Robótica - Wall Following

Ciencia de Datos e IA - 3º
Gonzalo Lope Carrasco - g.lope@alumnos.upm.es
Javier Pérez Vargas - javier.perez.vargas@alumnos.upm.es

Abril 2024

Índice

1. Introducción al problema

En esta práctica vamos a abordar el desafío de hacer que un robot pueda seguir una pared en un entorno simulado. Esta tarea presenta una serie de dificultades técnicas que son fundamentales para comprender los principios básicos de la robótica móvil. La capacidad de un robot para seguir una pared de manera eficiente es crucial en numerosas aplicaciones prácticas, como la navegación en interiores, la exploración de entornos desconocidos y la evitación de obstáculos.

En este contexto, exploraremos dos enfoques algorítmicos, empezando por heurísticas básicas hasta técnicas más avanzadas como el aprendizaje por refuerzo utilizando el algoritmo Q-Learning. Para el desarrollo de la práctica utilizaremos el lenguaje Python y el software CoppeliaSim, haciendo uso de un conector que permita manejar todo desde nuestro programa. El robot consta de 16 sensores LiDAR, 8 delante y 8 detrás, que nos permitirán detectar los obstáculos y las paredes a evitar.

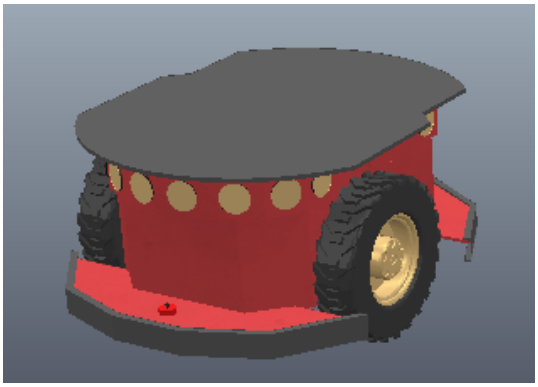

Figura 1: Robot desde arriba

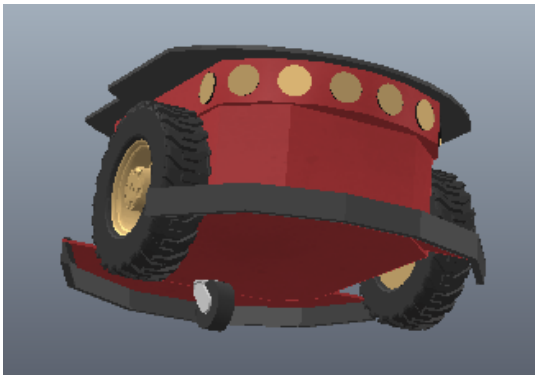

Figura 2: Robot desde abajo

2. Creación del entorno en CoppeliaSim

Para crear el entorno en CoppeliaSim, primero necesitamos diseñar un escenario que represente un espacio tridimensional donde nuestro robot pueda moverse y seguir una pared. Utilizaremos las herramientas proporcionadas por CoppeliaSim para construir un entorno realista y adecuado para nuestros propósitos.

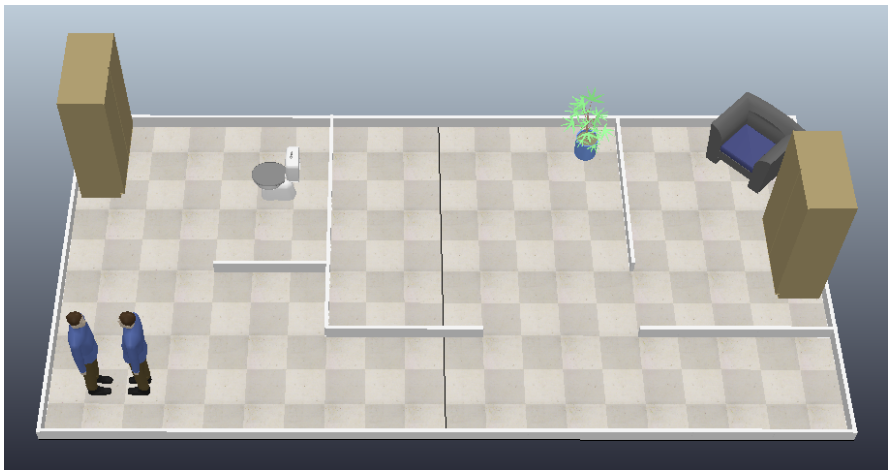

Figura 3: Escena

3. Desarrollo de la solución

3.1. Algoritmo heurístico

En primer lugar, tratamos de modificar las reglas ya existentes en el script de prueba para ver si podíamos mejorar el comportamiento del robot. A continuación, presentaremos la arquitectura de control usada para el movimiento del mismo y luego las técnicas de suavizado que usamos para probar con mayores velocidades o con menores distancias.

