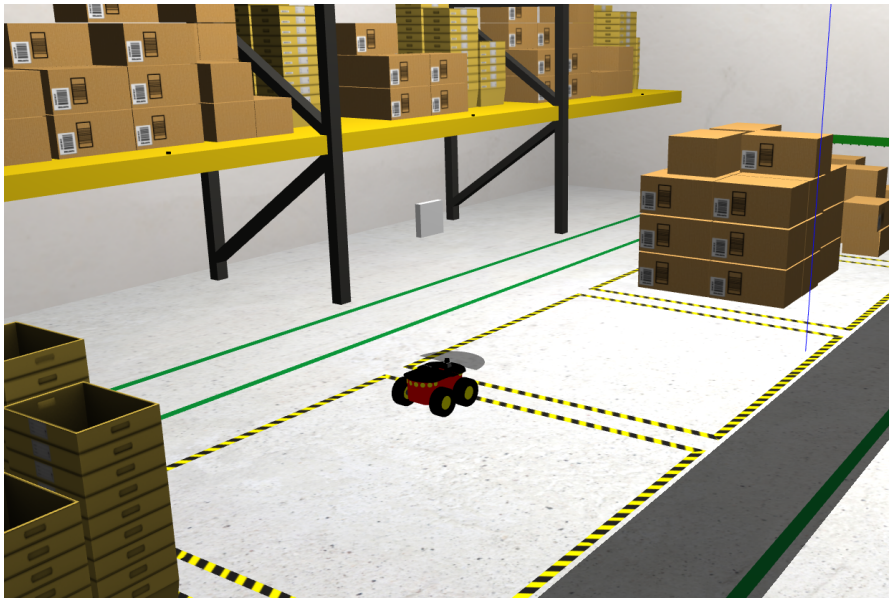
Scuola Superiore Sant'Anna

Scienze Sperimentali - Ingegneria

Robotics Programming

Simone Cirelli, Alessandro Meini, Cristian Perissutti, Leonardo Vico



A.A. 2022-2023

Contents

1 Introduzione

L'obiettivo del progetto è sviluppare un sistema di controllo per un robot skid-steer facendo in modo che essa sia in grado di muoversi in maniera autonoma all'interno di un ambiente evitando gli ostacoli fino a una posizione target.
Il movimento del robot seguirà una traiettoria ottimale, valutata attraverso un algoritmo di Q-learning, mentre il robot è controllato in velocità utilizzando ROS. Successivamente è stato implementato un algoritmo di trilatersazione per localizzare il robot, utilizzando il segnale di 3 tag RIFD.
Per il progetto è stato utilizzato un robot Pioneer3AT che si muove all'interno di un magazzino di Amazon [1].
Un video del codice in esecuzione può essere trovato al seguente link Youtube.

2 ROS Environment

Il codice è stato implementato all'interno di un workspace di ROS. All'interno del workspace sono presenti i seguenti pacchetti:

- **aws-robomaker-small-warehouse-world**: questo pacchetto contiene la mappa del mondo all'interno del quale il robot si muove. La mappa consiste in un magazzino di Amazon, ed è stata scaricata dal GitHub ufficiale di Amazon Web Service - Robotics.
- **move_robot**: questo pacchetto contiene il file "follow_waypoints.py", che implementa il nodo che si occupa della navigazione
- **rfid_sensor**: questo pacchetto implementa il nodo che processa i dati ottenuti dagli rfid al fine di ottenere la posizione del robot
- **rfid_sensor_test**: questo pacchetto contiene il file .launch che lancia la simulazione con Gazebo, carica il robot e attiva i sensori RFID.

Per lanciare il codice, occorre aprire delle finestre di terminale all'interno della cartella "rob_prog" ed eseguire i seguenti comandi:

```
// terminale n.1 -> nodo Master di ROS
source devel/setup.bash
roscore

// terminale n.2 -> lanciare Gazebo e caricare il mondo
source devel/setup.bash
roslaunch rfid_simulator_test simulation.launch

// terminale n.3 -> lanciare la navigazione
source devel/setup.bash
rosrun move_robot follow_waypoints.py

// terminale n.4 -> verifica precisione degli RFID
source devel/setup.bash
rosrun rfid_sensor rfid.py
```

3 Pianificazione della Traiettoria

3.1 Q-learning algorithm

Per la pianificazione della traiettoria è stato utilizzato un algoritmo di Q-learning.

Il Q-Learning è un algoritmo di reinforcement learning che consente di ricavare una politica ottimale per il raggiungimento di un obiettivo. per fare ciò viene massimizzata la ricompensa cumulativa ottenuta esplorando le azioni possibili in diversi stati dello spazio. L'elemento principale dell'algoritmo di Q-learning è la q_table: una tabella con un numero di righe pari al numero di stati e un numero di colonne pari al numero di azioni possibili. In ogni cella (x,y) della q_table è presente il punteggio associato all'applicazione dell'azione y partendo dalla cella x. L'equazione fondamentale del Q-Learning è l'aggiornamento dei valori della Q_table:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \tag{1}$$

Dove:

- s_t è lo stato corrente al tempo t.
- a_t è l'azione eseguita nello stato s_t.
- r_t è la ricompensa ottenuta dopo aver eseguito a_t in s_t.
- α è il tasso di apprendimento (learning rate).
- γ è il fattore di sconto (discount factor).
- $\max_a Q(s_{t+1}, a)$ è la stima del massimo valore Q per lo stato successivo s_{t+1}.

Applicando ripetutamente questa equazione, l'algoritmo converge ad una politica ottimale che massimizza la ricompensa attesa a lungo termine.

Applicheremo questo algoritmo per l'ottimizzazione del percorso in una griglia 2D verso una posizione target. In questo caso, gli stati sono rappresentati dalle celle della griglia, mentre le azioni possibili sono i movimenti nelle quattro direzioni (su, giù, sinistra, destra). L'obiettivo è di individuare il percorso ottimale dalla posizione iniziale al target, evitando ostacoli e minimizzando il numero di mosse.

