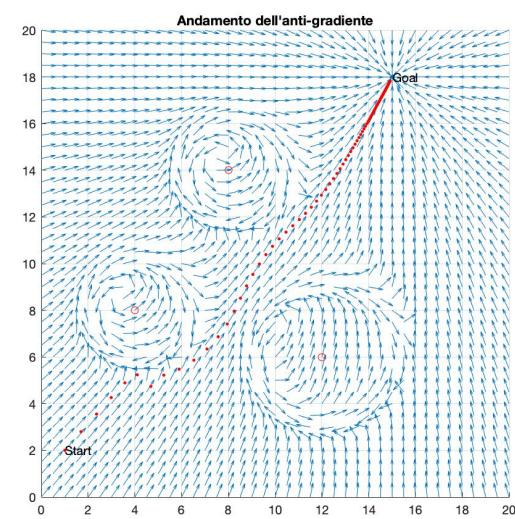
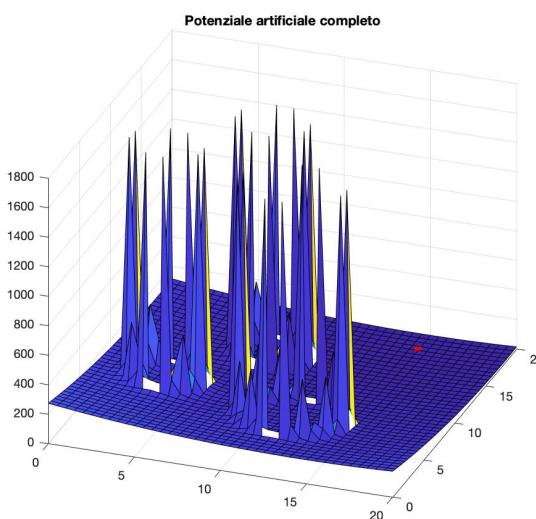
$$\text{Potenziale complessivo: } J(X) = w_a \cdot J_a(X) + w_r \cdot \sum_{X_{obs}} J_r(X, X_{obs})$$

i pesi w_a e w_r sono settabili dal *command window*

$$\text{Vortex field: } \nabla J_r = \pm \begin{pmatrix} \frac{\partial J_{r,obs}}{\partial y} \\ -\frac{\partial J_{r,obs}}{\partial x} \end{pmatrix}$$

per risolvere i problemi dei minimi locali, forzando il robot a ruotare intorno all'ostacolo.

Le ultime due funzioni servono rispettivamente per i plot legati alla tecnica dei *Potenziali artificiali* e per la costruzione del percorso dal punto di start verso il goal mediante un algoritmo di ricerca del minimo.



- *Discrete potential fields*

```

function X = dpf(Xs,Xg,Xobs,A)
    buildGrid(Xg,Xobs,A); ◀
    n = input("      Norma: ");
    if n ~= 1 && n ~= 2
        exception = MException('ThisComponent');
        throw(exception);
    end
    X = resolve(Xs,Xg,n); ◀
    figure("Name","Discrete Potential Field");
        plotGrid(X,Xs,Xg)
    hold off
end

%% Build DPF
function buildGrid(Xg,Xobs,A) [...]
function expand(x,y,value) [...]
function infect(x,y,INFINITY) [...]

%% Resolve DPF
function ret = resolve(Xs,Xg,n) [...]

%% Metodi per leggere/modificare la griglia
function setGrid(mtx) [...]
function ret = getGrid [...]
```

Per la costruzione della griglia

Per costruire il percorso da *start* a *goal*

La function *buildGrid* costruisce la griglia facendo uso dei metodi:

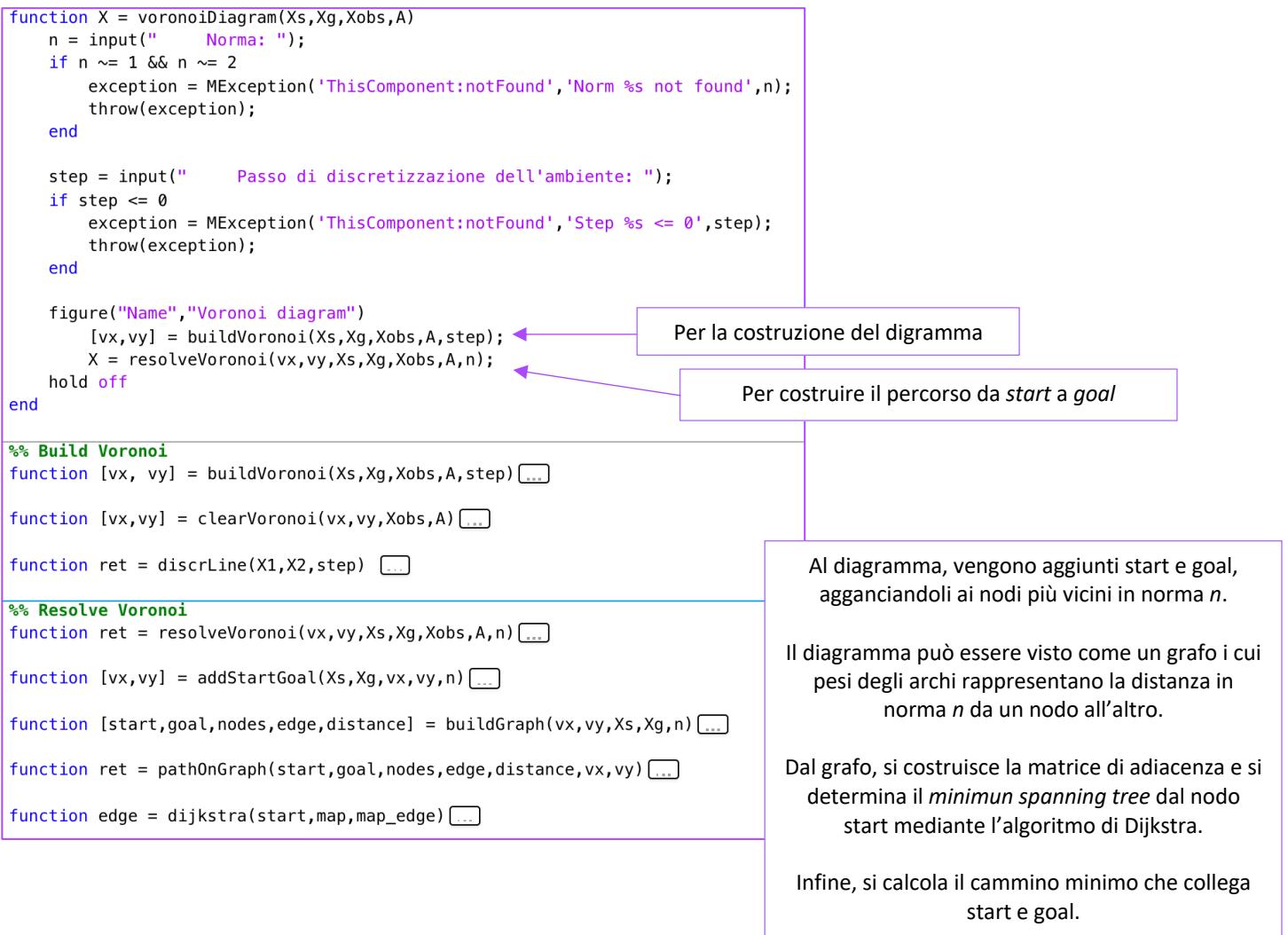
- *expand*, che istanzia dal punto di minimo globale (goal) tutte le celle della griglia, espandendosi, creando le *linee di livello* il cui valore dipende dalla distanza della cella dal goal.
 - *infect* ha il compito di infettare tutte le celle occupate dagli ostacoli, settando come valore infinito.

Rispetto al *Artificial Potential Fields*, *expand* funge da potenziale attrattivo, mentre *infect* da potenziale repulsivo.

In input è possibile settare il parametro n che rappresenta la tipologia di *norma*, utile per la fase di costruzione del *path*.

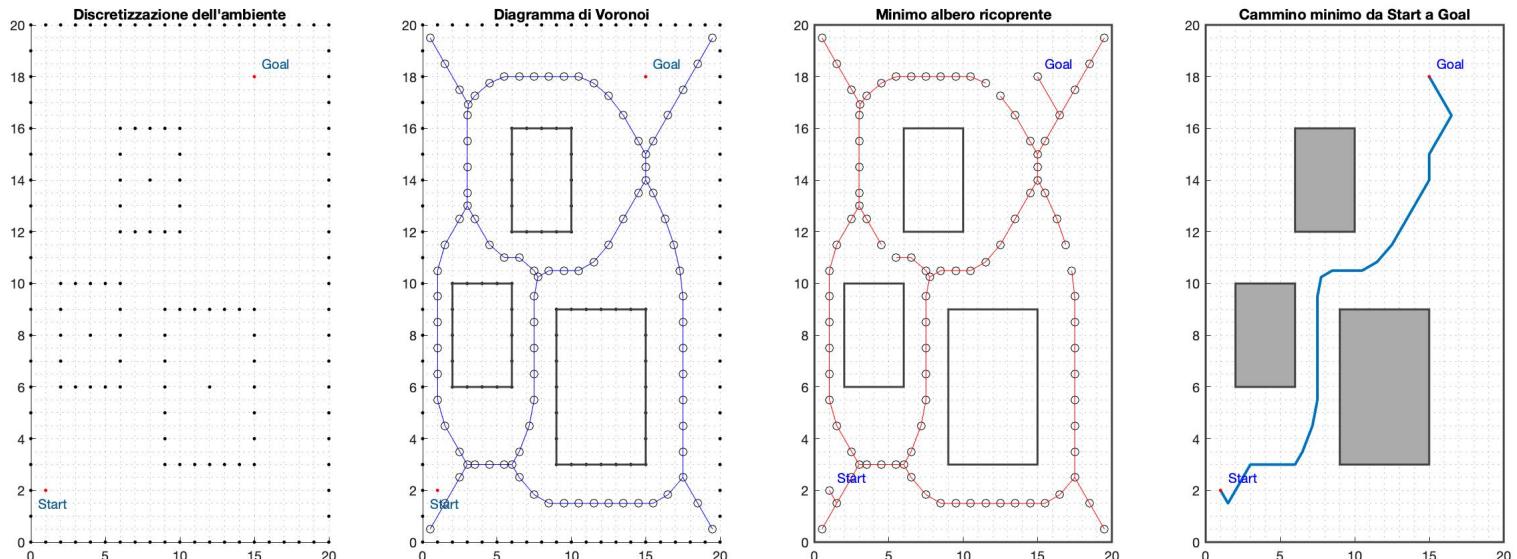
La *norma* serve a determinare quale delle celle adiacenti alla mia posizione corrente (con valore < Inf) considerare con distanza minima dal goal; l'uso della norma può essere paragonato all'uso del gradiente nell' APE.