3.1.1. Implementación

El robot sigue el siguiente esquema de control:

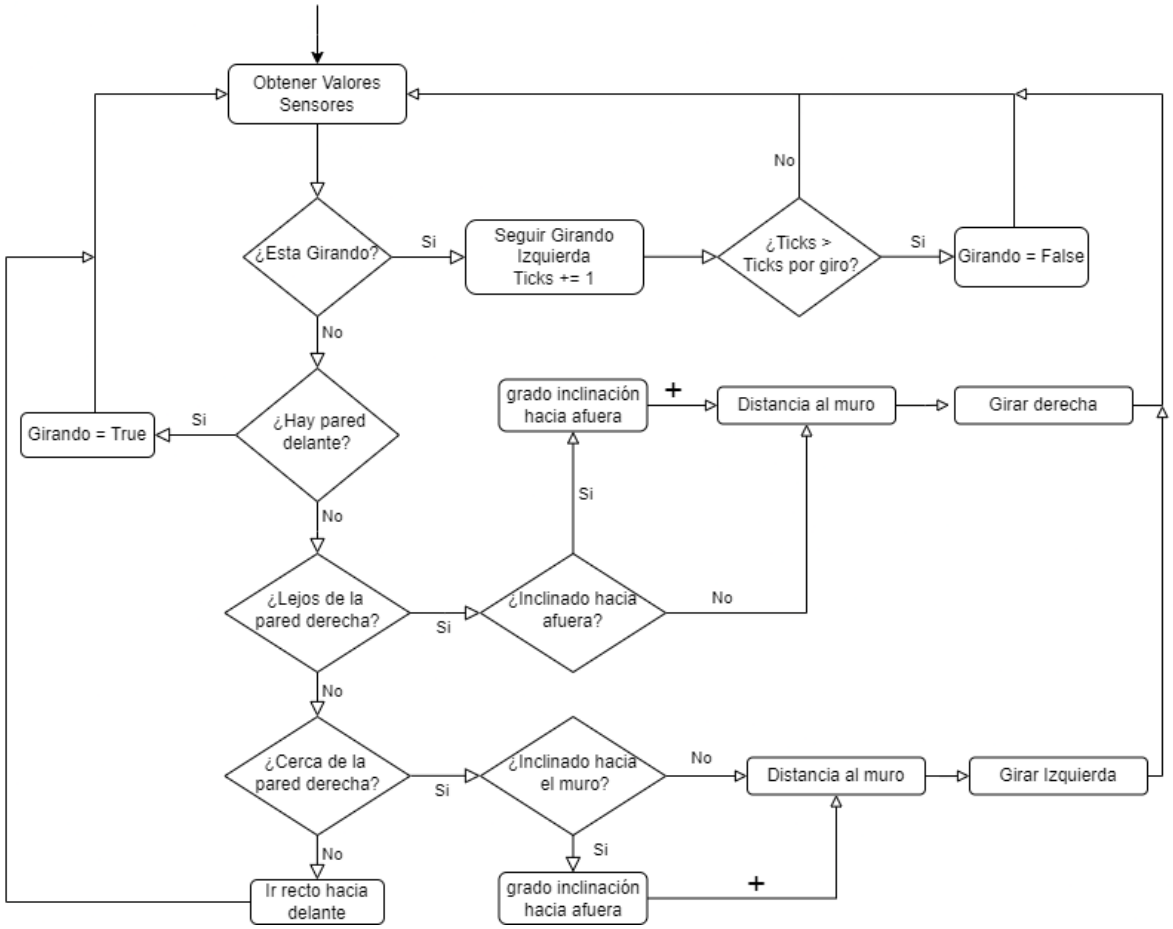

Figura 4: Esquema de Control del Robot

Nuestras aportaciones al esquema de control ya existente en el script de ejemplo son el estado de 'Girando' del robot para que cuando se encuentre una pared de frente gire un número determinado de ticks de la simulación para, de esta forma, permitir al robot hacer giros mucho más cerrados hasta que tiene vía libre para ir hacia delante. Esto lo conseguimos mediante una variable global que tenía la cuenta de ticks desde que se empezó a girar, y como se podrá ver en los resultados, permite salir de esquinas rodeadas, y girar los 90^{o} de las paredes más exactamente.

