UNIVERSITÀ
DELLA CALABRIA

Relazione Robotica 3

Vaccaro Francesca 239641
Anno accademico 2024/2025

Progetto Robotica

Vaccaro Francesca 239641

L.T. Ingegneria Informatica 24/25

Contents

1 Introduzione

1.1 Obiettivi

Il seguente progetto si pone di andare a sperimentare dei metodi differenti di mapping di un ambiente, lo scopo è quello di far muovere il robot da un punto A ad un punto B(quindi da un punto di start ad un punto di finish), senza andare ad attraversare quelli che sono degli ostacoli piazzati all'interno dell'ambiente. Oltre al punto di inizio e a quello di fine viene anche stabilita una *posa* di inizio e una di fine.

1.2 Descrizione del robot

Il robot che ci viene fornito è un robot ***dual drive***, questo implica che le due ruote di cui dispone siano indipendenti tra loro, inoltre ci vengono anche dati dei dettagli tecnici sulla composizione del robot, in particolare :

1. raggio di 0.15m.
2. distanza tra le ruote = 0.26 m.
3. raggio delle ruote = 0.05 m.
4. massimo avanzamento lineare = $\frac{1}{10}\frac{m}{s}$.

Il robot risulta inoltre essere dotato di sensori.

2 Creazione dell'ambiente

L'ambiente su cui andremo a lavorare è di forma rettangolare e ha dimensioni : $7.5m \cdot 10m$; gli ostacoli da inserire sono anche questi rettangolari e di dimensione $1m \cdot 2m$ e dovranno essere 4 in posizioni arbitrarie nell'ambiente. La posa di **partenza** è $(0.5, 0.5, 0)^T$ mentre quella di **arrivo** è $(9.5, 7, \pi)^T$. Andiamo a creare in matlab l'ambiente seguendo le specifiche, e tenendo conto del fatto che considereremo il robot come un punto materiale, quindi dovremo andare ad "ingrossare" gli ostacoli di due volte il raggio, quindi se in principio gli ostacoli avevano dimensione 1m in altezza e 2m in larghezza, adesso saranno di 2 m in altezza e 3 in larghezza. Per fare tutto questo in matlab, ci sono diverse strade, una potrebbe essere quella di creare l'ambiente manualmente disegnando gli ostacoli, e lavorando manualmente, andando a scrivere le varie funzioni. In alternativa si potrebbe ricorrere al tool proposto da matlab, che vede il suo utilizzo principale proprio nella robotica. Nonostante inizialmente la strada intrapresa virava verso la prima opzione, per questo progetto ho deciso in seguito di utilizzare il tool.

Il codice è il seguente :

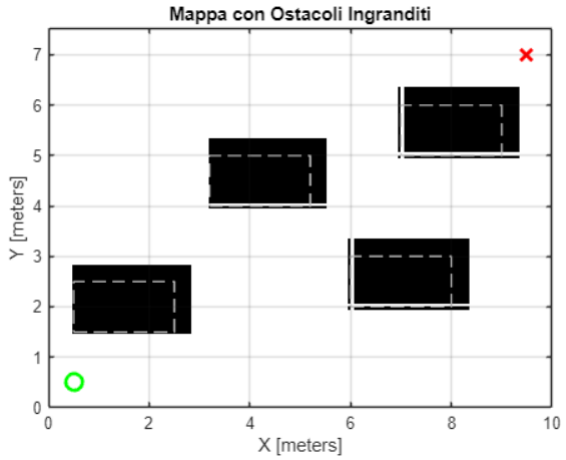
```
%%Propriet  Robot
raggioRuote = 0.05;
distanzaRuote = 0.26;
maxLinearSpeed = 0.1;     % m/s
maxAngularSpeed = maxLinearSpeed/(distanzaRuote/2);  % rad/s
```

```
robotRadius = 0.15;        % m

%% Mappa e Ostacoli
mapWidth = 10;
mapHeight = 7.5;
mapResolution = 10; % celle per metro
map = binaryOccupancyMap(mapWidth, mapHeight, mapResolution);

%posizione degli ostacoli la singola riga della matrice    : (x, y,
   largezza, altezza)
obstacles = [
    0.5, 1.5, 2, 1;
    6, 2, 2, 1;
    3.2, 4, 2, 1;
    7, 5, 2, 1
];


% Inflazione degli ostacoli per tenere conto delle dimensioni del robot
inflatedObstacles = obstacles;
inflatedObstacles(:,3:4) = inflatedObstacles(:,3:4) + 2*robotRadius;


for i = 1:size(inflatedObstacles,1)
    rect = inflatedObstacles(i,:);
    x = rect(1); y = rect(2); w = rect(3); h = rect(4);

    % Creazione di una mesh pi  densa per coprire tutta l'area
    xVals = linspace(x, x + w, ceil(w * mapResolution));
    yVals = linspace(y, y + h, ceil(h * mapResolution));
    [X, Y] = meshgrid(xVals, yVals);
    points = [X(:), Y(:)];

    setOccupancy(map, points, 1);
end



%% Pose Iniziale e Finale
startPose = [0.5, 0.5, 0]; % [x, y, theta]
goalPose = [9.5, 7, pi];  % [x, y, theta]
```

```
%% Creazione del Grafo di Visibilit
[vertices, adjacencyMatrix, costMatrix] = createVisibilityGraph(map,
    inflatedObstacles, startPose, goalPose, maxLinearSpeed);

%% Pianificazione del Percorso con Dijkstra
[pathIndices, totalTime, ~] = findPathWithDijkstra(vertices,
    adjacencyMatrix, maxLinearSpeed, maxAngularSpeed, startPose, goalPose);

%% Visualizzazione dei Risultati
plotResults(map, obstacles, inflatedObstacles, startPose, goalPose,
    vertices, adjacencyMatrix, pathIndices, totalTime);
```