- Voronoi diagram



In input è possibile settare oltre al parametro n , che rappresenta la tipologia di *norma* da utilizzare per pesare gli archi del grafo, anche il parametro $step$, utile per decidere con che passo discretizzare l'ambiente.

È consigliabile scegliere un valore di $step$ compreso tra $(0.5,1]$.



- *Visibility graph*

```

function X = visibilityGraph(Xs,Xg,Xobs,A)
    n = input("      Norma: ");
    if n ~= 1 && n ~= 2
        exception = MException('ThisComponent:notFound','Norm %s not found',n);
        throw(exception);
    end

    figure("Name","Visibility Graph")
    [vx,vy] = buildVisibility(Xs,Xg,Xobs,A);
    X = resolveVisibility(vx,vy,Xs,Xg,Xobs,A,n);
    hold off
end

%% Build Graph
function [vx,vy]=buildVisibility(Xs,Xg,Xobs,A) ...

%% Resolve Graph
function path = resolveVisibility(vx,vy,Xs,Xg,Xobs,A,n) ...

```

Per costruire il grafo

Per costruire il percorso da *start* a *goal*

Si costruisce il grafo di visibilità, creando collegamenti tra spigoli della room, spigoli degli ostacoli, start e goal

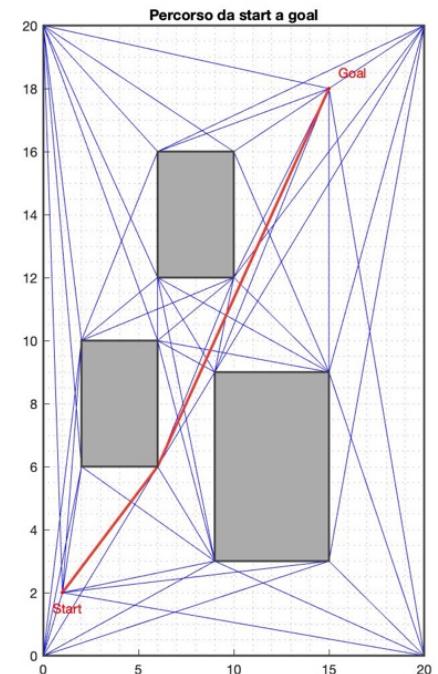
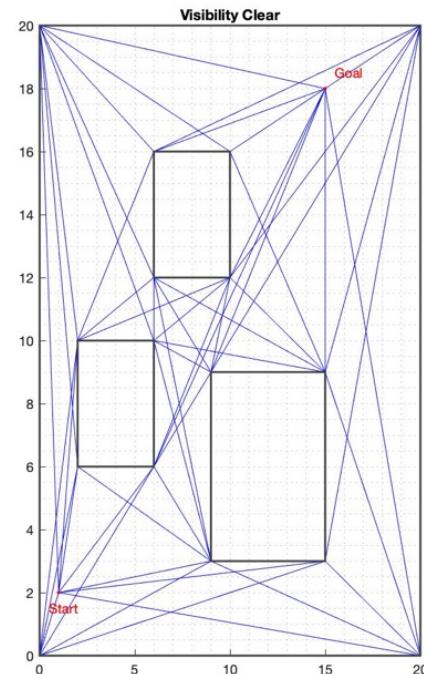
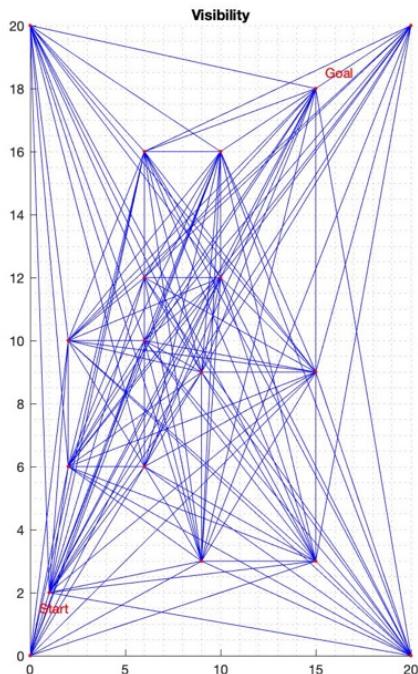
Per ripulire il grafo dagli archi che intersecano almeno un ostacolo, si usa il metodo *canSee*, che per ogni *Xobs* dell'ambiente e per ogni coppia di nodi del grafo, restituirà vero se i due nodi si 'vedono' senza intersecare l'ostacolo.