Por otro lado, para suavizar más los giros y permitir que la velocidad de las correcciones escale cuando aumentamos la velocidad hemos hecho que la 'cantidad de giro' se vea reflejada como (para girar a la derecha):

$$V_{izq} = V + (A_{incl} \cdot 2 + D_{pared} \cdot 3 \cdot V)$$
$$V_{der} = V - (A_{incl} \cdot 2 + D_{pared} \cdot 3 \cdot V)$$

Siendo A_{incl} un número directamente proporcional al ángulo con la pared, y D_{pared} la distancia a la pared desde el sensor lateral más cercano a esta. En el caso de girar a la derecha estando demasiado lejos de la pared e inclinado hacia fuera estos valores serían:

$$A_{incl} = S_7 - S_8$$
$$D_{pared} = S_8$$

Calculando y usando estos valores a cada tick, conseguimos que cuánto más cerca está de la distancia requerida con respecto a la pared menos giro haga el robot, dando lugar a las curvas suaves que podemos ver en los resultados.

3.1.2. Resultado

Como podemos observar en las imágenes y los vídeos, este algoritmo heurístico funciona muy bien con valores de velocidad bajos y un margen grande (0.35) con la pared. Sin embargo, hemos observado que el robot tiene problemas con obstáculos muy finos pues usa dos sensores laterales separados para hacer las mediciones de distancia con el obstáculo, y en caso de haber este obstáculo fino, puede dar lugar a inconsistencias que hacen que el robot se quede atascado.

A su vez, cuando aumentamos la velocidad o reducimos mucho la distancia deseada a las paredes obtenemos resultados aceptables, aunque se incrementa el número de colisiones o aparecen oscilaciones al seguir las paredes como podemos observar al usar una velocidad de 4.

3.2. Q-Learning

Nuestra segunda alternativa para resolver el problema de *Wall Following* es usar el algoritmo Q-Learning. El algoritmo Q-Learning es un método de aprendizaje por refuerzo que permite que un agente aprenda a tomar decisiones óptimas en un entorno desconocido. Utiliza una tabla de valores, conocida como *Q-table*, para estimar la utilidad esperada de tomar una acción en un estado dado, actualizando esta tabla en función de las recompensas recibidas. A través de la exploración y la explotación, el agente converge hacia una política de acción óptima que maximiza las recompensas acumuladas a largo plazo.

3.2.1. Implementación

En este apartado vamos a comentar los detalles de implementación más reseñables:

1. **Espacio de estados.** El espacio de estados dentro de nuestro entorno está determinado por los sensores LiDAR, que, al fin y al cabo, proporcionan mediciones de las distancias a las paredes circundantes en múltiples direcciones. Estas mediciones se convierten en un vector numérico que describe el estado actual del entorno inmediato al robot.

   No obstante, estas mediciones son valores continuos pertenecientes al rango $[0, 1]$. Es imposible almacenar tantos estados y el robot jamás aprendería porque sería altamente improbable que dos estados se repitieran. Por ello, hemos decidido **discretizar** el espacio de estados. Además, para reducir el número de estados, solamente hemos tenido en cuenta los 8 sensores LiDAR de delante, reduciendo así la complejidad del modelo.