Quindi, andando nel dettaglio, prima di si sono settate le dimensioni dell'ambiente, e con esse anche la risoluzione, ovvero il numero di celle per metro, questo perchè andremo a mettere il tutto all'interno di una binaryOccupancyMap crea un oggetto mappa di occupazione 2D, dove ogni cella della griglia avrà un booleano che indica se questa è occupata(true) oppure libera(false). Successivamente quello che si va a fare è il posizionamento degli ostacoli, viene dunque creata una matrice che prende come elementi le coordinate relative al piazzamento dell'oggetto e le dimensioni; in questo specifico caso si aveva una libera scelta sulla decisione delle posizioni, benchè ricreassero un ambiente non troppo semplice per il robot. Partendo proprio dalla mappa degli ostacoli si estraggono le singole informazioni, e si vanno a convertire in indici di griglia, in quanto è proprio qui che dovranno essere utilizzati. Per visualizzare la mappa, si opta di far visionare oltre che gli ostacoli, anche il punto di partenza e quello di arrivo, inizialmente si era pensato anche ad una leggenda(difatti è commentata), ma ai fini grafici risultava scomoda.

Ecco l'immagine che viene fuori da questo script :



Come dall'immagine si può notare gli ostacoli sono stati "ingrossati", in accordo con le dimensioni del raggio del robot. Così da poter considerare questo come un punto materiale.

3 Implementazione dei vari metodi

3.1 Visibility graph

Come prima annunciato per andare a visualizzare il grafo, dobbiamo andare a prendere quelli che sono i vertici validi, ovvero quelli relativi agli **ostacoli** e quelli relativi alle pose di **inizio e fine**, poi questi vertici li andremo a collegare tutti tra di loro con dei segmenti per cominciare a delineare un grafo, però questa operazione ha bisogno di una "pulizia", perchè quello che si ricerca non sono tutti i segmenti che si possono trovare dalla combinazione dei vertici, dovremo infatti andare ad eliminare quei segmenti che si trovano dentro agli ostacoli. Per procedere a questa operazione di pulizia, dobbiamo arrivare alla radice del problema, e dunque in che caso un segmento attraversa un ostacolo? Quando almeno un punto di quel segmento si trova in una zona occupata, ma per prendere punto per punto un segmento su matlab bisogna in qualche modo campionarlo. Questa operazione viene fatta per ogni segmento e quindi per ogni congiunzione da punto a punto nella mappa.

Nel mio script lo faccio nelle varie funzioni di supporto o ausiliarie, definite nella parte finale.

Altra cosa importante è andare a trovare il percorso migliore nel grafo, si deve però capire quale è la concezione di "migliore" che si vuole dare, quindi rispetto a quale parametro si vuole ottimizzare; in particolar modo si possono valutare due scelte di progettazione, la prima riguarda una minimizzazione rispetto alla distanza percorsa, la seconda invece è leggermente più complessa ma molto probabilmente più incline a quello che è l'obbiettivo ultimo del progetto, si tratta infatti di andare a minimizzare il tempo in termini di rotazioni e traslazioni, dunque una volta calcolato il tempo impiegato per andare "dritti" e per ruotare, bisogna scegliere il percorso che ne minimizzi la tempistica.

