CENTRO DE DATOS E
INTELIGENCIA ARTIFICIAL
UNIVERSIDAD DE CONCEPCIÓN

Adaptación de un Simulador de Múltiples Robots Turtlebot3 con ROS2 para la toma de decisiones mediante aprendizaje por refuerzo

Victor Leiva Espinoza

Febrero 2025

1 Introducción

Este tutorial guía al lector en la implementación de una solución sim2real, es decir, la transferencia de comportamientos aprendidos en simulación a robots físicos. Se utilizarán Webots, Deepbots y ROS para configurar un entorno con múltiples TurtleBot3. Se explicará cómo definir los espacios de observación y acción, entrenar un agente de aprendizaje por refuerzo con el algoritmo PPO e integrar el comportamiento aprendido en un robot real mediante ROS.

En el mundo de la robótica moderna, la transferencia de comportamientos aprendidos en entornos simulados a robots reales—conocida como sim2real—es un desafío fundamental.

El uso de simuladores como Webots permite desarrollar, probar y entrenar agentes de inteligencia artificial en un entorno controlado y seguro, evitando riesgos y costos asociados con pruebas en robots físicos [2]. Con el apoyo de Deepbots, se facilita la creación de entornos personalizados y la integración de algoritmos de aprendizaje por refuerzo, como PPO, optimizados para aprender tareas complejas a partir de las interacciones simuladas[1].

Por otro lado, ROS (Robot Operating System) es la herramienta estándar en la comunidad robótica para la integración, comunicación y control de sistemas robóticos[3]. En este tutorial, se utiliza ROS para trasladar el comportamiento aprendido en la simulación al TurtleBot3, un robot real ampliamente empleado tanto en investigación como en aplicaciones prácticas.

El código base proporcionado define los espacios de observación y acción, implementa el bucle de entrenamiento y gestiona el despliegue del modelo. Su propósito es garantizar que el comportamiento aprendido en la simulación se reproduzca de manera consistente en el robot físico.

2 Requisitos

Es necesario contar con conocimientos de programación en Python y tener nociones generales sobre aprendizaje por refuerzo. Se recomienda, además, tener experiencia previa en Webots, Deepbots o ROS2, ya que esto facilitará el proceso. También se debe instalar el siguiente software:

- Webots R2023b o superior (disponible para Windows 10 o Ubuntu 22.04/24.04)
- Python 3.x.x
- Deepbots v1.0.0 (paquete para Python 3.x.x)
- PyTorch (paquete para Python 3.x.x)
- ROS2 (disponible en diferentes distribuciones para Windows 10 y una amplia gama de distribuciones de Ubuntu, principalmente 24.04 y 22.04. Para efectos de este documento se toma como referencia la distribución Humble Hawksbill)
- Paquetes de ROS2 para manejo de Turtlebot 3 versión 2.x

ROS2 es la instalación que puede resultar más compleja, por lo que se recomienda seguir cuidadosamente las instrucciones, preferiblemente en Linux.

2.1 Turtlebot3

Es conveniente, además, repasar las especificaciones de la línea y modelo de Robots que usaremos en este ejercicio:

La línea de robots TurtleBot son pequeñas y asequibles máquinas que hacen uso de software de código abierto. Su objetivo es ofrecer una plataforma de desarrollo robótico simple sin sacrificar la calidad. En particular el TurtleBot3 fue desarrollado por ROBOTIS y Open Robotics en 2017 como un diseño modular a diferencia de sus antecesores, haciéndolo personalizable[4].

El objetivo de TurtleBot3 es servir como plataforma para su uso en educación, investigación, prototipaje o pasatiempos a bajo costo dentro del ecosistema Robot Operating System (ROS). Para ello, la personalización hace uso de componentes mecánicos y electrónicos que a través del uso de SLAM (Simultaneous Localization and Mapping), navegación y manipulación robótica lo hacen útil en una gama de aplicaciones robóticas[4].

Existen 2 modelos principales, Burger, que será el utilizado en este trabajo, y Waffle Pi, cuyas especificaciones técnicas se resumen en el cuadro 1, mientras que en términos de construcción, su diferencia principal está en los motores y sensores. Mientras que Burger cuenta con dos DYNAMIXEL (XL430-W250-T) y un LDS-01 o un LDS-02 (desde 2022), respectivamente; Waffle Pi cuenta con una Raspberry Pi Camera v2.1 y dos DYNAMIXEL (XM430-W210-T)[4]. El resumen de los componentes está en el cuadro 2.

Table 1: Lista de Especificaciones técnicas de TurtleBot3 Burger y Waffle Pi [4]

Ítems	**Burger**	**Waffle Pi**
Velocidad máxima de traslación	0.22 m/s	0.26 m/s
Velocidad máxima de rotación	2.84 rad/s (162.72 deg/s)	1.82 rad/s (104.27 deg/s)
Carga útil máxima	15kg	30kg
Tamaño (L x A x H)	138mm x 178mm x 192mm	281mm x 306mm x 141mm
Peso (+ SBC + Batería + Sensores)	1kg	1.8kg
Umbral de escalada	10 mm o menor	10 mm o menor
Tiempo de operación esperado	2h 30m	2h
Tiempo de carga esperado	2h 30m	2h 30m
SBC (Computadora de una sola placa)	Raspberry Pi 4	Raspberry Pi 4
MCU	ARM Cortex®-M7 de 32 bits con FPU (216 MHz, 462 DMIPS)	ARM Cortex®-M7 de 32 bits con FPU (216 MHz, 462 DMIPS)
Control remoto	-	RC-100B + BT-410 Set (Bluetooth 4, BLE)
Actuador	XL430-W250	XM430-W210
LDS (Sensor de distancia láser)	Sensor de distancia láser 360 LDS-02	Sensor de distancia láser 360 LDS-02
Cámara	-	Módulo de cámara Raspberry Pi v2.1

Ítems	**Burger**	**Waffle Pi**
IMU	Giroscopio de 3 ejes, Acelerómetro de 3 ejes	Giroscopio de 3 ejes, Acelerómetro de 3 ejes
Conectores de energía	3.3V / 800mA, 5V / 4A, 12V / 1A	3.3V / 800mA, 5V / 4A, 12V / 1A
Pines de expansión	GPIO 18 pines, Arduino 32 pines	GPIO 18 pines, Arduino 32 pines
Conexiones periféricas	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4
Puertos DYNAMIXEL	RS485 x 3, TTL x 3	RS485 x 3, TTL x 3
Audio	Secuencias de pitidos programables	Secuencias de pitidos programables
LEDs programables	LED de usuario x 4	LED de usuario x 4
LEDs de estado	LED de estado de la placa x 1, LED Arduino x 1, LED de energía x 1	LED de estado de la placa x 1, LED Arduino x 1, LED de energía x 1
Botones e interruptores	Botones de presión x 2, Botón de reinicio x 1, Interruptores DIP x 2	Botones de presión x 2, Botón de reinicio x 1, Interruptores DIP x 2
Batería	Polímero de litio 11.1V 1800mAh / 19.98Wh 5C	Polímero de litio 11.1V 1800mAh / 19.98Wh 5C
Conexión a PC	USB	USB
Actualización de firmware	vía USB / vía JTAG	vía USB / vía JTAG
Adaptador de corriente (SMPS)	Entrada: 100-240V, AC 50/60Hz, 1.5A @max, Salida: 12V DC, 5A	Entrada: 100-240V, AC 50/60Hz, 1.5A @max, Salida: 12V DC, 5A

Table 2: Lista de Componentes de TurtleBot3 Burger y Waffle Pi [4].
(*) La Raspberry Pi 3 Modelo B+ se incluyó estándar a partir de 2019. Los modelos anteriores están equipados con una Raspberry Pi 3 Modelo B. La Raspberry Pi 4 Modelo B se ha incluido como estándar desde septiembre de 2021.