Per un arco basta che questo intersechi con almeno un ostacolo per essere eliminato dal grafo.

Dal grafo, si costruisce la matrice di adiacenza e si determina il *minimum spanning tree* dal nodo *start* mediante l'algoritmo di Dijkstra.

Infine, si calcola il cammino minimo che collega *start* e *goal*.

In input è possibile settare il parametro che rappresenta la tipologia di *norma* da utilizzare per pesare gli archi del grafo.



Nella II° fase del problema di pianificazione della traiettoria

```
x = mySpline(X(:,1));
y = mySpline(X(:,2));

figure("Name","Path Planning")
plotPath(Xs,Xg,Xobs,x,y,X,A)
hold off

fprintf('- Plot path planning \n')
```

II° fase

Con l'uso delle *spline* si vuole ottenere un insieme di *polinomi* in grado di interpolare i punti della traiettoria calcolata nella I° fase di *planning*.

```
function p = mySpline(y)
xx = 0:length(y)-1;
pp = spline(xx,y(:));
p = {};
for k = 1:pp.pieces
    xk = pp.breaks(k);
    ak = pp.coefs(k,1);
    bk = pp.coefs(k,2);
    if pp.order < 3
        ck = 0;
    else
        ck = pp.coefs(k,3);
    end
    if pp.order < 4
        dk = 0;
    else
        dk = pp.coefs(k,4);
    end
    if k == 1
        p{k,1} = @(x) ((dk + ck*(x-xk) + bk*(x-xk).^2 + ak*(x-xk).^3).* (x~=xk)) + (y(1) .* (x==xk)); % p
        p{k,2} = @(x) (ck + 2*bk*(x-xk) + 3*ak*(x-xk).^2); % pdot
        p{k,3} = @(x) (2*bk + 6*ak*(x-xk)); % pdotdot
    else
        p{k,1} = @(x) ((dk + ck*(x-xk) + bk*(x-xk).^2 + ak*(x-xk).^3).* (x~=xk)) + (y(k) .* (x==xk)); % p
        p{k,2} = @(x) ((ck + 2*bk*(x-xk) + 3*ak*(x-xk).^2).* (x~=xk)) + (p{k-1,2}(xk) .* (x==xk)); % pdot
        p{k,3} = @(x) ((2*bk + 6*ak*(x-xk)).* (x~=xk)) + (p{k-1,3}(xk) .* (x==xk)); % pdotdot
    end
end
% p{1,1} polinomio, p{1,2} pdot, p{1,3} pdotdot
% p{1,1}(t) to call
end
```