Il metodo di calcolo e ottimizzazione è definito *Epsilon-greedy*. Iniziamo inizializzando la tabella $Q(s, a)$ a 0 per ogni coppia stato-azione (s, a). Poi per ogni iterazione di apprendimento:

- Partiamo dallo stato iniziale s_{t0}
- Selezionare un'azione a_t utilizzando la politica ϵ-greedy:
  - Con probabilità ϵ, selezionare un'azione casuale.
  - Con probabilità $1 - \epsilon$, selezionare l'azione a che massimizza $Q(s_t, a)$.
- Eseguire l'azione a_t, osservare la ricompensa r_t e lo stato successivo s_{t+1}.
- Aggiornare il valore di $Q(s_t, a_t)$ nella tabella utilizzando l'equazione 1
- Aggiornare $s_t \leftarrow s_{t+1}$.

- Ripetere fino a quando lo stato target non viene raggiunto, viene superato un numero massimo di tentativi definito dall'utente o lo stato corrisponde a un ostacolo o all'uscita dalla mappa.

Questo algoritmo viene ripetuto per un numero elevato di iterazioni. All'inizio, ϵ è alto per favorire l'esplorazione, poi viene gradualmente diminuito per sfruttare le conoscenze acquisite. La politica ϵ-greedy permette di bilanciare l'esplorazione (scegliere azioni casuali) e lo sfruttamento (scegliere l'azione che massimizza Q) durante l'apprendimento.

Dopo un numero sufficiente di iterazioni, la tabella Q convergerà verso i valori ottimali per le coppie stato-azione, consentendo all'agente di seguire la politica ottimale selezionando semplicemente l'azione che massimizza $Q(s, a)$ per ogni stato s. Una volta creata la tabella essa puo essere utilizzata per trovare la sequenza di scelte ottimali per arrivare al target da ciascuno stato di partenza.

3.2 Applicazione dell'algoritmo

L'ambiente è stato discretizzato: la mappa del magazzino è stata divisa in celle di dimensione 45 cm x 45 cm. Tali celle sono state organizzate sotto forma di una matrice, all'interno del file "map.py" scritto in linguaggio Python. Il valore di ciascuna cella è "1" se è presente un'ostacolo e "0" se invece la cella è libera.

Nel file "Q-learning.py" è definito l'algoritmo di Q-learning.

Sono state utilizzate le librerie di python numpy, random, math, pandas e matplotlib.pyplot. Sono state definite le dimensioni della matrice che rappresenta la mappa, è stata importata la mappa dal file map.py e sono stati definiti il vettore delle possibili azioni e la posizione di arrivo

```
# parametri tabella
dimrow = 30  # numero colonne (dimensione di una riga)
dimcol = 46  # numero righe (dimensioni di una colonna)

env = map.matrice

# definizione azioni
actions
# target position
tpos = (37,24) #(nell'ordine coordinata y  e x)

# Definizone tabella Q
# Ordine delle celle prima riga 0 colonna 0,1,2,3,4... ,
#riga 1 colonna 0,1,2,3,4 riga 2 ecc...
q_table = np.zeros((dimrow * dimcol, len(actions)))
```

In seguito sono stati definiti i parametri necessari all'apprendimento

```
# learning parameters
num_iter = 1000000
learning_rate = 0.01
discount_rate = 0.5
exp_rate = 1.0
```

```
max_iter_episode = 80
```

Sono state poi definite due funzioni: next_step ha come input la posizione attuale e la mossa e restituisce la posizione finale indicando con il flag uscita se l'azione ha fatto uscire la macchinina dai limiti della mappa. Get_Initial_state sceglie una casella libera casuale all'interno della mappa.

```
    # calcolo posizione successiva
def next_step(cur_pos, action):
    col = cur_pos[0]
    row = cur_pos[1]
    uscita = 0 # se esce dall'ambiente metto uscita = 1
    if action == 0 and row > 0:
        row -= 1
    elif action == 1 and col < (dimcol-1):
        col += 1
    elif action == 2 and row < (dimrow-1):
        row += 1
    elif action == 3 and col > 0:
        col -= 1
    else:
        uscita = 1
    return col, row, uscita


def get_initial_state():
    start_x = np.random.randint(dimrow)
    start_y = np.random.randint(dimcol)
    while env[start_y,start_x] != 0:
        start_x = np.random.randint(dimrow)
        start_y = np.random.randint(dimcol)
    return start_y, start_x
```

All'interno del ciclo for avviene la fase di addestramento dell'algoritmo: Viene scelta una posizione iniziale tramite la funzione definita sopra e vengono definite due variabili "finish" e "num_steps" che indicano rispettivamente il termine del percorso (sia per arrivo a destinazione che per incontro di ostacolo o uscita dalla mappa) e il numero di azioni svolte durante il percorso.

```
    for episode in range(num_iter):
    # Inizializzazione dello stato di partenza
    a_pos = get_initial_state()
    finish = False
    num_steps = 0
```

In seguito è presente un altro ciclo che termina quando finish = 1 in cui sono definiti i movimenti che formano ciascun percorso. Il numero di passi viene aumentato a ogni ciclo e viene scelta l'azione nel modo definito precedentemente. Una volta scelta la mossa, viene calcolata la posizione successiva.

```
    while not finish:
    num_steps += 1
```

```
    # Scelta dell'azione
    if random.uniform(0, 1) < exp_rate:
        action = random.choice(list(actions))
    else:
        action_index = np.argmax(q_table[a_pos[0] * dimrow + a_pos[1], :])
        action = actions[action_index]

    #calcolo nuova posizione
    next_state = next_step(a_pos, int(action))
```

Viene poi eseguito un controllo sulla nuova posizione:

- Se la nuova posizione è fuori dalla mappa o se è la stessa posizione di un ostacolo, viene assegnato un reward molto negativo;
- Se la nuova posizione è il target viene dato un reward molto positivo
- In tutti gli altri casi viene dato un reward leggermente negativo in modo da penalizzare i percorsi più lunghi.

```
if next_state[2] == 1: # sono uscito dall'ambiente
    reward = -50
elif env[next_state[0],next_state[1]] == 1: # ostacolo
    reward = -50
elif next_state[0] == tpos[0] and next_state[1] == tpos[1]:
    reward = 100
else:
    reward = -1
```