	Nombre de la Parte	**Burger**	**Waffle Pi**
Partes del Chasis	Waffle Plate	8	24
	Plate Support M3x35mm	4	12
	Plate Support M3x45mm	10	10
	PCB Support	12	12
	Wheel	2	2
	Tire	2	2
	Ball Caster	1	2
	Camera Bracket	0	1

	Nombre de la Parte	Burger	Waffle Pi
Motores	DYNAMIXEL (XL430-W250-T)	2	0
	DYNAMIXEL (XM430-W210-T)	0	2
Tableros	OpenCR1.0	1	1
	*Raspberry Pi	1	1
	USB2LDS	1	1
Controles Remotos	BT-410 Set (Bluetooth 4, BLE)	0	1
	RC-100B (Control Remoto)	0	1
Sensores	LDS-01 o LDS-02 (desde 2022)	1	1
	Raspberry Pi Camera v2.1	0	1
Memoria	MicroSD Card	1	1
Cables	Raspberry Pi Power Cable	1	1
	Li-Po Battery Extension Cable	1	1
	DYNAMIXEL to OpenCR Cable	2	2
	USB Cable	2	2
	Camera Cable	0	1
Poderes	SMPS 12V5A	1	1
	A/C Cord	1	1
	LIPO Battery 11.1V 1,800mAh	1	1
	LIPO Battery Charger	1	1
Herramientas	Screw driver	1	1
	Rivet tool	1	1
Varios	$\text{PH}_M 2x4mm_K$	8	8
	$\text{PH}_T 2x6mm_K$	4	8
	$\text{PH}_M 2x12mm_K$	0	4
	$\text{PH}_M 2.5x8mm_K$	16	16
	$\text{PH}_M 2.5x12mm_K$	0	20
	$\text{PH}_T 2.6x12mm_K$	16	0
	$\text{PH}_M 2.5x16mm_K$	4	4
	$\text{PH}_M 3x8mm_K$	44	140
	$\text{NUT}_M 2$	0	4
	$\text{NUT}_M 2.5$	20	24
	$\text{NUT}_M 3$	16	96
	Rivet_1	14	22
	Rivet_2	2	2
	Spacer	4	4
	Silicone Spacer	0	4
	Bracket	5	6
	Adapter Plate	1	1

3 Metodología

Para efectos de este tutorial, se creará un entorno en el que entrenemos a un grupo de 4 Turtlebot3 que jueguen un partido de fútbol como el de la Figura 1. Sin embargo, es sencillo adaptar los métodos presentados a cualquier tarea.

Figure 1: Escenario Propuesto Final

3.1 Entorno de Webots

Webots proporciona una interfaz gráfica robusta. Al iniciar, se verá una pantalla similar a la Figura 2.

Partiremos por presionar *Ctrl + Shift + N*. Esto abrirá la interfaz de creación de mundo presentada en la Figura 3 y la única opción que no se marca es "Add a rectangle arena". El nombre del mundo es irrelevante, lo llamaremos "mundo" para efectos de este documento y se creará un archivo .wbt dentro de las dependencias del programa. Este archivo tiene una estructura como la siguiente:

```
#VRML_SIM R2023b utf8
EXTERNPROTO "https://raw.githubusercontent.com/cyberbotics/
↪ webots/R2023b/projects/objects/backgrounds/protos/
↪ TexturedBackground.proto"
EXTERNPROTO "https://raw.githubusercontent.com/cyberbotics/
↪ webots/R2023b/projects/objects/backgrounds/protos/
↪ TexturedBackgroundLight.proto"
WorldInfo {
}
Viewpoint {
  orientation -0.5773 0.5773 0.5773 2.0944
  position 0 0 10
```

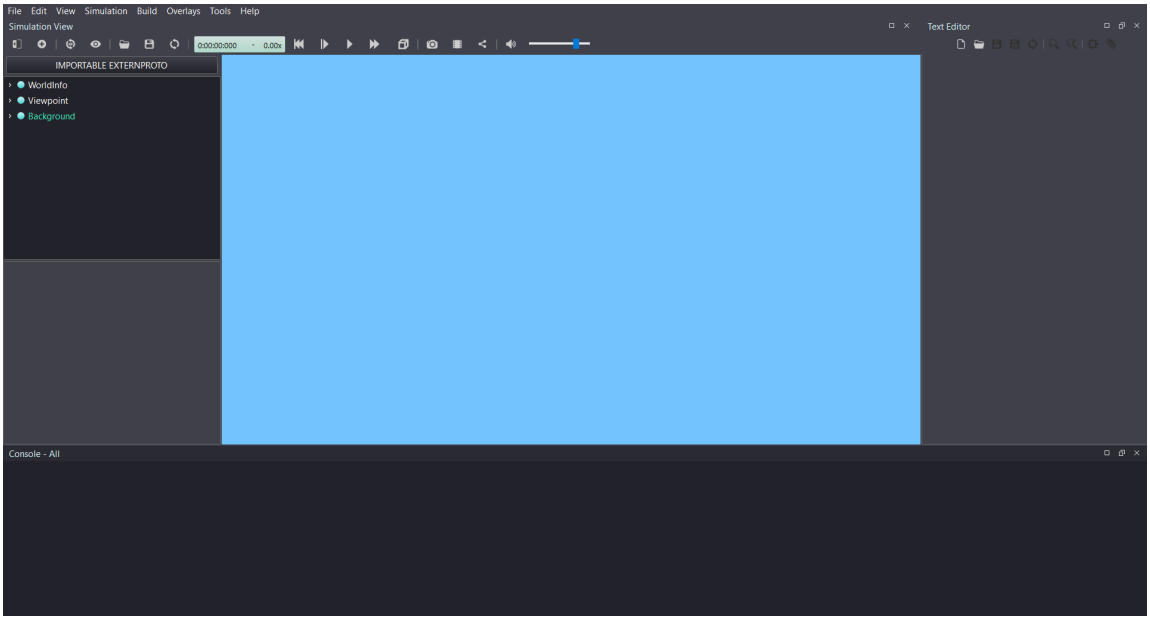

Figure 2: Interfaz inicial de Webots.

```
}
TexturedBackground {
}
TexturedBackgroundLight {
}
```

La estructura del .wbt incluye información de los nodos que se encuentren en el entorno, estos nodos son visibles en la parte izquierda de la interfaz, llamada "scene tree", por ejemplo en la Figura 2 se tiene los nodos "WorldInfo", "Viewpoint" y "Background" y al desplegarlos, pueden ser editados, lo que se reflejará en el archivo .wbt.