   Para la discretización hemos valorado diferentes alternativas, pero finalmente hemos decidido que cada sensor tenga 3 posibles valores:

   - Cercano a obstáculo $\rightarrow$ Indicador pertenece a $[0, 0.3)$
   - Distancia media a obstáculo $\rightarrow$ Indicador pertenece a $[0.3, 0.6)$
   - No detección de obstáculo $\rightarrow$ Indicador pertenece a $[0.6, 1]$

   Por lo tanto, teniendo 8 sensores LiDAR con 3 posibles valores cada uno, se obtienen un total de $3^8 = 6561$ posibles estados. La decisión intenta buscar un equilibrio entre la complejidad del modelo y su capacidad de generalización. Más posibles estados serían capaces de adaptarse a situaciones más complicadas, pero al mismo tiempo podría llevar más tiempo entrenarlo. Menos posibles estados podrían ser la causa de un aprendizaje poco efectivo y poco generalizable.

   Para los estados iniciales hemos elegido 3 posiciones en el entorno. Así, al principio de cada episodio, el robot aparecerá en una de estas 3 posiciones (elegida de manera aleatoria), con el objetivo de encontrar diferentes estados y no "jugar" siempre en las mismas condiciones.

2. **Espacio de acciones.** Determinar las posibles acciones dado un estado también ha sido una tarea complicada. En primer lugar elegimos 10 acciones distintas, que iban desde un giro brusco a la izquierda hasta un giro brusco a la derecha, pasando por girar más lentamente y avanzar a distintas velocidades. Pero rápidamente nos dimos cuenta de que el robot oscilaba demasiado entre acciones diferentes y le costaba aprender aprendía. Finalmente propusimos un sistema de 8 acciones: $\{(-1, 1), (0, 1), (1, 1), (1.5, 1.5), (2, 2), (2.5, 2.5), (1, 0), (1, -1)\}$, donde el primer valor se corresponde con la velocidad de la rueda izquierda y el segundo con la velocidad de la rueda derecha. Así conseguimos equilibrar las acciones de giro y las de avance, teniendo 4 opciones de cada tipo.

3. **Selección de la acción.**

   Para seleccionar una acción dada se ha utilizado la técnica *epsilon-greedy*, una estrategia utilizada para equilibrar la exploración y la explotación durante la toma de decisiones. La idea básica es que el agente elige aleatoriamente una acción con una probabilidad ε y elige la mejor acción conocida (la acción con mayor Q-Value) con una probabilidad $1 - \varepsilon$. El parámetro ε determina el grado de exploración versus explotación. Un valor alto de ε favorece la exploración, y un valor

bajo de ε favorece la explotación.

También hemos usado el método **Epsilon Decay**, que reduce gradualmente el valor de ε a lo largo del tiempo. Esto significa que el agente se vuelve menos propenso a explorar y más propenso a explotar a medida que avanza en su aprendizaje. Además, hemos añadido un parámetro **EPSILON_MIN**, un valor mínimo al que puede reducirse ε mediante el epsilon decay. Esto asegura que el agente siempre mantenga un cierto grado de exploración incluso después de un largo período de entrenamiento.

4. **Sistema de recompensas.**

   Antes de empezar, hemos de mencionar que hemos tenido en cuenta dos formas de dar recompensas.

   a) La forma clásica de dar recompensas, donde, dado un estado S, se realiza una acción A que te lleva a un estado S', con una recompensa r. Así, la Q-Table de (S, A) se actualiza con la recompensa inmediata.

   b) Utilizando lo que se conoce como *delayed reward*, que no actualiza un estado - acción con la recompensa inmediata, sino por la recompensa futura dentro de varias iteraciones. Esto podría ser útil ya que el robot solo debería ser recompensado si finalmente llega a una situación segura, pero no tiene porque hacerlo en una sola acción. Por poner un ejemplo, si un robot se va a chocar con una pared de frente y gira ligeramente para evitarla, a lo mejor llega a un estado que sigue siendo peligroso (porque aún tiene la pared cerca), recibiendo una recompensa negativa; pero es probable que esa sea la mejor acción para salir de la situación. La implementación sería más costosa y necesitaría guardar los n últimos estados con una cola, siendo n el delay que queramos introducir.

   Finalmente hemos decidido usar la forma clásica, pero añadiendo un delay (time.sleep()) en el bucle de los episodios, de forma que, una vez se ejecute una acción, se mantenga durante 0.3 segundos y luego se obtiene el nuevo estado.