Data la sequenza di punti x_1, x_2, \dots, x_n

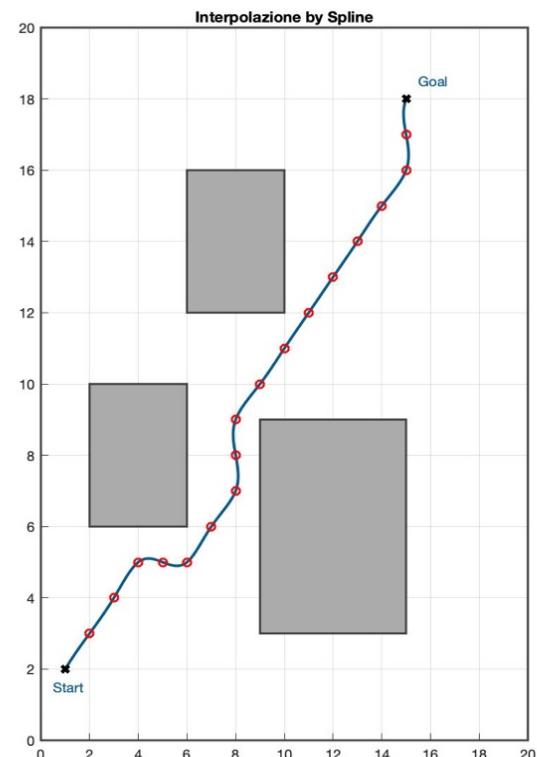
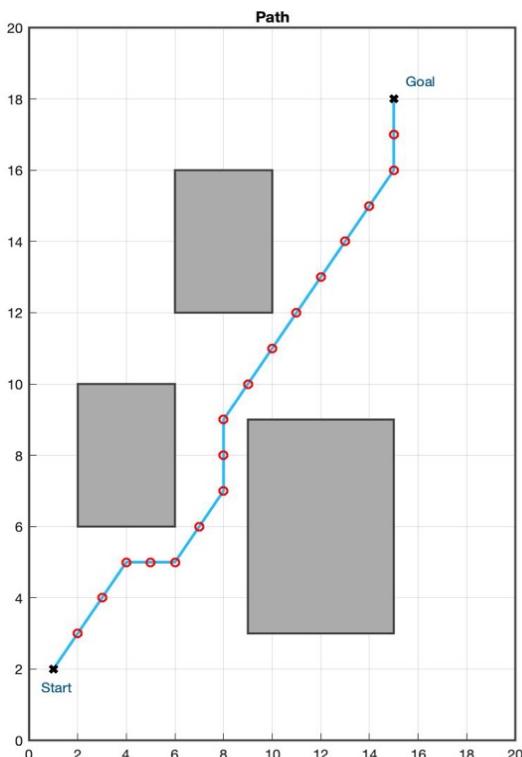
Si costruisce un oggetto $struct_{nx3}$ di *function handle* per cui ogni riga è composta dai seguenti polinomi:

$$\begin{aligned} p(x) &= p(k,1) = a_k \cdot (x - x_k)^3 + b_k \cdot (x - x_k)^2 + c_k \cdot (x - x_k) + d_k \\ \dot{p}(x) &= p(k,2) = 3 \cdot a_k \cdot (x - x_k)^2 + 2 \cdot b_k \cdot (x - x_k) + c_k \\ \ddot{p}(x) &= p(k,3) = 6 \cdot a_k \cdot (x - x_k) + 2 \cdot b_k \end{aligned}$$

$$\forall x_k, x_{k+1} \text{ con } k = 0, 1, \dots, n - 1$$

Sono imposti vincoli di continuità fino al 2° ordine di derivazione tra un intervallo e il suo 'precedente'.

Con il metodo *plotPath.m* si può visualizzare come l'uso delle spline rendano la traiettoria più liscia.



3. Controllo

In questa parte di codice della fase di controllo, vi è la possibilità dal *command window* di digitare quale strategia di controllo utilizzare.

```
%% CONTROL
whichT = input("\n2. Quale trajectory control? ");
paramT = {};
switch whichT
    case 'Linear'
        a = input("      a > 0 :");
        delta = input("      delta compreso tra (0,1) :");
        paramT{1} = a; paramT{2} = delta;
    case 'NotLinear'
        K1 = input("      K1(v,w) > 0 bounded :");
        K2 = input("      K2 > 0 :");
        K3 = input("      K3(v,w) > 0 bounded :");
        paramT{1} = K1; paramT{2} = K2; paramT{3} = K3;
    case 'IOLinear'
        b = input("      b~=0 :");
        Kx = input("      Kx>0 :");
        Ky = input("      Ky>0 :");
        paramT{1} = b; paramT{2} = Kx; paramT{3} = Ky;
end

whichP = input("\n3. Quale posture regulation? ");
paramP = {};
switch whichP
    case 'Complete'
        K1 = input("      K1 > 0 :");
        K2 = input("      K2 > 0 :");
        K3 = input("      K3 :");
        paramP{1} = K1; paramP{2} = K2; paramP{3} = K3;
    case 'Cartesian'
        K1 = input("      K1 > 0 :");
        K2 = input("      K2 > 0 :");
        paramP{1} = K1; paramP{2} = K2;
end

flag = input("\n Raggio dell'intorno centrato in Xg: ");
RHO = @(X,Xc,d) (X(1)-Xc(1))^2 + (X(2)-Xc(2))^2 > d^2;
```

La strategia di controllo è suddivisa nella fase di *trajectory tracking* e *posture regulation*, ognuna delle quali può essere implementata in modo diverso.

A seconda della tipologia di controllore scelta, vi è la possibilità in input di settare i parametri caratteristici di quel controllore.

- Implementazione *trajectory tracking*
 1. Controllo 'Linear'
 2. Controllo 'NotLinear'
 3. Controllo 'IOLinear'

- Implementazione *posture regulation*
 1. Regolazione 'Complete'
 2. Regolazione 'Cartesian'

Per schedulare l'azione delle due strategie di controllo, si è deciso di costruire un intorno centrato nel goal, tale per cui sul bordo dell'intorno vi è l'arresto del *trajectory tracking* e l'avvio del *posture regulation*.