Para crear un entorno personalizado, primero hay que asegurarse de detener la simulación con el botón de pausa sobre el render gráfico; se deben agregar nodos presionando el símbolo + inmediatamente arriba del scene tree, al hacerlo se despliega la interfaz presentada en la Figura 4. En esta interfaz agregaremos tres nodos: "RobotstadiumSoccerField (Solid)", "SoccerBall (Solid)" y "TurtleBot3Burger (Robot)",obteniendo algo similar a la Figura 5.

Dado que todos los elementos se cargan inicialmente en la misma posición, tanto el robot como la pelota están uno dentro del otro (de hecho, si se le diera play a la simulación, ambos elementos explotarían), entonces, es necesario modificar la posición del primero. Esto se puede hacer desde el render, pero es recomendable hacer cualquier cambio a la escena modificando el nodo directamente. Así, en el scene tree se modifica el campo de $Translation$ a $-0.5, 0, 0$, como se muestra en la Figura 6. Este robot es nuestro "Delantero"

Por ahora, además, es necesario modificar el valor de *supervisor* a $True$ y agregar dentro del puerto de extensión *extensionslot* un IMU, para ello hay que darle click y agregar un nodo de la

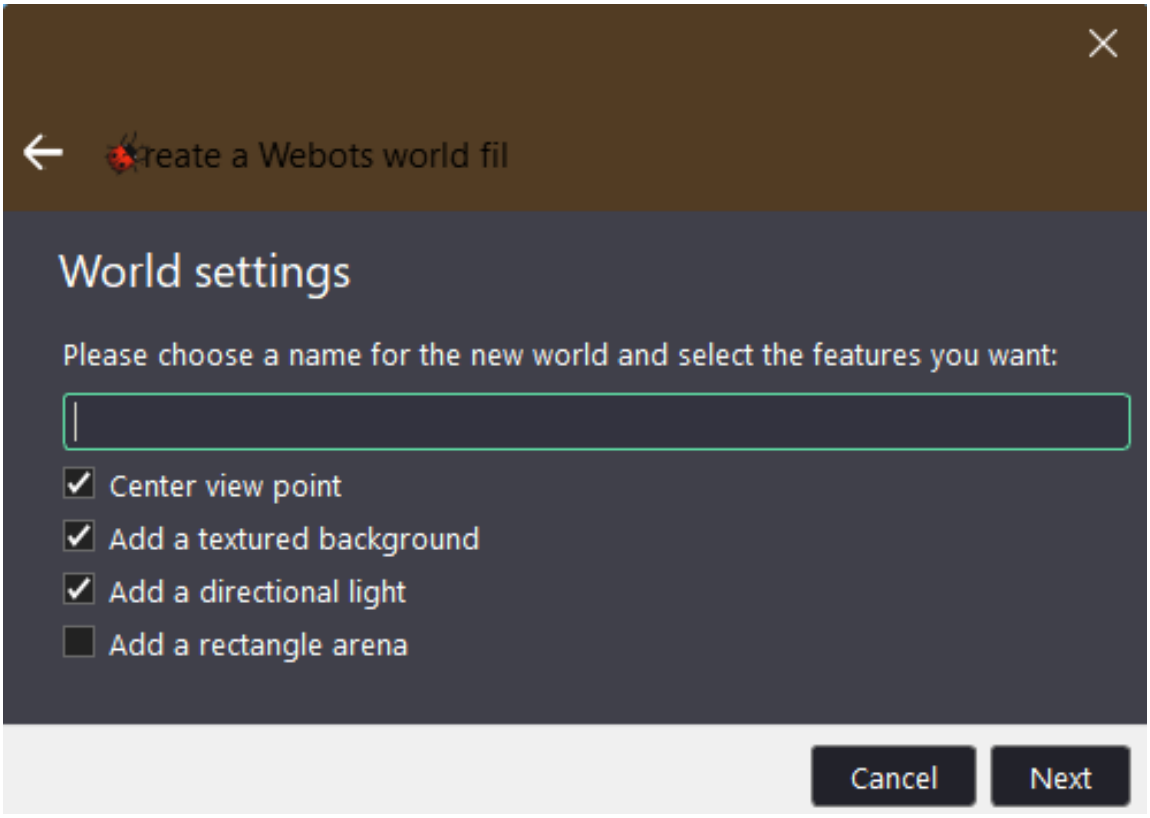

Figure 3: Interfaz de creación de mundo

misma forma anterior, presionando el signo +; el IMU genérico de webots se puede encontrar como $InertialUnit$, además, se notará que el Robot ya viene con un Lidar LDS-01.

El robot se considera un supervisor, ya que será quien gestione su propia política de recompensa y determinará su estado.

Otro campo a modificar es el DEF de la pelota, esto es un identificador que ayuda a referencia el nodo en el código. Usaremos el identificador "BALL".

Finalmente copiaremos el nodo del robot y lo pegaremos modificando su posición y rotación, idealmente y para complicar la tarea del delantero, tendremos a 3 defensores, pero si la computación se hace muy lenta, entrenar a un delantero y un defensor es igualmente válido, preferentemente descentrándolo un poco. Entonces, las nuevas posiciones serán $(4, 0, 0)$, $(3, 1, 0)$ y $(3, -1, 0)$, mientras que la rotación de todos debería ser $(0, 0, 1, \pi)$, esto es más sencillo hacerlo de manera interactiva en el render rotando el eje z hasta llegar a una rotación de $12/12\pi$. Con esto, el entorno está listo y debería verse como en la Figura 7. Ahora, es conveniente guardar y recargar el mundo.