Successivamente viene aggiornato il punteggio della q-table relativo alla casella di partenza e alla mossa appena eseguita utilizzando la formula 1. Nel caso in cui la casella di arrivo sia fuori dalla mappa, non essendoci un punteggio relativo alla casella di arrivo, viene dato alla variabile Q(s_t+1, a) un valore molto negativo.

```
    if next_state[2] == 1:
    q_table[a_pos[0] * dimrow + a_pos[1], action] =
    ((1 - learning_rate) * q_table[a_pos[0] * dimrow + a_pos[1], action]) ++
    learning_rate * (reward + discount_rate * (-100))
    finish = True
else:
    q_table[a_pos[0] * dimrow + a_pos[1], action] =
    ((1 - learning_rate) * q_table[a_pos[0] * dimrow + a_pos[1], action]) +
    + learning_rate * (reward + discount_rate * np.max(q_table[next_state[0] *
    * dimrow + next_state[1], :]))
    a_pos = next_state
```

Viene poi valutato se il percorso è terminato o se il numero di azioni eseguite è troppo grande. In caso affermativo viene posto a True il flag "finish". Come accennato precedentemente viene anche diminuita in maniera esponenziale la variabile exp_rate.

```
      if (a_pos[0] == tpos[0] and a_pos[1] == tpos[1])
      or (env[a_pos[0],a_pos[1]] == 1):
        finish = True

    # controllo se ho messo troppe mosse
    if num_steps >= max_iter_episode:
        finish = True

    reward_iter += reward

# Riduzione del tasso di esplorazione
exp_rate = np.exp(-10 * episode/num_iter)
```

Infine, viene costruita una copia della mappa in cui in ogni cella è indicata la presenza di un ostacolo, l'arrivo e l'azione più conveniente (indicata mediante una freccia). Tale mappa viene esportata come immagine

```
    final_map = np.empty((dimcol, dimrow), dtype=np.str_)

for k in range(dimrow * dimcol):
    act = np.argmax(q_table[k,:])
    if env[math.trunc(k/dimrow),k - (dimrow * math.trunc(k/dimrow)) ] == 1:
        final_map[math.trunc(k / dimrow), k - (dimrow * math.trunc(k / dimrow))] = ''
    elif (math.trunc(k/dimrow) == tpos[0] and
    k - (dimrow * math.trunc(k / dimrow)) == tpos[1]):
        final_map[math.trunc(k / dimrow), k - (dimrow * math.trunc(k / dimrow))] = ''
    elif act == 0:
        final_map[math.trunc(k/dimrow),k - (dimrow * math.trunc(k/dimrow)) ] =
        ='\N{LEFTWARDS ARROW}'
    elif act == 1:
        final_map[math.trunc(k/dimrow), k - (dimrow * math.trunc(k/dimrow))] =
        ='\N{DOWNWARDS ARROW}'
    elif act == 2:
        final_map[math.trunc(k/dimrow), k - (dimrow * math.trunc(k/dimrow))] =
        ='\N{RIGHTWARDS ARROW}'
    else:
        final_map[math.trunc(k/dimrow), k - (dimrow * math.trunc(k/dimrow))] =
        ='\N{UPWARDS ARROW}'

#df = pd.DataFrame(final_map)
#df.to_excel('final_map.xlsx', index=False)
```

```
# Creazione del grafico della tabella utilizzando matplotlib
fig, ax = plt.subplots(figsize=(6, 8))
ax.axis('off')
table = ax.table(cellText=final_map, loc='center')

# Salvataggio del grafico della tabella come immagine
plt.savefig('mappa_final.png')
```

Scegliendo come punto di arrivo il punto (37,24), il risultato dopo 1 milione di iterazioni è il seguente:

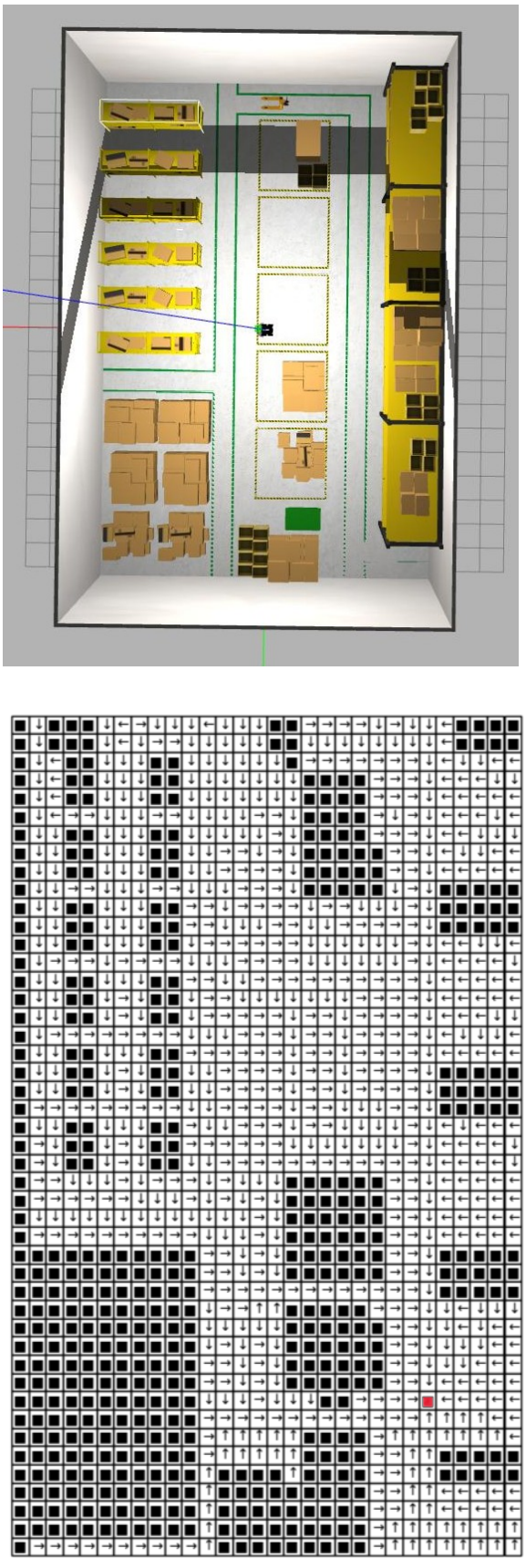


Figure 1: Top view of the room and results of the Q-learning algorithm

4 Controllo del mezzo

Il mezzo è controllato all'interno della mappa utilizzando controlli in velocità mandati sul topic *cmd_vel*. Per ridurre gli errori, il controllo sfrutta una catena chiusa che agisce su posizione e orientazione della macchinina.
Il nodo di ROS che controlla il robot è organizzato nel modo seguente:

4.1 Costanti e funzioni di supporto

All'inizio del codice vengono definite una serie di costanti globali, tra le queli le specifiche della griglia (dimensione delle celle; numero di righe e di colonne) e alcune specifiche per la navigazione (costanti dei controllori, obiettivo).
Vengono inoltre definite due funzioni che permettono di convertire coordinate globali in indici di cella, e viceversa.

```
# COSTANTS
PI = 3.14159265
K_DIST = 0.5
K_ANGLE = 0.2
CELL_LENGTH = 0.435
NUM_COLS = 30
BORDER_X_MIN = 6.50         # x-coordinate of the left border
BORDER_X_MAX = -6.50        # x-coordinate of the right border
BORDER_Y_MIN = -9.82        # y-coordinate of the left border
BORDER_Y_MAX = 9.82         # y-coordinate of the right border
TARGET = (37,24)            # cell_y, cell_x
Q_TABLE_PATH = "Q_learning_def/Q_map_target_37_24.csv"
ANGLE_ERR_MAX = 0.1

#------------------------------------------------------------
# FROM_POS_TO_CELL: go from world coordinates to grid cell
#------------------------------------------------------------
def from_pos_to_cell(pos_x, pos_y):
    x = int((BORDER_X_MIN - pos_x) / CELL_LENGTH)
    y = int((pos_y - BORDER_Y_MIN) / CELL_LENGTH)
    return x,y

#------------------------------------------------------------
# FROM_CELL_TO_POS: fo from grid cell to world coordinates
#------------------------------------------------------------
def from_cell_to_pos(x,y):
    pos_x = BORDER_X_MIN - CELL_LENGTH*x - CELL_LENGTH/2
    pos_y = BORDER_Y_MIN + CELL_LENGTH*y + CELL_LENGTH/2
    return pos_x, pos_y
```

4.2 Class "waypoint_list"

Questa classe implementa una coda FIFO per la memorizzazione dei waypoint che la macchinina deve seguire per raggiungere la destinazione. Essendo la traiettoria fornita dall'algoritmo Q-Learning una successione di celle all'interno della griglia, i waypoint corrispondono alle coordinate del centro delle singole celle.

La classe ha come variabili membro due array: "arr" e "arr_cell", contenenti rispettivamente le coordinate assolute dei centri delle celle, e gli indici di tali celle.

```
#-------------------------------------------------------------
# WAYPOINT_LIST: fifo queue for storing the waypoints
#-------------------------------------------------------------
class waypoint_list:
    def __init__(self):
        self.arr = []
        self.arr_cell = []
        # rospy.loginfo("STACK POPULATED")
        # rospy.loginfo(self.stack)
```

Vengono poi implementate le seguenti funzioni membro:

- **size()**: ritorna la lunghezza della coda
- **push() e push_cell()**: permettono di aggiungere una nuova cella in fondo alla coda
- **pop()**: restituisce il primo elemento della coda (ossia, quello inserito da più tempo) e lo elimina da essa.
- **peek() e peek_cell()**: permettono di ispezionare il primo elemento della coda, senza rimuoverlo
- **is_empty()**: ritorna 1 se la coda è vuota, altrimenti 0
- **print() e print_cell()**: stampano l'intero contenuto della coda, rispettivamente sotto forma di coordinate globali e indici di cella

```
    def size(self):
        return len(self.arr)

    def push(self, waypoint):
        self.arr.append(waypoint)

    def push_cell(self, waypoint):
        self.arr_cell.append(waypoint)

    def pop(self):
        if self.size() > 0:
```

```
            self.arr_cell.pop(0)
            return self.arr.pop(0)
        else:
            raise IndexError("Waypoint Array is empty")


    def peek(self):  # Returns the topmost waypoint from the stack without removing it.
        if self.size() > 0:
            return self.arr[0]
        else:
            raise IndexError("Waypoint Array is empty")

    def peek_cell(self):  # Returns the topmost waypoint from the stack without removing it.
        if self.size() > 0:
            return self.arr_cell[0]
        else:
            raise IndexError("Waypoint Array is empty")

    def is_empty(self):
        return len(self.arr) == 0

    def print(self):
        rospy.loginfo(f"Waypoints (world coordinates) = {self.arr}\n")

    def print_cell(self):
        rospy.loginfo(f"Waypoints (cell coordinates) = {self.arr_cell}\n")
```

4.3 Classe "waypoint_navigator"

Questa classe si occupa di tutte le questioni relative alla navigazione.

4.3.1 Inizializzazione

Nella fase di inizializzazione, viene inizializzato il nodo ROS con il nome "waypoint_navigator". Successivamente vengono inizializzate le variabili membro: un oggetto della classe "waypoint_list"; le variabili di posizione e orientamento; una variabile booleana utilizzata per non far muovere il robot finché non ha ricevuto almeno il primo messaggio contenente la sua posizione; il subscriber al topic che fornisce la posizione al robot; il publisher per mandare i comandi in velocità.

```
class waypoint_navigator:
def __init__(self):
    rospy.loginfo("STARTING NODE MOVE_ROBOT")
    rospy.init_node('waypoint_navigator')
```

```
        # Define waypoints
        self.waypoints = waypoint_list()

        # Current position
        self.curr_pos_x = 0
        self.curr_pos_y = 0
        self.curr_ang_x = 0
        self.curr_ang_y = 0
        self.curr_ang_z = 0

        self.first_position_message = True          # variable that makes the robot wait until the fi

        # Subscribe to gazebo "model_states" topic to get current position
        rospy.Subscriber('/gazebo/model_states', ModelStates, self.model_states_callback)

        # Publisher for velocity commands
        self.velocity_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)

        # Rate for the control loop
        self.rate = rospy.Rate(10)
```