Per fare questo bisogna ricordarsi delle proprietà fisiche e intrinseche del robot e delle pose di ogni nodo(bisohna capire quando si sta compiendo una rotazione e di quanto soprattutto). La velocità lineare massima è di $0.1\frac{m}{s}$, dunque :

$$\omega_{max} = v_{max}/r_{curv}$$

Dove r_{curv} sta per raggio di curvatura. Altro dettaglio importante da tenere a mente è che quando viene fatta una rotazione quello che accade alle ruote(sinistra e destra) è :

$$\omega_l = -\omega_r$$

Mentre se si muove sul rettilineo :

$$\omega_l = \omega_r$$

Questo perchè la velocità angolare viene calcolata come :

$$\omega_r = \frac{v + \frac{d}{2} \cdot \omega}{r}$$

$$\omega_l = \frac{v - \frac{d}{2} \cdot \omega}{r}$$

Dove d = distanza tra ruote, r = raggio ruote e v = velocità lineare.

```
%% === Ricostruzione traiettoria ===
stati = zeros(length(tempo), 3);
x = startPose(1); y = startPose(2); theta = startPose(3);
stati(1,:) = [x y theta];

for i = 2:length(tempo)
    dt = tempo(i)-tempo(i-1);
    x = x + v(i-1)*cos(theta)*dt;
    y = y + v(i-1)*sin(theta)*dt;
    theta = theta + omega_z(i-1)*dt;
    stati(i,:) = [x y theta];
end

%% === Grafico traiettoria e velocit  ===
velocita = [v omega_z];
plotCinematica(tempo, stati, velocita, omega_l, omega_r);

%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% ======================== FUNZIONI ======================================
%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


function [vertices, adjMat, costMat] = createVisibilityGraph(map,
    obstacles, startPose, goalPose, maxV)
     vertices = [startPose(1:2); goalPose(1:2)];
     for i = 1:size(obstacles,1)
         x = obstacles(i,1); y = obstacles(i,2); w = obstacles(i,3); h =
             obstacles(i,4);
         vertices = [vertices;
             x y;
             x+w y;
             x y+h;
             x+w y+h];
     end
```

```
    vertices = unique(vertices, 'rows', 'stable');
    N = size(vertices,1);
    adjMat = zeros(N); costMat = inf(N);

    for i = 1:N
        for j = i+1:N
            if ~collides(vertices(i,:), vertices(j,:), obstacles)
                dist = norm(vertices(i,:) - vertices(j,:));
                time = dist / maxV;
                adjMat(i,j) = 1; adjMat(j,i) = 1;
                costMat(i,j) = time; costMat(j,i) = time;
            end
        end
    end
end

function col = collides(p1, p2, obstacles)
    col = false; N = 10;
    xs = linspace(p1(1), p2(1), N); ys = linspace(p1(2), p2(2), N);
    for i = 1:length(xs)
        for j = 1:size(obstacles,1)
            r = obstacles(j,:);
            if xs(i) > r(1) && xs(i) < r(1)+r(3) && ys(i) > r(2) && ys(i)
               < r(2)+r(4)
                col = true; return;
            end
        end
    end
end

function [path, totalTime] = findPathWithDijkstra(vertices, adj, v_max,
   omega_max, startPose, goalPose)
    N = size(vertices,1);
    dist = inf(1,N); prev = zeros(1,N); orient = zeros(1,N);
    startIdx = find(ismember(vertices, startPose(1:2), 'rows'));
    goalIdx = find(ismember(vertices, goalPose(1:2), 'rows'));
    dist(startIdx) = 0; orient(startIdx) = startPose(3);
    Q = 1:N;

    while ~isempty(Q)
        [~, idx] = min(dist(Q)); u = Q(idx); Q(idx) = [];
```

```
        if u == goalIdx, break; end
        for v = find(adj(u,:))
            p1 = vertices(u,:); p2 = vertices(v,:);
            d = norm(p2 - p1); t_lin = d / v_max;
            theta_des = atan2(p2(2)-p1(2), p2(1)-p1(1));
            dtheta = wrapToPi(theta_des - orient(u));
            t_rot = abs(dtheta) / omega_max;
            cost = t_lin + t_rot;
            if dist(u) + cost < dist(v)
                dist(v) = dist(u) + cost;
                prev(v) = u; orient(v) = theta_des;
            end
        end
    end

    path = goalIdx;
    while path(1) ~= startIdx
        path = [prev(path(1)); path];
    end
    totalTime = dist(goalIdx);
end

function [v, omega_z, omega_r, omega_l, tempo] =
   computeVelocitiesTrapezoidal(vertices, pathIdx, v_max, omega_max, r, d,
    Ts, goalPose)
    v = []; omega_z = []; omega_r = []; omega_l = []; tempo = []; t = 0;

    for i = 1:length(pathIdx)-1
        p1 = vertices(pathIdx(i), :);
        p2 = vertices(pathIdx(i+1), :);
        dx = p2(1)-p1(1); dy = p2(2)-p1(2);
        theta_des = atan2(dy, dx);

        if i==1, theta_prev = 0;
        else
            p0 = vertices(pathIdx(i-1), :);
            theta_prev = atan2(p1(2)-p0(2), p1(1)-p0(1));
        end

        % === ROTAZIONE SUL POSTO ===
        delta = wrapToPi(theta_des - theta_prev);
```