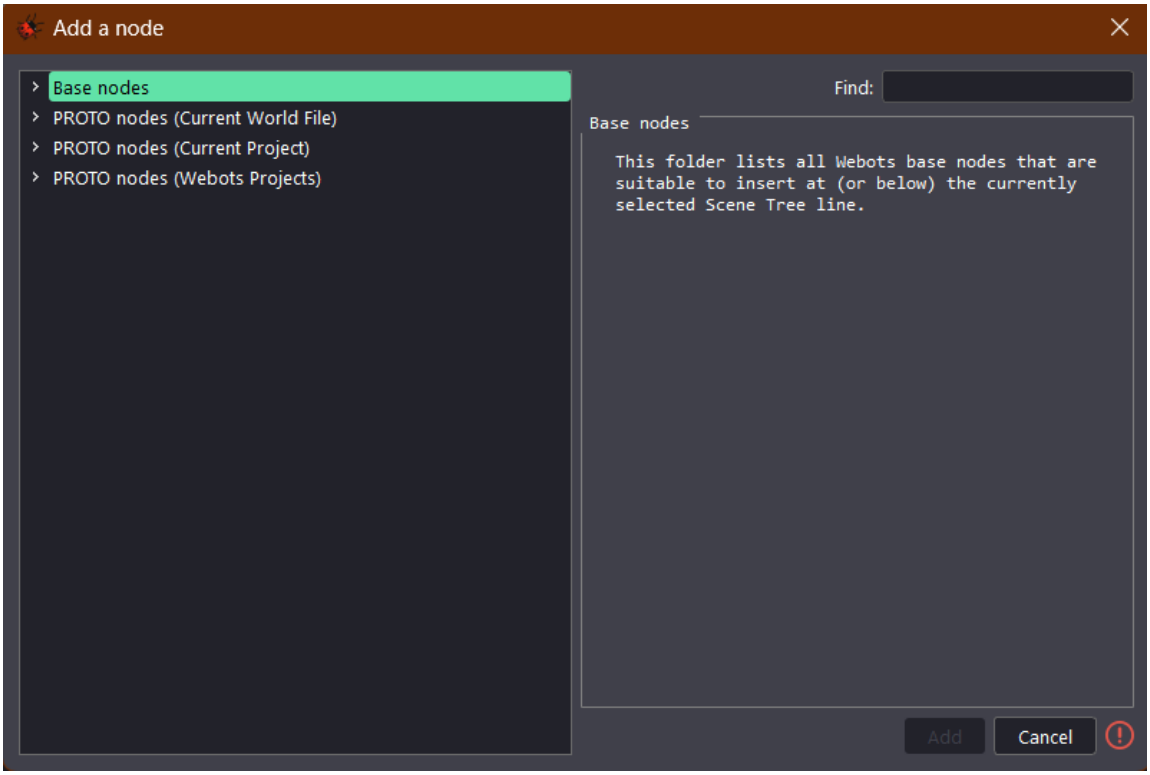

Figure 4: Interfaz de nodos disponibles

3.1.1 Calibración de PROTO Source Files

Para trasladar simulaciones a la vida real, la fidelidad de las primeras deben ser lo más altas posibles y aunque el motor físico puede ser complejo de modificar y muy por fuera del alcance de este documento, hay un factor que sí es sencillamente modificable y eso es la configuración de los documentos que definen las características de cada componente del entorno, llamados "Proto Sources" o. En particular y como ejemplo, el Proto del Lidar LDS-01 de cualquiera de los Robots es accesible al desplegar *extensionslot*, hacer click derecho y seleccionar *edit PROTO source*, éste será abierto a la derecha de el render y se verá de la siguiente manera:

```
#VRML_SIM R2023b utf8
# license: Copyright Cyberbotics Ltd. Licensed for use only with Webots.
# license url: https://cyberbotics.com/webots_assets_license
# documentation url:
↪  https://webots.cloud/run?url=https://github.com/cyberbotics/webots/blob/released/projects/device
# keywords: sensor/lidar
# Model of the Robotis LDS-01 Laser Distance Sensor
# Reference: http://www.robotis.us/360-laser-distance-sensor-lds-01-lidar/
```

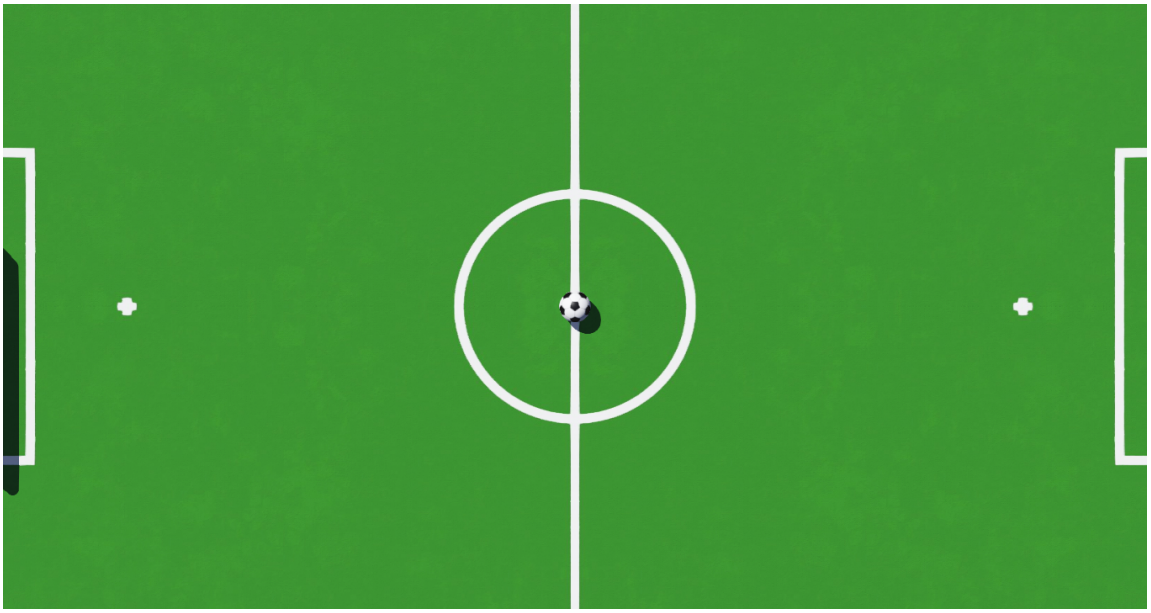

Figure 5: Configuración inicial de la escena

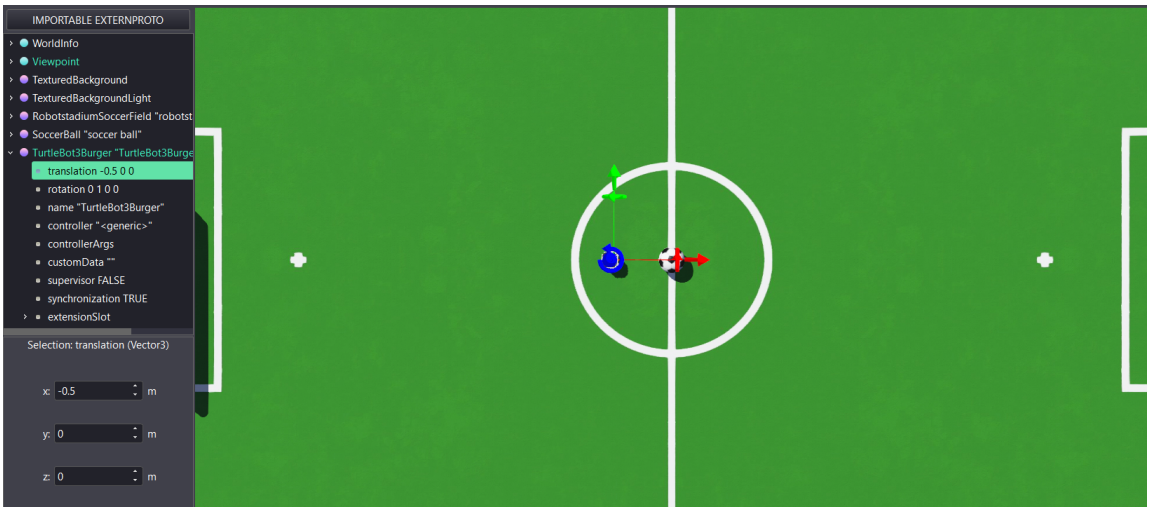

Figure 6: Configuración de la Traslación de un nodo

```
# template language: javascript

EXTERNPROTO "webots://projects/appearances/protos/RoughPolymer.proto"
```