4.3.2 Callback della posizione

Il robot ottiene la sua posizione dai messaggi pubblicati da Gazebo sul topic "gazebo/model_states". Ogniqualvolta viene pubblicato un nuovo messaggio, viene eseguita una funzione di callback che estrae dal messaggio la posizione del robot e la salva all'interno delle variabili membro.
Inoltre, la prima volta che viene ricevuto un messaggio di posizione, viene eseguita la funzione "populate_waypoint_list()", che popola la lista di waypoint sfruttando la tabella del Q-Learning (vedere prossima sottosezione).

```
def model_states_callback(self, msg):
        # Get the index of the robot
        try:
            model_index = msg.name.index("pioneer3at_robot")
        except ValueError:
            rospy.logerr("Model not found in model states message")
            return

        # Get the position of the model
        model_position = msg.pose[model_index].position
        #rospy.loginfo("Position of the robot: x={}, y={}, z={}".format(model_position.x, model_posit
        self.curr_pos_x = model_position.x
        self.curr_pos_y = model_position.y

        # Get the orientation of the model and convert it to Euler angles
        model_orientation = msg.pose[model_index].orientation
```

```
        (roll, pitch, yaw) = euler_from_quaternion([model_orientation.x, model_orientation.y, model_o
        self.curr_ang_x = roll
        self.curr_ang_y = pitch
        if yaw > PI:
            yaw = yaw - 2*PI
        self.curr_ang_z = yaw

        if self.first_position_message:
            self.first_position_message = False
            self.populate_waypoint_list()
```

4.3.3 Ottenimento della traiettoria

Per ottenere la traiettoria che porta la macchinina dalla sua posizione al traguardo, viene utilizzata la tabella restituita dall'algoritmo Q-Learning.
La seguente funzione importa la tabella e, passo dopo passo, ottiene le celle da cui deve passare per raggiungere l'obiettivo e le aggiunge alla coda di waypoints.
Per evitare cicli infiniti nel caso in cui l'algoritmo di Q-learning abbia dato un risultato fallace (ossia, che non permette di arrivare all'obiettivo), il ciclo si interrompe dopo un massimo di 100 passi.

```
def populate_waypoint_list(self):
        Q = np.loadtxt(Q_TABLE_PATH, delimiter=',')
        (x,y) = from_pos_to_cell(self.curr_pos_x, self.curr_pos_y)

        count = 0
        while (y,x) != TARGET:
            cell_index = y*NUM_COLS + x
            next_move = np.argmax(Q[cell_index])
            if next_move == 0:
                next_x = x-1
                next_y = y
            elif next_move == 1:
                next_x = x
                next_y = y+1
            elif next_move == 2:
                next_x = x+1
                next_y = y
            elif next_move == 3:
                next_x = x
                next_y = y-1
            x_pos, y_pos = from_cell_to_pos(next_x, next_y)
            self.waypoints.push((x_pos, y_pos))
            self.waypoints.push_cell((next_x, next_y))

            x = next_x
            y = next_y
```

```
            count += 1
            if count > 100:
                rospy.logerr("PATH TO TARGET NOT FOUND!")
                quit()

        self.waypoints.print_cell()
        self.waypoints.print()
```

4.3.4 Navigation

La seguente funzione implementa l'algoritmo di controllo del robot.
Innanzitutto, il robot aspetta fino alla ricezione del primo messaggio indicante la sua posizione. Dopodiché, ad ogni ciclo il robot ottiene la posizione del primo waypoint nella lista. Innanzitutto, un controllo di tipo proporzionale viene utilizzato per fare girare il robot fino a farlo allineare con la direzione del waypoint successivo. Quando tale angolo viene raggiunto (a meno di una tollernza di 0.1 radianti), il controllo dell'angolo viene combinato con il controllo della velocità (sempre di tipo proporzionale) fino al raggiungimento del successivo waypoint.
Quando il robot entra nella cella di destinazione, il waypoint corrente viene tolto dalla coda e il robot comincia a dirigersi verso il waypoint successivo.

```
    def navigate(self):
        while self.first_position_message:
            rospy.loginfo("Waiting for first position message")
            time.sleep(0.3)


        while not rospy.is_shutdown():
            # Calculate direction to next waypoint
            next_waypoint = self.waypoints.peek()
            next_waypoint_cell = self.waypoints.peek_cell()

            rospy.loginfo("\n\nNext waypoint_cell = {},\t\tNext Waypoint = {} \nCurrent Cell = {} \t\
                next_waypoint_cell, next_waypoint, from_pos_to_cell(self.curr_pos_x, self.curr_pos_y)
            dx = next_waypoint[0] - self.curr_pos_x
            dy = next_waypoint[1] - self.curr_pos_y
            linear_velocity, angular_velocity = self.compute_velocity(dx, dy)

            # Publish velocity commands
            cmd_vel = Twist()
            cmd_vel.linear.x = linear_velocity
            cmd_vel.angular.z = angular_velocity
            self.velocity_pub.publish(cmd_vel)

            # Check if waypoint reached
            #if math.sqrt(dx**2 + dy**2) < ACCURACY:
```

```
            if from_pos_to_cell(self.curr_pos_x, self.curr_pos_y) == next_waypoint_cell:
                print("\n\n\nWAYPOINT REACHED!!!\n\n\n")
                self.waypoints.pop()
                if self.waypoints.is_empty():
                    break

            self.rate.sleep()

        rospy.loginfo("TARGET REACHED!")
        cmd_vel = Twist()           # stop the robot
        self.velocity_pub.publish(cmd_vel)

    def compute_velocity(self, dx, dy):
        angle_to_waypoint = math.atan2(dy,dx)

        # Calculate angular velocity
        angle_err = angle_to_waypoint - self.curr_ang_z
        if angle_err > PI:
            angle_err -= 2*PI
        elif angle_err < -PI:
            angle_err += 2*PI
        angular_velocity = K_ANGLE * angle_err
        if abs(angular_velocity) < 0.01:
            angular_velocity = 0

        # Calculate linear velocity
        distance_to_waypoint = math.sqrt(dx**2 + dy**2)
        linear_velocity = K_DIST * min(0.5, distance_to_waypoint)
        if abs(linear_velocity) < 0.01 or abs(angle_err) > ANGLE_ERR_MAX:
            linear_velocity = 0


        print("Parameters: \n\t\tdx={}, \n\t\tdy={}, \n\t\tlinear_velocity = {}, \n\t\tangle_to_waypo
        return linear_velocity, angular_velocity
```