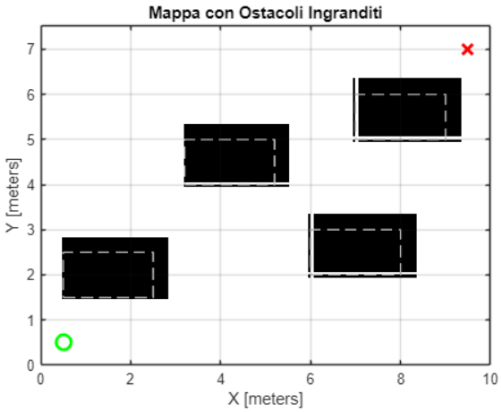
```
t_rot = abs(delta)/omega_max;
Trot = 0:Ts:t_rot;
omega = sign(delta)*omega_max * ones(size(Trot));
v_rot = zeros(size(Trot));
w_r = (d/2 * omega) / r;
w_l = -w_r;

v = [v; v_rot'];
omega_z = [omega_z; omega'];
omega_r = [omega_r; w_r'];
omega_l = [omega_l; w_l'];
tempo = [tempo; t + Trot'];
t = t + t_rot;

% === TRASLAZIONE ===
dist = hypot(dx, dy);
a = 0.1;  % accelerazione
t_a = v_max / a;
d_a = 0.5*a*t_a^2;

if dist < 2*d_a
    t_a = sqrt(dist/a);
    t_c = 0; d_c = 0;
else
    d_c = dist - 2*d_a;
    t_c = d_c / v_max;
end
T1 = 0:Ts:t_a;
T2 = Ts:Ts:t_c;
T3 = Ts:Ts:t_a;

v_prof = [a*T1, v_max*ones(1,length(T2)), v_max - a*T3];
omega_seg = zeros(size(v_prof));
w_r2 = v_prof ./ r;
w_l2 = w_r2;

v = [v; v_prof'];
omega_z = [omega_z; omega_seg'];
omega_r = [omega_r; w_r2'];
omega_l = [omega_l; w_l2'];
tempo = [tempo; (t + Ts*(0:length(v_prof)-1))'];
```

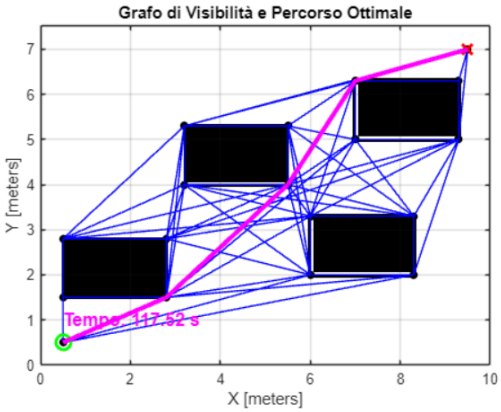
```
        t = tempo(end);
    end

    % === ROTAZIONE FINALE PER RISPETTARE LA POSA ===
    theta_finale = atan2(vertices(pathIdx(end),2)-vertices(pathIdx(end-1)
        ,2), ...
                         vertices(pathIdx(end),1)-vertices(pathIdx(end-1)
                            ,1));
    delta = wrapToPi(goalPose(3) - theta_finale);
    t_rot = abs(delta)/omega_max;
    Trot = 0:Ts:t_rot;

    if t_rot > 0
        omega = sign(delta)*omega_max * ones(size(Trot));
        v_rot = zeros(size(Trot));
        w_r = (d/2 * omega) / r;
        w_l = -w_r;

        v = [v; v_rot'];
        omega_z = [omega_z; omega'];
        omega_r = [omega_r; w_r'];
        omega_l = [omega_l; w_l'];
        tempo = [tempo; t + Trot'];
    end
end

function plotResults(map, obstacles, inflatedObstacles, startPose,
    goalPose, vertices, adjacencyMatrix, pathIndices, totalTime)
    %figure;
    figure('Position', [100, 100, 1200, 600]);
    subplot(1,2,1);
    show(map);
    hold on;
    title('Mappa con Ostacoli Ingranditi');

    for i = 1:size(obstacles, 1)
        rectangle('Position', obstacles(i,:), 'EdgeColor', [0.8 0.8 0.8],
            'LineWidth', 1, 'LineStyle', '--');
    end

    plot(startPose(1), startPose(2), 'go', 'MarkerSize', 10, 'LineWidth',
```

```
    2, 'DisplayName', 'Start');
plot(goalPose(1), goalPose(2), 'rx', 'MarkerSize', 10, 'LineWidth', 2,
    'DisplayName', 'Goal');

% legend({'OccupancyGrid', 'Ostacoli Originali', 'Start', 'Goal'});
grid on;
axis equal;
xlim([0 map.XWorldLimits(2)]);
ylim([0 map.YWorldLimits(2)]);

subplot(1,2,2);
show(map);
hold on;
title('Grafo di Visibilit  e Percorso Ottimale');

plot(vertices(:,1), vertices(:,2), 'ko', 'MarkerSize', 5, '
    MarkerFaceColor', 'k', 'DisplayName', 'Nodi Grafo');

for i = 1:size(adjacencyMatrix, 1)
    for j = i+1:size(adjacencyMatrix, 2)
        if adjacencyMatrix(i, j) == 1
            plot([vertices(i,1), vertices(j,1)], [vertices(i,2),
                vertices(j,2)], 'b-', 'LineWidth', 0.5, 'DisplayName',
                'Archi Visibili');
        end
    end
end

plot(startPose(1), startPose(2), 'go', 'MarkerSize', 10, 'LineWidth',
    2, 'DisplayName', 'Start');
plot(goalPose(1), goalPose(2), 'rx', 'MarkerSize', 10, 'LineWidth', 2,
    'DisplayName', 'Goal');

if ~isempty(pathIndices)
    pathCoords = vertices(pathIndices, :);
    plot(pathCoords(:,1), pathCoords(:,2), 'm-', 'LineWidth', 3, '
        DisplayName', 'Percorso Ottimale');
    text(pathCoords(1,1), pathCoords(1,2) + 0.5, ['Tempo: ', num2str(
        totalTime, '%.2f'), ' s'], ...
          'Color', 'm', 'FontSize', 12, 'FontWeight', 'bold');
end
```

```
    % legend show;
    grid on;
    axis equal;
    xlim([0 map.XWorldLimits(2)]);
    ylim([0 map.YWorldLimits(2)]);

end
```