Figure 7: Configuración Final de la escena

```
EXTERNPROTO "webots://projects/objects/factory/tools/protos/CapScrew.proto"

PROTO RobotisLds01 [
  field SFVec3f    translation       0 0 0.02
  field SFRotation rotation          0 0 1 0
  field SFString   name              "LDS-01"
  field SFFloat    noise             0.0043
  field SFBool     enablePhysics     TRUE
]
{
  Lidar {
    translation IS translation
    rotation IS rotation
    children [
      Pose {
        rotation 0 0.707103 0.70711 3.141585
        children [
          HingeJoint {
            jointParameters HingeJointParameters {
              axis 0 1 0
            }
            device [
```

```
    RotationalMotor {
      name %<= '"' + fields.name.value + '_main_motor"' >%
      maxVelocity 40
    }
  ]
  endPoint Solid {
    translation 0 -0.01 0
    rotation 1 0 0 -1.5708
    children [
      Transform {
        scale 0.001 0.001 0.001
        children [
          Shape {
            appearance DEF MAIN_APPEARANCE RoughPolymer {
              baseColor 0 0 0
              textureTransform TextureTransform {
                scale 4 4
              }
            }
            geometry Mesh {
              url "meshes/turret.obj"
            }
          }
        ]
      }
    ]
    name "lower"
  }
}
HingeJoint {
  jointParameters HingeJointParameters {
    axis 0 1 0
    anchor -0.046 0 0
  }
  device [
    RotationalMotor {
      name %<= '"' + fields.name.value + '_secondary_motor"' >%
      maxVelocity 150
    }
  ]
  endPoint Solid {
    translation 0 -0.01 0
    rotation 1 0 0 -1.5708
    children [
      Transform {
        scale 0.001 0.001 0.001
```

```
        children [
          Shape {
            appearance PBRAppearance {
              baseColor 0.886671 0.833646 0.78027
              metalness 0
            }
            geometry Mesh {
              url "meshes/wheel.obj"
            }
          }
        ]
      }
    ]
  }
}
Pose {
  translation 0 -0.01 0
  rotation 1 0 0 -1.5708
  children [
    Shape {
      appearance PBRAppearance {
        baseColor 0 0 0
        roughness 1
        metalness 0
      }
      geometry Mesh {
        url "meshes/belt.obj"
      }
    }
  ]
}
Transform {
  translation 0 -0.01 0
  rotation -1 0 0 1.5708
  scale 0.001 0.001 0.001
  children [
    Shape {
      appearance DEF MAIN_APPEARANCE RoughPolymer {
        baseColor 0 0 0
        textureTransform TextureTransform {
          scale 4 4
        }
      }
      geometry Mesh {
        url "meshes/base.obj"
      }
```

```
          }
        ]
      }
    ]
  }
  CapScrew {
    translation 0.035 0.0245 -0.0105
    rotation 0 -1 0 -1.570787
    name "screw0"
    screwRadius 0.0015
    screwLength 0.02
    enablePhysics IS enablePhysics
  }
  CapScrew {
    translation 0.035 0.0245 -0.0105
    rotation 0 -1 0 -1.570787
    name "screw1"
    screwRadius 0.0015
    screwLength 0.02
    enablePhysics IS enablePhysics
  }
  CapScrew {
    translation 0.035 0.0245 -0.0105
    rotation 0 -1 0 -1.570787
    name "screw2"
    screwRadius 0.0015
    screwLength 0.02
    enablePhysics IS enablePhysics
  }
  CapScrew {
    translation 0.035 0.0245 -0.0105
    rotation 0 -1 0 -1.570787
    name "screw3"
    screwRadius 0.0015
    screwLength 0.02
    enablePhysics IS enablePhysics
  }
]
name IS name
boundingObject Group {
  children [
    Pose {
      translation 0.014 0 -0.014
      rotation -0.577352 0.577348 0.577352 -2.094405
      children [
        Box {
```

```
            size 0.07 0.01 0.1
          }
        ]
      }
      Pose {
        translation 0 0 0
        children [
          Cylinder {
            height 0.021
            radius 0.032
          }
        ]
      }
    ]
  }
  %< if (fields.enablePhysics.value) { >%
  physics Physics {
    density -1
    mass 0.125
  }
  %< } >%
  horizontalResolution 360
  fieldOfView 6.28318
  numberOfLayers 1
  near 0.07
  minRange 0.12
  maxRange 3.5
  noise IS noise
 }
}
```

Este documento reproduce de manera fiel el aparato real[4], pero podría ser problemático usarlo con modelos más recientes, pues cuentan con un LDS-02 desde 2022 (Ver Cuadro 2), entonces si se decidiese usar éste último, los cambios a hacer en este PROTO source deben ser los siguientes:

- **Nombre del Proto:** Cambiar `RobotisLds01` por `RobotisLds02`.
- **Nombre del Sensor:** Actualizar el nombre de `"LDS-01"` a `"LDS-02"`.
- **Ruido:** Reducir de `0.0043` a `0.002`.
- **Resolución Angular:** Aumentar de `360` a `720`.
- **Rango Mínimo:** Cambiar de `0.12 m` a `0.16 m`.
- **Rango Máximo:** Extender de `3.5 m` a `8.0 m`.
- **Velocidad del Motor:** Incrementar de `40 rad/s` a `43 rad/s`.

- **Peso:** Ajustar de `0.125 kg` a `0.131 kg`.

Al guardarlo se creará una copia dentro del entorno que puede reemplazar a cualquie LDS-01 que haya en la escena. Este cambio debe ser manual.

Por otro lado, la creación de PROTO source files se escapa del objetivo de este tutorial.

3.2 Código Supervisor

Esta subsección ha sido adaptada directamente del siguiente enlace del tutorial de Deepbots

Ahora que el entorno está creado, escribiremos el código del delantero y posteriormente se modificará para aplicarlo a los defensores. Se hace click en *File* > *New* > *New Robot Controller*. Se selecciona Python como lenguaje y se elige un nombre, para efectos de este documento, lo llamaremos "controlador1". Esto creará un fichero con el mismo nombre dentro de la carpeta de controladores de nuestro proyecto que contiene el .py y éste último será abierto directamente en el editor de texto provisto por Webots.

Crearemos una clase heredada de Deepbots con métodos personalizados siguiendo la lógica de Open AI Gym.

El controlador del robot que también actúa como supervisor, recopilará datos, generará observaciones y aplicará acciones mediante un ciclo que se repetirá hasta cumplir una condición de terminación. El entrenamiento se hace bajo la Política de Optimización Proximal (PPO)[5].

3.2.1 Agente y utilities

Antes de iniciar, nos dirigiremos a la carpeta "controlador1" que contiene al controlador "controlador1.py" y crearemos dos archivos (también provistos por el enlace anterior).