Con il *trajectory tracking* si arriva in prossimità del punto di goal, mentre con la *posture regulation*, si regola l'orientamento sul goal secondo le specifiche desiderate.

```
ROBOT = @(t,X) ( trajectoryControl(X,piece(t,x,y),whichT,paramT).*RHO(X,Xg,flag)+ ...
    postureRegulation(X,Xg,whichP,paramP).*(~RHO(X,Xg,flag)) );
```

Implementazione del controllo completo

Nella parte di controllo si è fatto riferimento al modello di robot mobile *unicycle*.

- *trajectoryTracking.m*

```

function Xdot = trajectoryControl(X,piece,which,param)
    x = X(1);
    y = X(2);
    theta = X(3);

    xstar = piece(1); ystar = piece(2);
    xdotstar = piece(3); ydotstar = piece(4);
    xddotstar = piece(5); yddotstar = piece(6);

    thetastar = atan2(ydotstar,xdotstar);

    vstar = sqrt(xdotstar^2 + ydotstar^2);
    wstar = (yddotstar*xdotstar - xddotstar*ydotstar)/(xdotstar^2 + ydotstar^2);
    switch which
        case 'Linear'
            a=param{1}; delta=param{2};
            [v,w] = linear(x,y,theta,xstar,ystar,thetastar,vstar,wstar,a,delta);
        case 'NotLinear'
            K1=param{1}; K2=param{2}; K3=param{3};
            [v,w] = notLinear(x,y,theta,xstar,ystar,thetastar,vstar,wstar,K1,K2,K3);
        case 'IOLinear'
            b=param{1}; Kx=param{2}; Ky=param{3};
            [v,w] = IOLinear(x,y,theta,xstar,ystar,thetastar, ...
                xdotstar,ydotstar,xddotstar,yddotstar,b,Kx,Ky);
        otherwise
            exception = MException('ThisComponent:notFound','Controlllore %s non trovato',which);
            throw(exception);
    end
    Xdot = [ v*cos(theta);
              v*sin(theta);
              w ];
end

%% Controllo lineare
function [v,w] = linear(x,y,theta,xstar,ystar,thetastar,vstar,wstar,a,delta) ...  
...

%% Controllo non lineare
function [v,w] = notLinear(x,y,theta,xstar,ystar,thetastar,vstar,wstar,K1,K2,K3) ...  
...

%% Controllo IO linearizzato
function [v,w] = IOLinear(x,y,theta,xstar,ystar,thetastar,... ...  
...

```

$$X(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix}$$

Istanziò i valori dalla traiettoria di riferimento costruita mediante *spline*

$$\dot{X}(t) = \begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \dot{\theta}(t) \end{bmatrix} = \begin{bmatrix} v(t) \cdot \cos(\theta(t)) \\ v(t) \cdot \sin(\theta(t)) \\ \omega(t) \end{bmatrix}$$

Lo script ha il compito di interfacciare il *main* con le varie strategie di controllo per il *trajectory tracking*.

Tutte le tecniche di controllo restituiscono come output i valori di $v(t)$ e $\omega(t)$

La function *trajectoryControl.m* prende come parametro i punti della traiettoria di riferimento X e l'output della chiamata al metodo *piece(t,x,y)*.

```

function xy = piece(t,x,y)
    k = size(x,1);
    if t == 0
        k = 1;
    elseif ceil(t) < k
        k = ceil(t);
    end
    xy = [ x{k,1}(t); y{k,1}(t);
           x{k,2}(t); y{k,2}(t);
           x{k,3}(t); y{k,3}(t) ];
end

```

Considerando il k -esimo polinomio cubico delle *spline* x e y , tale per cui l'istante t appartiene a quel k -esimo intervallo, allora si hanno i valori di:

$$\begin{array}{ll} x(t) & y(t) \\ \dot{x}(t) & \dot{y}(t) \\ \ddot{x}(t) & \ddot{y}(t) \end{array}$$

Si ricorda come ogni riga delle *spline* costruite, contengono tre *function handle* per ogni riga

$$p(t) \quad \dot{p}(t) \quad \ddot{p}(t)$$

- *postureRegulation.m*

```

function Xdot = postureRegulation(X,Xg,which,param)
    x = X(1);
    y = X(2);
    theta = X(3);

    xstar = Xg(1);
    ystar = Xg(2);
    thetastar = Xg(3);

    ex = xstar-x;
    ey = ystar-y;
    etheta = thetastar-theta;

    switch which
        case 'Complete'
            K1=param{1}; K2=param{2}; K3=param{3};
            [v,w]=completeRegulation(ex,ey,etheta,theta,K1,K2,K3);
        case 'Cartesian'
            K1=param{1}; K2=param{2};
            [v,w]=cartesianRegulation(ex,ey,theta,K1,K2);
        otherwise
            exception = MException('ThisComponent:notFound', 'Controllore %s non trovato',which);
            throw(exception);
    end

    Xdot = [ v*cos(theta);
              v*sin(theta);
              w ];

```

$X(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix}$

Considero i valori in corrispondenza del goal come riferimento.