Per rendere il tutto più chiaro ho deciso di affiancare i grafici degli ostacoli prima senza grafo, poi con grafo e anche delimitazione del migliore percorso con il tempo.



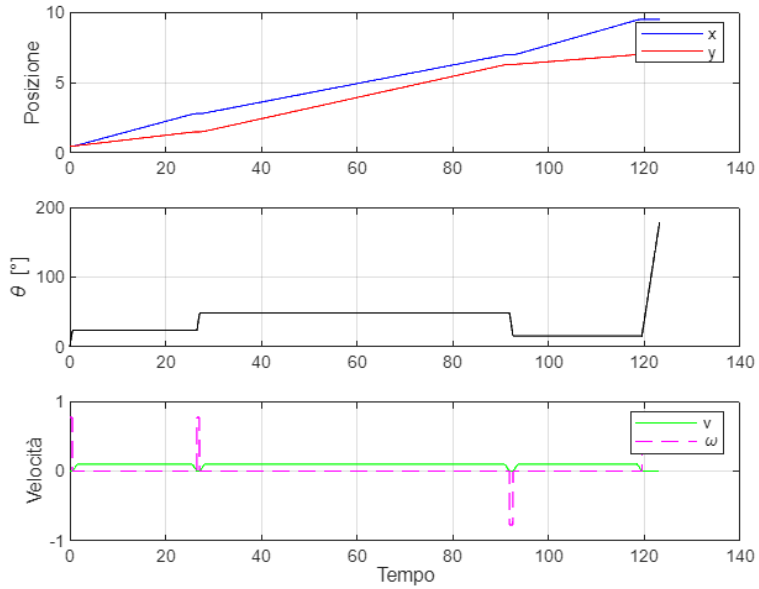


A questo punto plottiamo la velocità, facendo seguire un profilo di velocità trapezoidale :

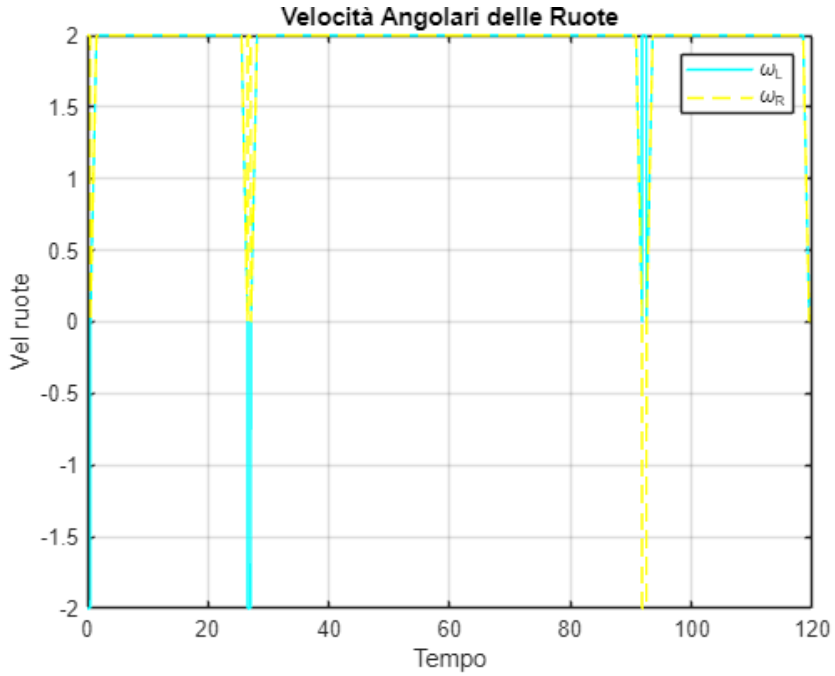
```
function plotCinematica(tempo, stati, velocita, omegaL, omegaR)
    figure;
    subplot(3,1,1);
    plot(tempo, stati(:,1), 'b', tempo, stati(:,2), 'r');
    legend('x','y'); ylabel('Posizione'); grid on;

    subplot(3,1,2);
    plot(tempo, rad2deg(stati(:,3)), 'k'); ylabel('\theta [ ]'); grid on;

    subplot(3,1,3);
    plot(tempo, velocita(:,1), 'g', tempo, velocita(:,2), 'm--');
    legend('v','\omega'); ylabel('Velocit '); grid on; xlabel('Tempo');

    figure;
    plot(tempo, omegaL, 'c', tempo, omegaR, 'y--');
    legend('\omega_L','\omega_R'); ylabel('Vel ruote'); grid on; xlabel('
       Tempo');
```

```
    title('Velocit  Angolari delle Ruote');
end
```