El primer archivo a crear será el agente PPO, al que llamaremos "PPO_agent.py":

```
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Categorical
from torch import from_numpy, no_grad, save, load, tensor, clamp
from torch import float as torch_float
from torch import long as torch_long
from torch import min as torch_min
from torch.utils.data.sampler import BatchSampler, SubsetRandomSampler
import numpy as np
from torch import manual_seed
from collections import namedtuple

# Definimos una estructura de datos llamada "Transition" para almacenar
↪  transiciones en el entrenamiento del agente
Transition = namedtuple('Transition', ['state', 'action', 'a_log_prob', 'reward',
↪  'next_state'])


class PPOAgent:
```

```
"""
PPOAgent implementa el algoritmo de Aprendizaje por Refuerzo PPO
↪  (https://arxiv.org/abs/1707.06347).
Funciona con un conjunto de acciones discretas.
Utiliza las clases de redes neuronales Actor y Crítico definidas más abajo.
"""

def __init__(self, number_of_inputs, number_of_actor_outputs, clip_param=0.2,
↪  max_grad_norm=0.5, ppo_update_iters=5,
            batch_size=8, gamma=0.99, use_cuda=False, actor_lr=0.001,
            ↪  critic_lr=0.003, seed=None):
    super().__init__()
    if seed is not None:
        manual_seed(seed)  # Establecer una semilla para reproducibilidad

    # Hiperparámetros
    self.clip_param = clip_param
    self.max_grad_norm = max_grad_norm
    self.ppo_update_iters = ppo_update_iters
    self.batch_size = batch_size
    self.gamma = gamma
    self.use_cuda = use_cuda

    # Modelos
    self.actor_net = Actor(number_of_inputs, number_of_actor_outputs)
    self.critic_net = Critic(number_of_inputs)

    if self.use_cuda:
        self.actor_net.cuda()
        self.critic_net.cuda()

    # Creación de optimizadores
    self.actor_optimizer = optim.Adam(self.actor_net.parameters(), actor_lr)
    self.critic_net_optimizer = optim.Adam(self.critic_net.parameters(),
    ↪  critic_lr)

    # Estadísticas del entrenamiento
    self.buffer = []

def work(self, agent_input, type_="simple"):
    """
    type_ == "simple"
        Implementación de una pasada directa por la red neuronal.
    type_ == "selectAction"
        Implementación que devuelve una acción seleccionada según la
        ↪  distribución de probabilidad
```

```
            y su probabilidad correspondiente.
        type_ == "selectActionMax"
            Implementación que devuelve la acción con la mayor probabilidad.
        """
        agent_input = from_numpy(np.array(agent_input)).float().unsqueeze(0)  #
        ↪  Añadir dimensión de lote con unsqueeze
        if self.use_cuda:
            agent_input = agent_input.cuda()
        with no_grad():
            action_prob = self.actor_net(agent_input)

        if type_ == "simple":
            output = [action_prob[0][i].data.tolist() for i in
            ↪  range(len(action_prob[0]))]
            return output
        elif type_ == "selectAction":
            c = Categorical(action_prob)
            action = c.sample()
            return action.item(), action_prob[:, action.item()].item()
        elif type_ == "selectActionMax":
            return np.argmax(action_prob).item(), 1.0
        else:
            raise Exception("Tipo incorrecto en agent.work(), devolviendo
            ↪  entrada")

    def get_value(self, state):
        """
        Obtiene el valor del estado actual según el modelo crítico.

        :param state: El estado actual
        :return: Valor del estado
        """
        state = from_numpy(state)
        with no_grad():
            value = self.critic_net(state)
        return value.item()

    def save(self, path):
        """
        Guarda los modelos del actor y del crítico en la ruta proporcionada.

        :param path: Ruta donde guardar los modelos
        """
        save(self.actor_net.state_dict(), path + '_actor.pkl')
        save(self.critic_net.state_dict(), path + '_critic.pkl')
```

```
def load(self, path):
    """
    Carga los modelos del actor y del crítico desde la ruta proporcionada.

    :param path: Ruta donde están guardados los modelos
    """
    actor_state_dict = load(path + '_actor.pkl')
    critic_state_dict = load(path + '_critic.pkl')
    self.actor_net.load_state_dict(actor_state_dict)
    self.critic_net.load_state_dict(critic_state_dict)

def store_transition(self, transition):
    """
    Almacena una transición en el buffer para ser utilizada más tarde.

    :param transition: Contiene estado, acción, probabilidad de acción,
    ↪  recompensa, siguiente estado
    """
    self.buffer.append(transition)

def train_step(self, batch_size=None):
    """
    Realiza un paso de entrenamiento o actualización para los modelos de
    ↪  actor y crítico,
    basado en las transiciones almacenadas en el buffer. Luego, vacía el
    ↪  buffer.

    :param batch_size: Tamaño del lote (opcional)
    :return: None
    """
    # Si el tamaño del lote no se especifica, espera a que el buffer tenga
    ↪  suficientes transiciones
    if batch_size is None:
        if len(self.buffer) < self.batch_size:
            return
        batch_size = self.batch_size

    # Extrae estados, acciones, recompensas y probabilidades de acción de las
    ↪  transiciones en el buffer
    state = tensor([t.state for t in self.buffer], dtype=torch_float)
    action = tensor([t.action for t in self.buffer],
    ↪  dtype=torch_long).view(-1, 1)
    reward = [t.reward for t in self.buffer]
    old_action_log_prob = tensor([t.a_log_prob for t in self.buffer],
    ↪  dtype=torch_float).view(-1, 1)
```

```
# Cálculo de recompensas acumuladas
R = 0
Gt = []
for r in reward[::-1]:
    R = r + self.gamma * R
    Gt.insert(0, R)
Gt = tensor(Gt, dtype=torch_float)

if self.use_cuda:
    state, action, old_action_log_prob = state.cuda(), action.cuda(),
    ↪  old_action_log_prob.cuda()
    Gt = Gt.cuda()

# Actualización del modelo en varias iteraciones
for _ in range(self.ppo_update_iters):
    for index in
    ↪  BatchSampler(SubsetRandomSampler(range(len(self.buffer))),
    ↪  batch_size, False):
        Gt_index = Gt[index].view(-1, 1)
        V = self.critic_net(state[index])
        delta = Gt_index - V
        advantage = delta.detach()

        # Obtener las probabilidades actuales y calcular la relación
        ↪  entre la nueva y la antigua política
        action_prob = self.actor_net(state[index]).gather(1,
        ↪  action[index])
        ratio = (action_prob / old_action_log_prob[index])
        surr1 = ratio * advantage
        surr2 = clamp(ratio, 1 - self.clip_param, 1 + self.clip_param) *
        ↪  advantage

        # Actualización de la red del actor
        action_loss = -torch_min(surr1, surr2).mean()
        self.actor_optimizer.zero_grad()
        action_loss.backward()
        nn.utils.clip_grad_norm_(self.actor_net.parameters(),
        ↪  self.max_grad_norm)
        self.actor_optimizer.step()

        # Actualización de la red del crítico
        value_loss = F.mse_loss(Gt_index, V)
        self.critic_net_optimizer.zero_grad()
        value_loss.backward()
        nn.utils.clip_grad_norm_(self.critic_net.parameters(),
        ↪  self.max_grad_norm)
```

```
                self.critic_net_optimizer.step()

        del self.buffer[:]  # Vaciar buffer


class Actor(nn.Module):
    def __init__(self, number_of_inputs, number_of_outputs):
        super(Actor, self).__init__()
        self.fc1 = nn.Linear(number_of_inputs, 10)
        self.fc2 = nn.Linear(10, 10)
        self.action_head = nn.Linear(10, number_of_outputs)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        action_prob = F.softmax(self.action_head(x), dim=1)
        return action_prob


class Critic(nn.Module):
    def __init__(self, number_of_inputs):
        super(Critic, self).__init__()
        self.fc1 = nn.Linear(number_of_inputs, 10)
        self.fc2 = nn.Linear(10, 10)
        self.state_value = nn.Linear(10, 1)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        value = self.state_value(x)
        return value
```