   Ahora bien, pasemos a las recompensas dadas:

   - $r = -50$, si alguno de los sensores es menor a 0.1. Esto implica que el robot se ha chocado, y por tanto, es el **fin del episodio**.
   - $r = -5$, si alguno de los 2 sensores que miran al frente está por debajo de 0.4 o si alguno de los 2 sensores a sus lados están por debajo de 0.3. Es decir, cuando el robot se va a chocar con una pared en frente. Es una forma de advertir que no es un estado seguro.
   - $r = +20$, si alguno de los 3 sensores que apuntan a la derecha está entre 0.4 y 0.6. Esta situación se da cuando el robot está cerca de la pared de la derecha, pero tampoco lo demasiado como para no chocarse. Es una manera de conseguir el objetivo de acercarse siempre a la pared de la derecha.
   - $r = -1$, en otro caso.

5. **Q-Table.**

   La Q-Table almacena los Q-values (o utilidad) de cada par estado - acción. Por tanto, las dimensiones de esta dependerán del espacio de estados y el espacio de acciones. En nuestro caso, $6561 \cdot 8 = 52488$. En muchas ocasiones se usa una matriz para almacenar la Q-Table, pero nosotros hemos decidido utilizar una tabla *hash*, donde la *key* es el estado y el *value* son los Q-values. Nos pareció una mejor idea por la eficiencia en el acceso y por la flexibilidad que aporta, dado que la tabla hash irá creciendo a medida que el robot experimenta con el entorno y no es necesario inicializarla con todos los posibles estados.

Como es lógico, hemos probado diferentes combinaciones de posibles estados, acciones y recompensas, buscando aquella que nos llevara a completar la tarea de manera correcta y eficiente.

3.2.2. Resultado

El funcionamiento del algoritmo Q-Learning es mejorable, especialmente si lo comparamos con el algoritmo heurístico, que para esta tarea es más adecuado. No obstante, observamos que el robot aprende, poco a poco, a evitar los muros y parece que acaba intentando seguir la pared. Aquí dejamos las gráficas de recompensas por episodios y la media de recompensas en los últimos 20 episodios; es decir, en el episodio x se muestra la media de las recompensas en los últimos 20 episodios.

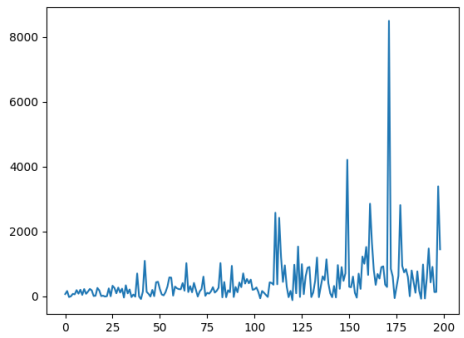

Figura 5: Recompensa por episodio

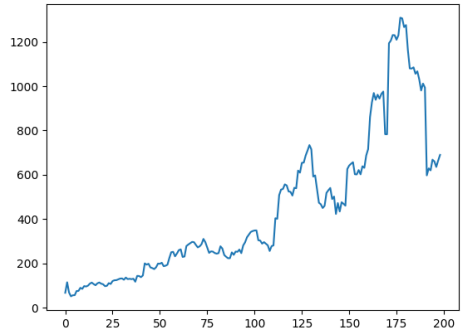

Figura 6: Media de la recompensa (últimos 20 episodios)

Observamos que el aprendizaje es bastante inestable. No obstante, a partir del episodio 100 la recompensa en ciertos episodios es mayor, algo que podemos verificar con la gráfica de la derecha, que nos permite ver que el robot está aprendiendo.

Los factores que pueden influir en la mejora del aprendizaje van desde cambiar todo el sistema de recompensas hasta modificar las posibles acciones, así como teniendo menos o más posibles estados. La discretización de los estados es un paso necesario pero al mismo tiempo una manera de perder información, ya que en ciertas situaciones límite puede ser muy relevante si un muro está a una distancia de 0.1 o 0.3, mientras que en nuestro modelo esa diferencia no se valora.

Otro forma de mejorarlo sería dejarlo entrenando durante muchos episodios (llegando a horas o incluso días), hasta que el robot sea capaz de cumplir el objetivo de seguir la pared, consiguiendo generalizar y superar cualquier situación. Esto es inviable por la falta de recursos y de tiempo.

4. Conclusiones

Dadas las observaciones del algoritmo heurístico y del algoritmo Q-Learning, podemos extraer varias conclusiones significativas.