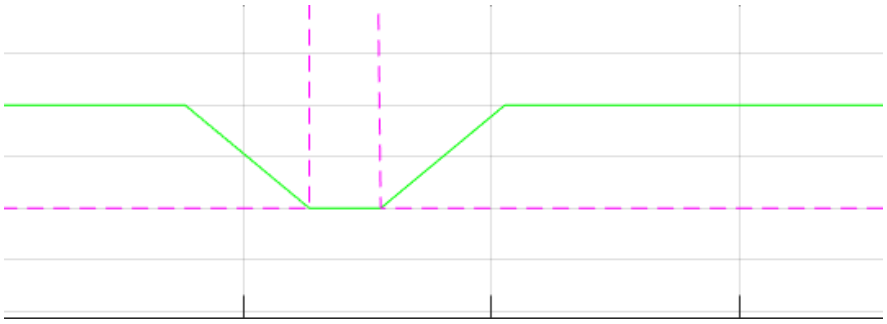
Adesso, visioniamo prima le varie info come orientamento e velocità, tenendo conto del profilo trapezoidale della velocità :



Ora visioniamo quelle che sono le velocità angolari delle singole ruote :



Come possiamo notare le velocità angolari delle ruote sono opposte, a riconferma di quello che era stato specificato in precedenza. Inoltre non si nota molto il profilo *trapezoidale* delle velocità dunque vado a zoommare per far capire meglio quello che accade.



In questo caso siamo in procinto di una rotazione, quindi la velocità lineare arriva a 0 e quando la rotazione finisce questa ricomincia al massimo.

3.2 Campi potenziali

Questa tecnica consiste nell'andare a stabilire un campo attrattivo globale posizionato sul punto da raggiungere, e poi dei campi repulsivi attorno agli ostacoli Questo permetterà al robot di essere *"spinto"* verso l'obiettivo, andando ad *evitare* gli ostacoli. Il *gradiente negativo* del potenziale totale viene trattato come una forza applicata al robot. Dunque avremo :

$$U(q) = U_{goal}(q) + \sum U_{ostacoli}(q)$$

Questo è il gradiente negativo del potenziale :

$$F(q) = \nabla U(q)$$

Il codice che è stato implementato su matlab segue proprio questo schema e questi vincoli, in particolare ho preferito in questo caso fare una sorta di animazione per far capire bene il percorso che il robot segue e come viene *deviato* dagli ostacoli. Inoltre per sperimentare varie combinazioni nelle diverse simulazioni, ho apportato cambiamenti in quelli che sono i coefficienti di *repulsione* e *attrazione*, osservando come più si aumenta la repulsione, più si rischia di cadere in quelli che sono dei ***minimi locali***, soprattutto in corrispondenza di ostacoli troppo vicini. Questo è un grande problema del metodo appena descritto, ma lo analizzo meglio nel confronto.

Visioniamo il codice :

```
clc;
clear;
close all;

% Parametri
k_att = 1;
k_rep = 2.5;
rho_0 = 1.5;
```

```
v_max = 0.1;
omega_max = pi/4;
dt = 0.1;
threshold = 0.1;

% Posa iniziale e goal
x = 0.5; y = 0.5; theta = 0;
goal = [9.5, 7];
%ostacoli
obstacles = [
    0.5, 1.5, 2.30, 1.30;
    6, 2, 2.30, 1.30;
    2, 5, 2.30, 1.30;
    7, 5, 2.30, 1.30
];

% Inizializza percorso
path_x = x;
path_y = y;

% Griglia per visualizzare campo
[x_grid, y_grid] = meshgrid(0:0.2:10, 0:0.2:7.5);
U = zeros(size(x_grid));
V = zeros(size(y_grid));

% Calcola campo potenziale
for i = 1:numel(x_grid)
    p = [x_grid(i); y_grid(i)];
    F_att = k_att * (goal' - p);

    F_rep = [0; 0];
    for j = 1:size(obstacles,1)
        [d, grad_d] = distanza_rettangolo(p, obstacles(j,:));
        if d < rho_0 && d > 0
            F_rep = F_rep + k_rep * (1/d - 1/rho_0) * (1/d^2) * grad_d;
        end
    end

    F = F_att + F_rep;
    U(i) = F(1);
    V(i) = F(2);
```