Errore sul riferimento.

$\dot{X}(t) = \begin{bmatrix} \dot{x}(t) \\ \dot{y}(t) \\ \dot{\theta}(t) \end{bmatrix} = \begin{bmatrix} v(t) \cdot \cos(\theta(t)) \\ v(t) \cdot \sin(\theta(t)) \\ \omega(t) \end{bmatrix}$

```

%% Controllore completo
function [v,w]=completeRegulation(ex,ey,etheta,theta,K1,K2,K3) ...
```

```

%% Controllore cartesiano
function [v,w]=cartesianRegulation(ex,ey,theta,K1,K2) ...
```

Come per il *trajectoryTracking.m*, il compito di questo script è di interfacciare il *main* con le varie strategie di controllo per la *posture regulation*.

Tutte le tecniche di controllo restituiscono come output i valori di $v(t)$ e $\omega(t)$

4. Simulazione

Infine, vi è la fase di simulazione:

```

%% SIMULATION
tsim = 0:size(X,1);
[t,sol] = ode45(ROBOT,[tsim(1)+0.0001 tsim(end)-0.001],[Xs(1);Xs(2);Xs(3)]);

figure("Name","Control")
plotControl(Xs,Xg,flag,Xobs,x,y,sol,A)
fprintf('\n- Plot control \n')

pause

figure("Name","Simulation")
plotSimulation(robot,Xg,obs,sol,A)
hold off
fprintf('\n- Plot simulation \n')
```

Sul primo plot si effettua un confronto nell'ambiente astratto della traiettoria di riferimento calcolata nella fase di *path planning* e della traiettoria che seguirà il robot *unicycle*, mentre con l'ultimo plot si effettua una vera e propria simulazione sullo scenario iniziale.

Di seguito un esempio di impostazione da *command window* con la relativa simulazione.

```
Command Window
#Attenzione: Input di tipo stringa tra apici singoli.

- Plot environment

1. Quale path planning? 'DPF'
   Norma: 1

- Plot path planning

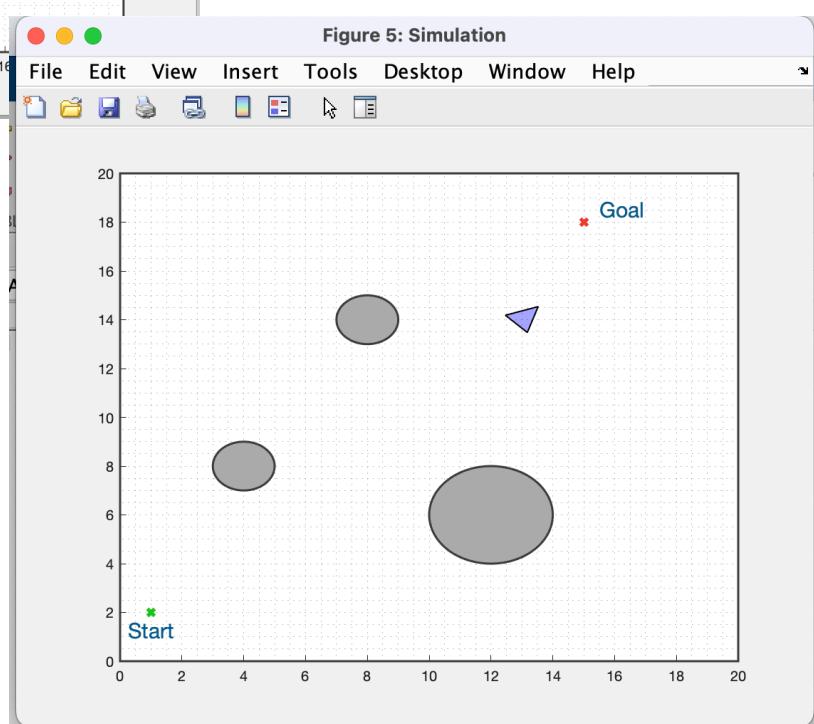
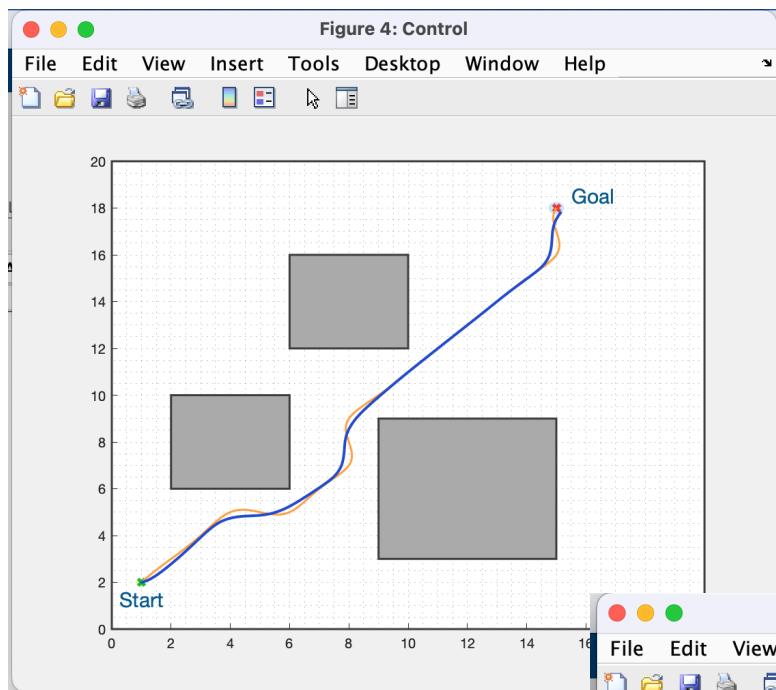
2. Quale trajectory control? 'IOLinear'
   b~>0 :0.95
   Kx>0 :1.2
   Ky>0 :1.6

3. Quale posture regulation? 'Complete'
   K1>0 :1.5
   K2>0 :1.7
   K3 :0.1

>> Raggio dell'intorno centrato in Xg: 0.25

- Plot control

- Plot simulation
```