En este caso, se asume que no será necesario tener soporte de GPU y que las redes neuronales pequeñas son capaces de resolver el problema. Esto se hace por mera simplicidad y accesibilidad.

Por otro lado agregaremos un "utilities.py" que contenga una función de normalización:

```
import numpy as np


def normalize_to_range(value, min_val, max_val, new_min, new_max, clip=False):
    """
    Normaliza un valor a un nuevo rango especificado, proporcionando el rango
    ↪  actual.

    :param value: valor a normalizar
    :param min_val: valor mínimo del rango actual, value  [min_val, max_val]
```

```
    :param max_val: valor máximo del rango actual, value  [min_val, max_val]
    :param new_min: valor mínimo del nuevo rango normalizado
    :param new_max: valor máximo del nuevo rango normalizado
    :param clip: indica si se debe recortar el valor normalizado dentro del nuevo
    ↪  rango o no
    :return: valor normalizado  [new_min, new_max]
    """
    value = float(value)
    min_val = float(min_val)
    max_val = float(max_val)
    new_min = float(new_min)
    new_max = float(new_max)

    if clip:
        return np.clip((new_max - new_min) / (max_val - min_val) * (value -
        ↪  max_val) + new_max, new_min, new_max)
    else:
        return (new_max - new_min) / (max_val - min_val) * (value - max_val) +
        ↪  new_max
```

3.2.2 Configuración del controlador

La clase descrita en esta sección puede ser escrita en un archivo aparte de "controlador1.py". Por simpleza se decide no hacerlo.

Ahora debemos borrar el contenido de "controlador1.py". La importación de módulos es simple, fuera de lo ya descrito, deepbots proporciona la clase base RobotSupervisorEnv, gym.spaces define el espacio de observación y acción del entorno y os se usa para manejar los archivos de checkpoint

```
from deepbots.supervisor.controllers.robot_supervisor_env import
↪  RobotSupervisorEnv
from utilities import normalize_to_range
from PPO_agent import PPOAgent, Transition
from gym.spaces import Box, Discrete
import numpy as np
from controller import Robot
import os
```

Se definen los límites de la cancha y del arco (en metros), así como un conjunto de valores dummy mínimos y máximos del detector Lidar LDS-01 con el que cuenta el TurtleBot3 Burger que usaremos:

```
# Definición de los límites del entorno para la posición (x, y) y otros
↪  parámetros
xlimits = [-4.5, 4.5]
ylimits = [-3, 3]
glimits = [-0.8, 0.8]
```

```
# Configuración inicial para el LiDAR: 360 medidas con valores mínimos y máximos
lidar_min = np.zeros(360)
lidar_max = np.full(360, 1e3)
```

Luego se crea la clase TBot3 heredada de RobotSupervisorEnv, permitiendo que interactúe con el simulador:

```
class TBot3(RobotSupervisorEnv):
    def __init__(self):
        """
        Inicializa el entorno del robot, definiendo el espacio de observación y
        ↪  acción,
        y configurando los dispositivos: robot, unidad inercial, LiDAR, motores y
        ↪  pelota.
        """
        super().__init__()
```

Dentro del mismo, se define el espacio de observación y el espacio de acción. El primero considera la posición, velocidad, "yaw" o rotación en el eje z y finalmente las 360 lecturas del Lidar. El espacio de acción es discreto, de 3 acciones, avanzar y rotar a derecha e izquierda.

En particular, es posible reducir la resolución del Lidar LDS-01, pero no es conveniente, pues idealmente, el Lidar podría determinar a corta distancia si se dirige a la pelota.

Además, una condición asumida es que el Robot en la vida real sea capaz de obtener su propia posición, velocidad y yaw, con lo que se recomiendo tener instalado un IMU y un RTK-GPS, pero si no fuese posible, se podría comprometer por medio de cámaras o determinar el estado de observación única y exclusivamente con el Lidar.

```
        # Definición del espacio de observación: [x, y, vx, vy, ángulo, 360
        ↪  valores LiDAR]
        self.observation_space = Box(
            low=np.array([xlimits[0], ylimits[0], -1e3, -1e3, 0, *lidar_min]),
            high=np.array([xlimits[1], ylimits[1], 1e3, 1e3, 2*np.pi,
            ↪  *lidar_max]),
            dtype=np.float64
        )
        # Definición del espacio de acción: 3 acciones discretas
        self.action_space = Discrete(3)
```

Se inicializan los devices asociados al robot, ruedas, IMU, Lidar y se identifica la pelota por medio de su DEF, "BALL":

```
        # Configuración del LiDAR y activación del point cloud
        self.lidar = self.getDevice("LDS-01")
        self.lidar.enable(self.timestep)
        self.lidar.enablePointCloud()
        self.lidar_width = self.lidar.getHorizontalResolution()
        self.lidar_max_range = self.lidar.getMaxRange()
```

```
        # Referencia a la pelota del entorno
        self.ball = self.getFromDef("BALL")

        # Inicialización de los motores de las ruedas y configuración de su
        ↪  velocidad inicial
        self.right_motor = self.getDevice("right wheel motor")
        self.left_motor = self.getDevice("left wheel motor")
        self.right_motor.setPosition(float('inf'))
        self.left_motor.setPosition(float('inf'))
        self.right_motor.setVelocity(0.0)
        self.left_motor.setVelocity(0.0)
```

Se configura la cantidad de pasos por episodio y la lista de puntajes, lo que determina, fuera del modelo, el éxito del mismo:

```
        # Parámetros para controlar la duración de cada episodio y almacenar
        ↪  puntajes
        self.steps_per_episode = 5000
        self.episode_score = 0
        self.episode_score_list = []
```

Empezando con los métodos del entorno, tenemos observaciones por defecto, que es una lista dummy necesaria para el algoritmo del agente:

```
    def get_default_observation(self):
        """
        Retorna un vector de observación por defecto compuesto de ceros.
        El tamaño es la suma de 5 variables base y los 360 valores del LiDAR.
        """
        return np.zeros(360 + 5)
```