El algoritmo Q-Learning (y sus variantes avanzadas, como DQN) tienen la fortaleza de aprender a interaccionar con un entorno sin tener ningún conocimiento previo del mismo, generalizando a otros entornos similares y donde el espacio de estados puede ser gigante. Esto lo hace útil en tareas complejas donde puede ser más complicado diseñar un algoritmo heurístico basado en reglas. No obstante, el proceso de construcción no es trivial; en este caso, la dificultad estaba en decidir cuál debía ser el espacio de estados y acciones, así como el sistema de recompensas, convirtiendo este ejercicio en una experimentación prueba - error hasta encontrar el mejor diseño.

Es por ello que un buen algoritmo heurístico como el que hemos diseñado, que incorpora conocimientos específicos del dominio como la distancia al muro, velocidades o suavizado de las curvas, realizará la tarea de *Wall Following* de una manera mucho más eficiente, con pocos fallos y sin quedarse en mínimos locales.

5. Enlaces

A continuación proporcionamos vídeos del funcionamiento de cada algoritmo:

- Algoritmo heurístico
  - Velocidad 1, Distancia 0.15: Link
  - Velocidad 1, Distancia 0.35: Link
  - Velocidad 4, Distancia 0.15: Link
- Q-Learning: Link

6. Código

6.1. Heurístico

```
def avoid(readings,turning):
    wanted_distance = 0.30
    velocity = 4
    global ticks_sincer_turn
    if turning and ( ticks_sincer_turn > 7):
        turning = False
        ticks_sincer_turn = 0
    elif turning:
        ticks_sincer_turn += 1
        return -1, +1, True

    # Avance
    if ( readings[3] > wanted_distance and readings[4] > wanted_distance ):

        if (readings[7] > wanted_distance):
            if (readings[8] < readings[7]):
                sensor_diff = (readings[7] - readings[8])*2 + (readings[8] - wanted_distance) * 3 *
                ↪  velocity
                print("2:",sensor_diff)
                lspeed, rspeed = velocity + sensor_diff, velocity - sensor_diff
            else:
                sensor_diff = (wanted_distance - readings[7]) * velocity * 1/2
                print("3:",sensor_diff)
                lspeed, rspeed = velocity - sensor_diff, velocity + sensor_diff
        elif (readings[7] < wanted_distance ):
            if (readings[8] > readings[7]):
                sensor_diff = (readings[8] - readings[7])*2 + (wanted_distance - readings[7]) * 3 *
                ↪  velocity
                print("1:",sensor_diff)
                lspeed, rspeed = velocity - sensor_diff, velocity + sensor_diff
            else:
                sensor_diff = (readings[7] - wanted_distance) * velocity * 1/2
                print("4:",sensor_diff)
                lspeed, rspeed = velocity + sensor_diff, velocity - sensor_diff
        else:
            lspeed, rspeed = +velocity, +velocity
    # Giros
    else:
        lspeed, rspeed = -1, +1
        turning = True
        ticks_sincer_turn = 0
    return lspeed, rspeed, turning


def main(args=None):
    coppelia = robotica.Coppelia()
    robot = robotica.P3DX(coppelia.sim, 'PioneerP3DX')
    coppelia.start_simulation()
    turning = False
    global ticks_sincer_turn
```

```
    ticks_sincer_turn = 0
    while coppelia.is_running():
        readings = robot.get_sonar()
        lspeed, rspeed, turning = avoid(readings,turning)
        robot.set_speed(lspeed, rspeed)
    coppelia.stop_simulation()
```