Un ulteriore esempio completo del sistema con una configurazione differente di start e goal:

```
Command Window
#Attenzione: Input di tipo stringa tra apici singoli.

- Plot environment
1. Quale path planning? 'Visibility'
Norma: 2

- Plot path planning

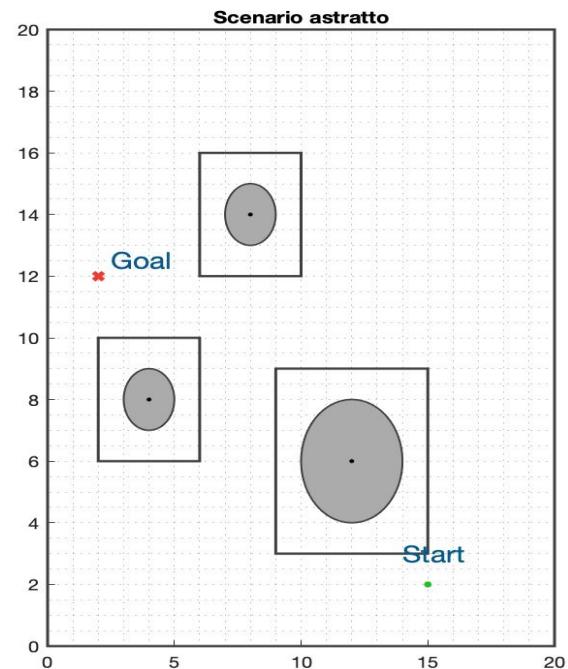
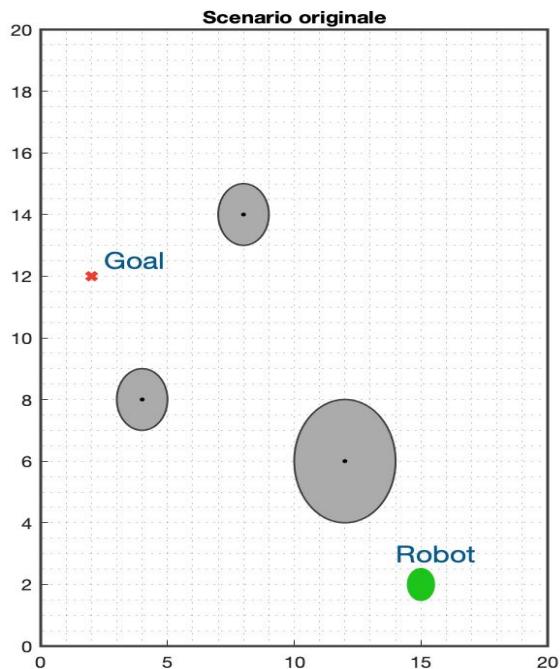
2. Quale trajectory control? 'IOLinear'
b~0 :0.8
Kx>0 :1.2
Ky>0 :1.78

3. Quale posture regulation? 'Cartesian'
K1>0 :1.5
K2>0 :0.75

>> Raggio dell'intorno centrato in Xg: 0.25

- Plot control
- Plot simulation
```

%% ENVIRONMENT



%% PATH PLANNING