4.4 Main

Il main non fa altro che inizializzare la classe navigator e lanciare la navigazione.

```
if __name__ == '__main__':

    navigator = waypoint_navigator()
    rospy.loginfo("Starting navigation")

    navigator.navigate()
```

5 Calcolo posizione con tag RFID

Per l'ottenimento della posizione del robot, è stato implementato un metodo di localizzazione basato sull'utilizzo di tag RFID.
La processazione dei dati è stata affidata al nodo ROS implementato nello script "rfid_sensor/src/rfid.py". Il codice si iscrive al topic "/gazebo/antenna1_robot/data" sul quale vengono pubblicati i messaggi dei sensori RFID, implementa le funzioni per ottenere la distanza dai singoli RFID a partire dalla potenza del segnale ricevuto, compie una triangolazione per ottenere la posizione del robot e infine fa una media mobile dei risultati per diminuire i disturbi sulla lettura della posizione.
Dopo un'analisi dei risultati, è stato constatato che questo tipo di lettura non fornisce dati sufficientemente precisi da permettere la navigazione.
Di seguito un'analisi più dettagliata del codice:

5.1 Variabili globali e costanti

```
# GLOBAL CONSTANTS
FREQ = 865.7 * 10**6
RFID_POSITIONS = np.array([[2.9, -1.24, 2.05], [2.9, -3.03, 2.05], [-4.85, 1.44, 1.85]])
ANTENNA_Z = 1.5
ROBOT_START_POS_X = 0
ROBOT_START_POS_Y = 0
```

5.2 Classe MovingAverage

Questa classe implementa una struttura dati che restituisce una media mobile su un numero di dati specificato dall'utente.

```
class MovingAverage:
    def __init__(self, window_size):
        self.window_size = window_size
        self.x_values = []
        self.y_values = []

    def add_value(self, new_x, new_y):
        self.x_values.append(new_x)
        self.y_values.append(new_y)
        if len(self.x_values) > self.window_size:
            self.x_values.pop(0)
            self.y_values.pop(0)

    def get_average(self):
        if not self.x_values:
            return None
        x_average = sum(self.x_values) / len(self.x_values)
        y_average = sum(self.y_values) / len(self.y_values)
        return x_average, y_average
```

5.3 Classe RFID_position

Questa classe implementa tutte le funzioni necessarie per l'inizializzazione degli RFID e per la localizzazione utilizzando i segnali ricevuti.
All'interno della mappa sono stati piazzati tre tag RFID tutti alla stessa quota ed è stato analizzato il segnale ricevuto dall'antenna montata sul robot.

5.3.1 Variabili globali

```
    # GLOBAL CONSTANTS
    LIGHT_SPEED     = 299792458
    FREQ            = 865.7 * 10**6
    LAMBDA          = LIGHT_SPEED / FREQ
    COMM_GAIN       = 3 # 250   # communication gain
    PI              = 3.14159265358979323846
    RFID_POSITIONS  = np.array([[2.9, -1.24, 2.05], [2.9, 4, 2.05], [-4.85, 1.44, 2.05]])
    ANTENNA_Z       = 2.05 # 1.5
    ROBOT_START_POS_X = 0
    ROBOT_START_POS_Y = 0
    IMAGE_PATH = "trajectories.png"
    MAX_COUNTER = 25000 # 2500

    # global vectors for storing the positions
    real_positions = []
    estimated_positions = []
```

5.3.2 Costruttore

Il costruttore inizializza le variabili membro, i subscriber ed il publisher.

```
class RFID_position:
    def __init__(self):
        rospy.init_node('rfid_node', anonymous=True)

        self.det_pow = np.zeros(3)                  # Valore di potenza ricevuto
        self.mean_pos = MovingAverage(50)

        self.rob_x = 0
        self.rob_y = 0

        # Subscriber to the topic publishing data received from RFIDs
        rospy.Subscriber('/gazebo/antenna1_robot/data', TagArray, self.sensor_distances_callback)
        # Subscriber to the model states topic
        rospy.Subscriber('/gazebo/model_states', ModelStates, self.model_states_callback)
        # Publisher for publishing the RFID position
        self.rfid_position_pub = rospy.Publisher('rfid_position', Point, queue_size=10)
```

5.3.3 Localizzazione

Le seguenti tre funzioni servono per localizzare la posizione del robot, note le potenze dei segnali ricevuti dai tre tag RFID e le posizioni dei tre tag.

```
# data RSSI, trova la distanza
def dist_calc(self, rssi):
    dist = COMM_GAIN * LAMBDA / (4* PI * 10**(rssi/40))
    return dist

# ricava le reali distanze dai sensori, per scopo di debug
def real_dist_calc(self):
    d1 = sqrt( (self.rob_x - RFID_POSITIONS[0][0])**2 + (self.rob_y - RFID_POSITIONS[0][1])**2)
    d2 = sqrt( (self.rob_x - RFID_POSITIONS[1][0])**2 + (self.rob_y - RFID_POSITIONS[1][1])**2)
    d3 = sqrt( (self.rob_x - RFID_POSITIONS[2][0])**2 + (self.rob_y - RFID_POSITIONS[2][1])**2)
    return d1,d2,d3

#date coordinate dei tre rfid tag e distanze rilevate trova posizione antenna
def triangolazione(self, d):
    x1, y1 = RFID_POSITIONS[0][0:2]
    x2, y2 = RFID_POSITIONS[1][0:2]
    x3, y3 = RFID_POSITIONS[2][0:2]

    # Estraiamo le distanze note
    d1, d2, d3 = d

    A = 2 * x2 - 2 * x1
    B = 2 * y2 - 2 * y1
    C = d1**2 - d2**2 - x1**2 + x2**2 - y1**2 + y2**2
    D = 2 * x3 - 2 * x2
    E = 2 * y3 - 2 * y2
    F = d2**2 - d3**2 - x2**2 + x3**2 - y2**2 + y3**2
    x = (C * E - F * B) / (E * A - B * D)
    y = (C * D - A * F) / (B * D - A * E)
    return x, y

# funzione complessiva, chiama le altre funzioni e da chiamare a ogni ciclo per trovare la posizi
def calc_pos(self):
    d1 = self.dist_calc(self.det_pow[0], self.pow_0_rfid[0])
    d2 = self.dist_calc(self.det_pow[1], self.pow_0_rfid[1])
    d3 = self.dist_calc(self.det_pow[2], self.pow_0_rfid[2])

    # z-correction to bring to 2D
    d1 = sqrt(d1**2 - (RFID_POSITIONS[0][2] - ANTENNA_Z)**2)
    d2 = sqrt(d2**2 - (RFID_POSITIONS[1][2] - ANTENNA_Z)**2)
    d3 = sqrt(d3**2 - (RFID_POSITIONS[2][2] - ANTENNA_Z)**2)
```

```
        return self.triangolazione([d1, d2, d3])
```