6.2. Q-Learning

Estas serían las funciones utilizadas:

```
def initialize_position(robot):
    initial_positions = [(-1.74447, +1.750), (+4.78053, -1.875), (+4.68053, +0.550)]
    random_pos = initial_positions[np.random.randint(0, len(initial_positions))]
    robot.set_position(random_pos[0], random_pos[1])

def learn(qtable, state_discrete, action_index, reward, next_state_discrete, alpha, gamma):
    if state_discrete not in qtable:
        qtable[state_discrete] = np.zeros(N_ACTIONS)
    if next_state_discrete not in qtable:
        qtable[next_state_discrete] = np.zeros(N_ACTIONS)
    qtable[state_discrete][action_index] = (1 - alpha) * qtable[state_discrete][action_index] + alpha
    ↪  * (reward + gamma * np.max(qtable[next_state_discrete]))

def select_action(qtable, state_discrete, exploration_rate):
    if np.random.rand() < exploration_rate:
        return np.random.randint(0, len(ACTIONS))
    else:
        if state_discrete not in qtable:
            qtable[state_discrete] = np.zeros(N_ACTIONS)
        return np.argmax(qtable[state_discrete])

def discretize_state(state):
    discrete_state = ""
    for i in range(8):
        if state[i] < 0.3:
            discrete_state += "0"
        elif state[i] < 0.6:
            discrete_state += "1"
        else:
            discrete_state += "2"
    return discrete_state
```

Es importante destacar que la función **set_position()** usada en **initialize_position()** es un método añadido a la clase **P3DX()**, que usa dos atributos también incorporados. Esto nos permite que el robot reaparezca tras acabar el episodio.

```
class P3DX():
    ...
    def __init__(self, sim, robot_id, use_camera=False, use_lidar=False):
        ...
        self.robot = self.sim.getObject(f'/{robot_id}')
        self.Z = +0.25879
        ...
    def set_position(self, x, y):
        self.sim.setObjectPosition(self.robot, [x, y, self.Z])
```

Y el código principal, donde se entrena el modelo:

```
def main(args=None):
    """Constants"""
    global ACTIONS, N_STATES, N_ACTIONS, EPSILON, EPSILON_DECAY, EPSILON_MIN, ALPHA, GAMMA
    EPISODES = 200
    ACTIONS = [(-1, 1), (0, 1), (1, 1), (1.5, 1.5), (2, 2), (2.5, 2.5), (1, 0), (1, -1)]
    N_STATES = 3**8
    N_ACTIONS = len(ACTIONS)
```

```
EPSILON = 1
EPSILON_DECAY = 0.99
EPSILON_MIN = 0.05
ALPHA = 0.3
GAMMA = 1

"""Initialization"""
coppelia = robotica.Coppelia()
robot = robotica.P3DX(coppelia.sim, 'PioneerP3DX')
qtable = {}
array_acc_rewards = []

"""Training"""
for i in range(EPISODES):
    coppelia.start_simulation()
    initialize_position(robot)
    state = robot.get_sonar()
    state_discrete = discretize_state(state)
    terminal = False
    acc_reward = 0
    while coppelia.is_running():

        # Epsilon greedy
        action_index = select_action(qtable, state_discrete, EPSILON)
        action = ACTIONS[action_index]

        # Execute action
        robot.set_speed(action[0], action[1])
        time.sleep(0.3)

        # Get new state and reward
        new_state = robot.get_sonar()
        new_state_discrete = discretize_state(new_state)

        # Reward system
        if min(new_state[:8]) < 0.1:
            reward = -50
            terminal = True
        else:
            if new_state[2] < 0.3 or new_state[3] < 0.4 or new_state[4] < 0.4 or new_state[5] <
            ↪  0.3:
                reward = -5
            elif (new_state[5] > 0.4 and new_state[5] < 0.6) or (new_state[6] > 0.3 and
            ↪  new_state[6] < 0.6) or (new_state[7] > 0.3 and new_state[7] < 0.6):
                reward = +20
            else:
                reward = -1

        acc_reward += reward

        # Update Q-table
        learn(qtable, state_discrete, action_index, reward, new_state_discrete, ALPHA, GAMMA)

        # Update state
        state = new_state
        state_discrete = discretize_state(state)

        # Check if terminal state
        if terminal:
            break

    # Update exploration rate
    EPSILON *= EPSILON_DECAY
    EPSILON = max(EPSILON_MIN, EPSILON)

    coppelia.stop_simulation()

    print("Episode", i+1, "| Acc. Reward:", acc_reward, "| Epsilon:", round(EPSILON, 2))
    array_acc_rewards.append(acc_reward)
```

```
    plt.plot(array_acc_rewards)
    plt.show()
    accumulated_mean = []

    for i in range(1, len(array_acc_rewards)+1):
        accumulated_mean.append(sum(array_acc_rewards[max(0, i-20):i])/min(i, 20))

    plt.plot(accumulated_mean)
    plt.show()
```