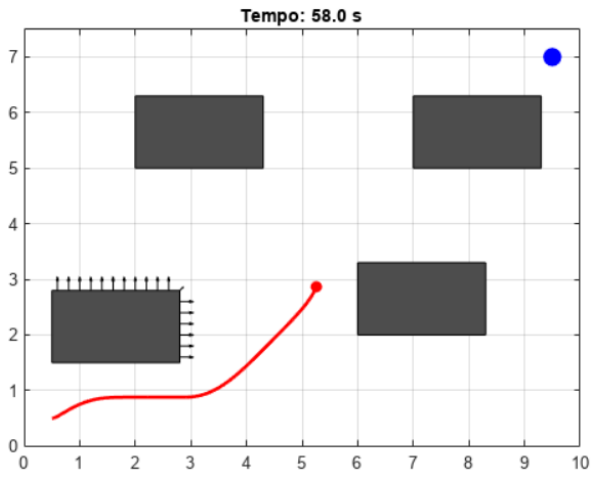
```
end

% Simulazione
t = 0;
v_log = [];
omega_log = [];
t_log = [];
while norm([x; y] - goal') > threshold && t < 200
    p = [x; y];
    F_att = k_att * (goal' - p);

    F_rep = [0; 0];
    for j = 1:size(obstacles,1)
        [d, grad_d] = distanza_rettangolo(p, obstacles(j,:));
        if d < rho_0 && d > 0
            F_rep = F_rep + k_rep * (1/d - 1/rho_0) * (1/d^2) * grad_d;
        end
    end

    F = F_att + F_rep;
    theta_desired = atan2(F(2), F(1));
    error_theta = wrapToPi(theta_desired - theta);

    omega = max(-omega_max, min(omega_max, 1.5 * error_theta));
    v = v_max * exp(-abs(error_theta)); % rallenta se troppo disallineato

    v_log(end+1) = v;
    omega_log(end+1) = omega;
    t_log(end+1) = t;

    % Aggiorna posa
    x = x + v * cos(theta) * dt;
    y = y + v * sin(theta) * dt;
    theta = wrapToPi(theta + omega * dt);

    % Salva percorso
    path_x(end+1) = x;
    path_y(end+1) = y;
    t = t + dt;

    % Visualizza
```

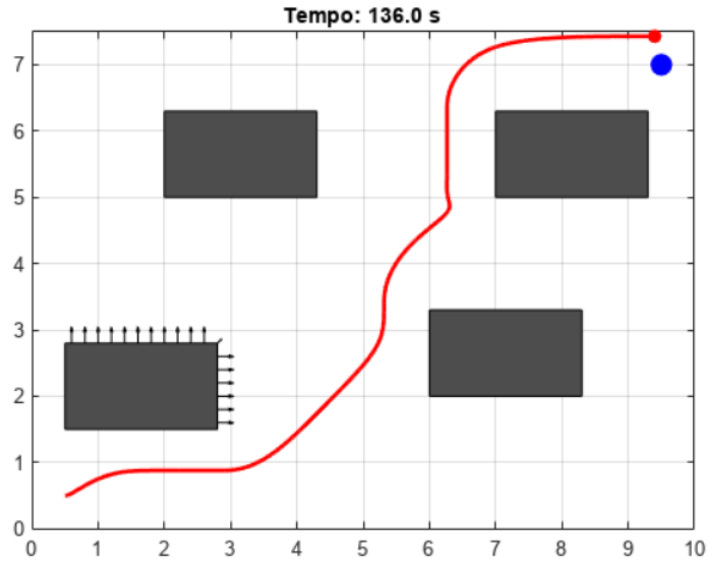
```
    if mod(round(t/dt), 10) == 0
        figure(1); clf;
        quiver(x_grid, y_grid, U, V, 'k'); hold on;
        for k = 1:size(obstacles,1)
            rectangle('Position', obstacles(k,:), 'FaceColor', [0.3 0.3
                0.3 0.5], 'EdgeColor', 'k');
        end
        plot(path_x, path_y, 'r-', 'LineWidth', 2);
        plot(x, y, 'ro', 'MarkerFaceColor', 'r');
        plot(goal(1), goal(2), 'bo', 'MarkerSize', 10, 'MarkerFaceColor',
            'b');
        title(sprintf('Tempo: %.1f s', t));
        axis equal; grid on;
        xlim([0 10]); ylim([0 7.5]);
        drawnow;
    end
end

v_log_smooth = movmean(v_log, 10);
omega_log_smooth = movmean(omega_log, 10);

% Plot finale
figure;

subplot(2,1,1);
plot(t_log, v_log, 'b', t_log, v_log_smooth, '--k');
title('Andamento delle velocit ');
ylabel('Velocit  lineare v (m/s)');
legend('v', 'v filtrata');

subplot(2,1,2);
plot(t_log, omega_log, 'r', t_log, omega_log_smooth, '--k');
xlabel('Tempo (s)');
ylabel('Velocit  angolare \omega (rad/s)');
legend('\omega', '\omega filtrata');

% Funzione distanza e gradiente continuo
function [d, grad_d] = distanza_rettangolo(p, rect)
    x = p(1); y = p(2);
    rx = rect(1); ry = rect(2); w = rect(3); h = rect(4);
```

```
        dx = max([rx - x, 0, x - (rx + w)]);
        dy = max([ry - y, 0, y - (ry + h)]);
        d = hypot(dx, dy);

        if d == 0
            grad_d = [0; 0];
        else
            px = min(max(x, rx), rx + w);
            py = min(max(y, ry), ry + h);
            grad_d = [(x - px); (y - py)] / (d + eps);
        end
    end
```