5.3.4 Callback per gli RFID

Questa funzione viene chiamata ogniqualvolta viene ricevuto un messaggio sul topic con i messaggi ricevuti dagli RFID. Questa funzione ottiene i valori della potenza ricevuta dai tre tag utilizzati (tag 2,3 e 9). Dopodiché, viene calcolata la posizione del robot e viene stampato un messaggio su terminale per comparare la posizione fornita dagli RFID con quella fornita da Gazebo.

```
    def sensor_distances_callback(self, msg):
        tags = TagArray()
        tags = msg.tags
        # Find the indexes of the tags (they can change from message to message)
        for idx, element in enumerate(msg.tags):
            # Check if the "name" field contains "tag2" or "tag3" or "tag9"
            if "tag2" in element.name:
                self.det_pow[0] = element.rssi
            elif "tag3" in element.name:
                self.det_pow[1] = element.rssi
            elif "tag9" in element.name:
                self.det_pow[2] = element.rssi



        # get new position
        x,y = self.calc_pos()
        # insert it in the moving average
        self.mean_pos.add_value(x,y)
        x,y = self.mean_pos.get_average()

        rospy.loginfo(f"\n\tESTIMATED POSITION: x = {x},  \ty = {y} \n\tREAL POSITION:      x = {self
        position = Point()
        position.x = x
        position.y = y
        # Publish the position
        self.rfid_position_pub.publish(position)
        estimated_positions.append((x, y))
        real_positions.append((self.rob_x, self.rob_y))
```

5.3.5 Callback per Gazebo

Questa callback viene chiamata ogni volta che viene ricevuto un nuovo messaggio sul topic "model_states", contenente le posizioni fornite da Gazebo. La funzione copia semplicemente la posizione del robot su una variabile membro della classe.

```
    def model_states_callback(self, msg):
        # Get the index of the robot
```

```
try:
    model_index = msg.name.index("pioneer3at_robot")
except ValueError:
    rospy.logerr("Model not found in model states message")
    return

# Get the position of the model
model_position = msg.pose[model_index].position
#rospy.loginfo("Position of the robot: x={}, y={}, z={}".format(model_position.x, model_posit
self.rob_x = model_position.x
self.rob_y = model_position.y
```

5.3.6 Grafico della traiettoria

Quando la macchina raggiunge l'obiettivo e un messaggio vuoto viene pubblicato sul topic "/target_reached", il task rfid stampa il grafico delle due traiettorie a confronto, con un'indicazione dell'errore di localizzazione medio.

```
def calculate_error(self, actual, estimated):
    error = np.sqrt((actual[0] - estimated[0])**2 + (actual[1] - estimated[1])**2)
    return error

def plot_trajectories(self):
    if len(real_positions) == 0 or len(estimated_positions) == 0:
        return

    fig, ax = plt.subplots()
    actual_x, actual_y = zip(*real_positions)
    estimated_x, estimated_y = zip(*estimated_positions)

    ax.plot(actual_x, actual_y, label='Real Trajectory')
    ax.plot(estimated_x, estimated_y, label='Estimated Trajectory')

    # Calculate and plot error
    errors = [self.calculate_error(actual, estimated) for actual, estimated in zip(real_positions
    ax.set_title(f'Mean Error: {np.mean(errors):.2f}')

    ax.legend()
    plt.show()

    # Save the plot
    plt.savefig(IMAGE_PATH)
    rospy.loginfo(f"Plot saved to {IMAGE_PATH}")
    plt.close(fig)
```

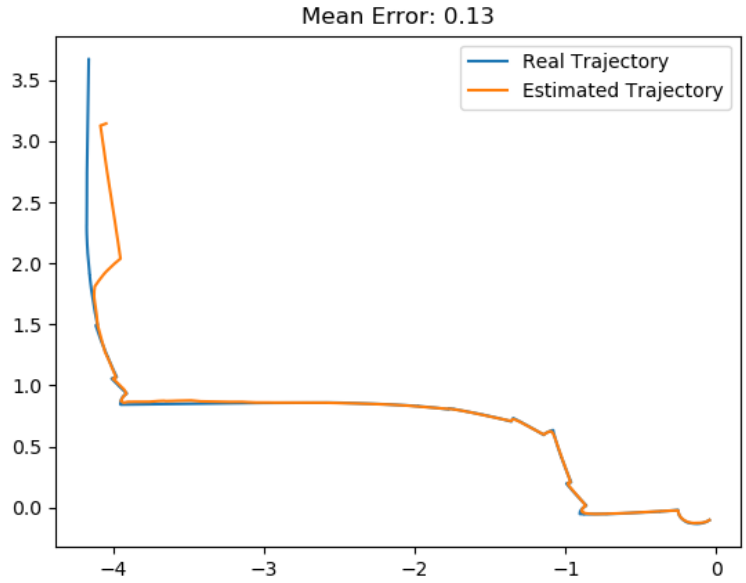


Figure 2: Traiettoria reale e stimata dagli RFID, a confronto. Rumore nullo.