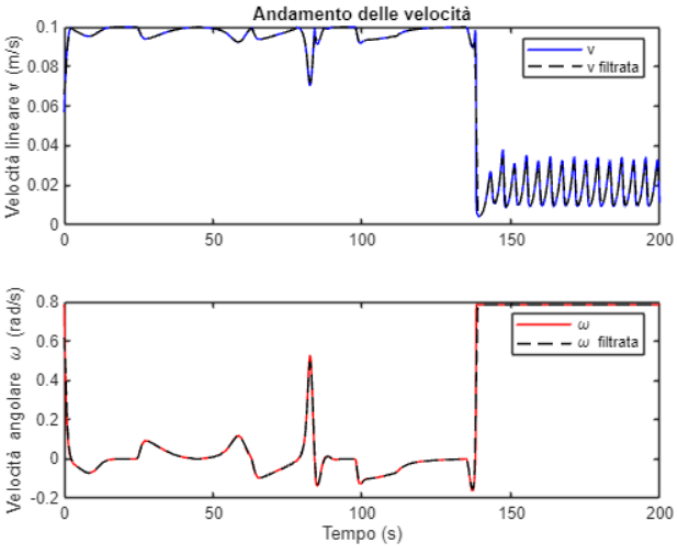
Ci sono anche diverse funzioni che permettono di plottare le velocità e in particolar modo di andare a filtrare quest'ultime, perchè l'influenza che hanno gli ostacoli potrebbe portare a cambiamenti troppo bruschi di velocità soprattutto in quella angolare.

Ecco il risultato :





Vediamo la velocità :



Sicuramente una cosa che si poteva migliorare era il criterio di arresto, difatti nonostante il robot arriva al suo obiettivo continua a girarci attorno, tentando di avvicinarsi, questo lo si poteva fare con un *if* nel loop.

NB. nello script finale ho modificato leggermente la forza attrattiva, e alcune caratteristiche del grafico finale, a solo scopo visivo

4 Conclusione

Abbiamo osservato due metodi che vengono applicati allo stesso fine, in quanto per entrambi bisogna avere una conoscenza dell'ambiente totale, bisogna sapere dunque dove sono inseriti gli ostacoli, e dunque anche dove si trova l'obiettivo, ma tutti e due presentano pro e contro. Il primo metodo, ovvero quello dei ***grafi***

di visibilità fa parte degli approcci roadmap, nei quali vi è anche il metodo **voronoi**. Il pro principale è che è ottimale rispetto alla distanza(per distanza intendo il metodo di minimizzazione del percorso rispetto al quale si è scelto di operare, nel nostro caso il tempo), è abbastanza preciso(anche questo però dipende dalle caratteristiche intrinseche del robot, come per esempio i sensori), e poi è semplice da implementare, in quanto esistono parecchi algoritmi che lavorano sui grafi. Un contro però potrebbe essere la complessità computazionale, parliamo infatti di algoritmi che hanno una complessità pari a $O(n^2)$ dove per n intendiamo il numero di vertici, e quindi di nodi che andiamo ad attraversare, e che quindi ci suggerisce che in presenza di una moltitudine di ostacoli è parecchio complesso.

Per quanto riguarda il secondo metodo, invece, i pro sono diversi : il planning e il controllo sono uniti in un'unica funzione, poi è anche molto semplice da implementare e computazionalmente "leggero", il problema principale però, come già in precedenza anticipato è che si rischia di ricadere in dei *minimi locali*, che non permettono al robot di raggiungere l'obiettivo, come se questo si trovasse bloccato, per uscirne spesso si usano algoritmi di backtracking.