5.4 Main

Il main non fa altro che creare un oggetto della classe RFID_position.

```
if __name__ == '__main__':
print("Node: RFID")
try:
    distance_calculator_node = RFID_position()
    rospy.spin()
except rospy.ROSInterruptException:
    pass
```

5.5 Analisi dei risultati

A seguito di numerosi esperimenti è potuto osservare che la misura dei sensori RFID può essere utilizzata per ottenere una stima della posizione, a patto che il robot si trovi vicino ai tag e che le letture non siano affette da rumore.
Per verificare quanto affermato, sono stati stampati i grafici della traiettoria reale e stimata, nonché una misura dell'errore di posizione medio. Di seguito possono essere osservati i diagrammi ottenuti.

Il grafico 2 mostra le due traiettorie in assenza di rumore. L'errore è molto piccolo e la precisione è sufficiente a permettere una navigazione con RFID.

Successivamente sono state effettuate delle misure spostando i tag all'interno del magazzino. Si è potuto notare che la precisione della lettura cala sensibilmente quando il robot si allontana dagli RFID e in particolare quando esce dal triangolo avente per vertici i tre RFID. Nella figura 3 si può vedere

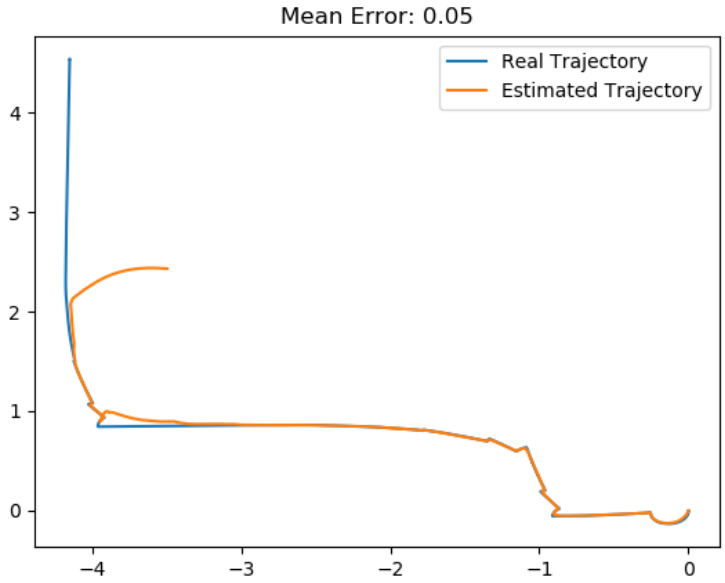


Figure 3: Traiettoria reale e stimata dagli RFID, a confronto. Rumore nullo, robot che si allontana dai tag.

come le letture risultano precise nella prima parte della traiettoria, ossia quando il robot si trova vicino ai tag, per poi peggiorare sensibilmente quando il robot si allontana.

Per quanto riguarda la robustezza al rumore, è stato osservato che la lettura diventa inutilizzabile in presenza di interferenze. L'inserimento di rumore gaussiano nella misurazione della potenza emessa dagli RFID provoca un peggioramento significativo della qualità della lettura, rendendo questo metodo sostanzialmente inutilizzabile. Questo effetto è dovuto al fatto che, nella formula utilizzata per calcolare la distanza a partire dalla potenza ricevuta, quest'ultima compare come esponente. Di conseguenza, piccole oscillazioni nella lettura della potenza si traducono in grandi errori nel risultato finale.

6 Risultati e conclusioni

Il nostro codice ha dimostrato di essere in grado di guidare con successo un robot all'interno di una mappa. Un esempio pratico del funzionamento del codice è disponibile al seguente link YouTube.
La velocità di navigazione del robot può essere migliorata mediante l'adattamento dei parametri del controllore. È tuttavia opportuno considerare la potenza computazionale del proprio computer, al fine di evitare instabilità causate da tempi di discretizzazione eccessivamente lunghi.
Per quanto riguarda la scelta dell'algoritmo di Reinforcement Learning si può osservare che, pur essendo funzionale e facilmente comprensibile, non raggiunge l'ottimalità: algoritmi operanti in spazi continui potrebbero fornire risultati superiori rispetto a quelli che operano in spazi discretizzati.
Infine, è stata effettuata un'analisi di utilizzabilità degli RFID per ottenere la posizione del robot, che ha mostrato che tale metodo sia utilizzabile solo in presenza di rumore nella lettura contenuto.

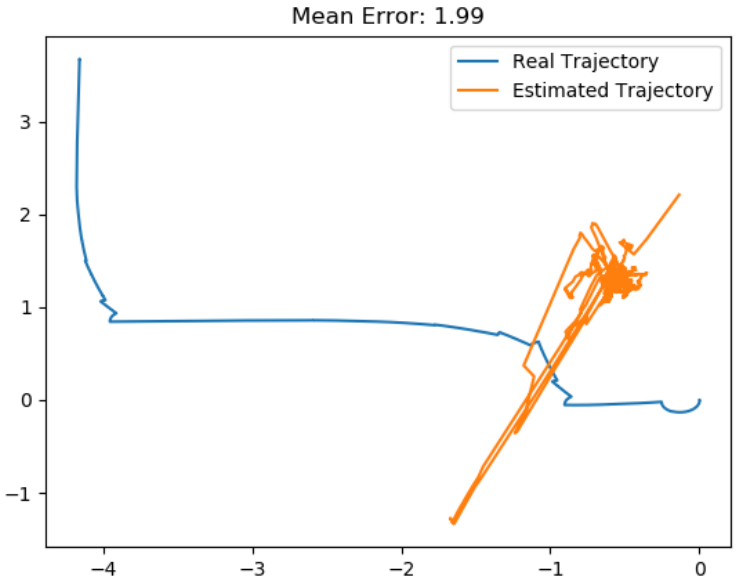


Figure 4: Traiettoria reale e stimata dagli RFID, a confronto. Rumore NON nullo.

References

[1] GitHub di AWS-Robotics, *AWS RoboMaker Small Warehouse World*, 2020.

[2] Salvatore D'Avella, *RFID Gazebo-Based Simulator With RSSI andPhase Signals for UHF Tags Localizationand Tracking*, IEEE Access, 2022.

[3] GitHub di Salvatore D'Avella, *RFID_simulator_test*, 2022.