Obtención y normalización de observaciones: En general se normalizará de -1 a 1. El Lidar necesita un tratamiento especial que elimina lecturas al infinito. Es importante notar que el Lidar tiene una distancia máxima $maxRange$ de $3.5m$, pero se normaliza a 10 por simplicidad y seguridad:

```
    def get_observations(self):
        """
        Obtiene las observaciones actuales del robot y las normaliza:
          - Posición (x, y) normalizada entre -1 y 1.
          - Velocidad (x, y) normalizada entre -1 y 1.
          - Ángulo de orientación normalizado entre -1 y 1.
          - Valores del LiDAR normalizados entre -1 y 1.
        Retorna un vector con todas estas observaciones.
        """

        minlim, maxlim = -10, 10
```

```
        # Normalización de la posición y velocidad del robot
        robot_xpos = normalize_to_range(self.robot.getPosition()[0],
        ↪  minlim,maxlim, -1, 1)
        robot_ypos = normalize_to_range(self.robot.getPosition()[1],
        ↪  minlim,maxlim, -1, 1)
        robot_xvel = normalize_to_range(self.robot.getVelocity()[0],
        ↪  minlim,maxlim, -1, 1)
        robot_yvel = normalize_to_range(self.robot.getVelocity()[1],
        ↪  minlim,maxlim, -1, 1)
        robot_angle = normalize_to_range(self.orientation.getRollPitchYaw()[2],
        ↪  -2*np.pi, 2*np.pi, -1, 1)

        # Se vectoriza la función de normalización para aplicarla al LiDAR
        normalize_vectorized = np.vectorize(normalize_to_range)

        # Obtención y normalización de los valores del LiDAR
        lidar_values = self.lidar.getRangeImage()
        if lidar_values is None:
            lidar_values = np.zeros(self.lidar.getHorizontalResolution())
        else:
            # Sustituye valores NaN o infinitos por números finitos
            lidar_values = np.nan_to_num(lidar_values, nan=10, posinf=10,
            ↪  neginf=-10)
        lidar_values = normalize_vectorized(lidar_values, -10, 10, -1, 1)

        # Combina todas las variables en un único vector de observación
        observations = np.array(
            [robot_xpos, robot_ypos, robot_xvel, robot_yvel, robot_angle,
            ↪  *lidar_values],
            dtype=np.float64
        )
        return observations
```

El método para obtener recompensas será el que se modificará principalmente para el defensor, ambas versiones consideran que la meta está en $x = 4.5$ e $y = [-0.8, 0.8]$ y calculan la distancia euclidiana d de la pelota a ella. Mientras que la recompensa del delantero es d^{-2}, la del defensor es d^2, promoviendo que uno acerque y el otro aleje el balón a la portería. Además, hay una penalización común por salirse de los límites de -1.000 puntos, pues se considera grave, y una penalización aún mayor de -10.000 si la pelota lo hace, así mismo se considera una penalización si la pelota se mantiene quieta de -10. Por último hay una recompensa para el delantero y una penalización de ± 100.000 al llegar directamente a la meta.

La versión del delantero es:

```
    def get_reward(self, action=None):
        """
```

```
Calcula la recompensa actual basada en la posición y velocidad de la pelota y del robot.
Se incentiva que la pelota se acerque a la meta (definida en x=4.5) y se penaliza si:
  - El robot o la pelota se acercan a los límites del campo.
  - La pelota se detiene (velocidad cero).
Retorna un valor de recompensa.
"""
target_x, target_y_min, target_y_max = 4.5, -0.8, 0.8

# Obtención de las posiciones y velocidades de la pelota
ball_x, ball_y = self.ball.getPosition()[0], self.ball.getPosition()[1]
ball_velx, ball_vely = self.ball.getVelocity()[0], self.ball.getVelocity()[1]

# Obtención de la posición y velocidad del robot
robot_x, robot_y = self.robot.getPosition()[0], self.robot.getPosition()[1]

# Cálculo de la distancia de la pelota a la meta y normalización de la misma
distance = np.sqrt((ball_x - target_x)**2 + (ball_y)**2)
#distance = normalize_to_range(distance, 0, 10, 0, 1)

# Recompensa inversamente proporcional al cuadrado de la distancia (se evita división por cero)
reward = 1/(distance + 1e-10)**2

# Penalización si el robot se acerca a los límites del campo
margin = 0  # margen de seguridad en metros
if (robot_x <= xlimits[0] + margin or robot_x >= xlimits[1] - margin or
    robot_y <= ylimits[0] + margin or robot_y >= ylimits[1] - margin):
    reward -= 1000

# Penalización si la pelota se acerca a los límites del campo
if (ball_x <= xlimits[0] + margin or ball_x >= xlimits[1] - margin or
    ball_y <= ylimits[0] + margin or ball_y >= ylimits[1] - margin):
    reward -= 10000

# Gran recompensa si la pelota entra en la meta
if ball_x >= 4.5 and target_y_min <= ball_y <= target_y_max:
    reward += 100000

# Penalización si la pelota se encuentra detenida
if ball_velx == 0 or ball_vely == 0:
    reward -= 10
```

```
        return reward
```

Para determinar el final de un episodio se define una función que determina si la pelota se ha salido de los límites, lo que incluye el momento en que se produce un gol:

```
        def is_done(self):
        """
        Determina si el episodio ha terminado.
        Se finaliza el episodio si la pelota o el robot se salen de los límites
        ↪  definidos.
        Retorna True si se cumple alguna de estas condiciones.
        """
        ball_x, ball_y = self.ball.getPosition()[0], self.ball.getPosition()[1]
        robot_x, robot_y = self.robot.getPosition()[0],
        ↪  self.robot.getPosition()[1]

        if not (xlimits[0] <= ball_x <= xlimits[1] and ylimits[0] <= ball_y <=
        ↪  ylimits[1]):
            return True
        #if not (xlimits[0] <= robot_x <= xlimits[1] and ylimits[0] <= robot_y <=
        ↪  ylimits[1]):
         #   return True

        return False
```

Además se agrega un método de relleno que determina el momento en que la tarea ha finalizado. Es sencillo cambiar los valores si así se quisiese:

```
    def solved(self):
        """
        Evalúa si la tarea ha sido resuelta según el desempeño en episodios
        ↪  anteriores.
        Si se han registrado suficientes episodios y el promedio de los últimos
        ↪  100 puntajes
        supera un umbral (500), retorna True; de lo contrario, False.
        """
        if len(self.episode_score_list) > 50000:
            if np.mean(self.episode_score_list[-100:]) > 500.0:
                return True
        return False
```

Por requerimiento del modelo, se agregan un par de métodos no utilizados:

```
    def get_info(self):
        """
        Retorna información adicional del entorno.
        En este caso, no se utiliza información extra y se retorna None.
```

```
        """
        return None

    def render(self, mode='human'):
        """
        Método para visualizar el entorno.
        Actualmente no se implementa la visualización.
        """
        pass
```

Y finalmente, se agrega un método que define y aplica acciones antes descritas, avanzar a hacia adelante y girar en dos direcciones, para ello se aplica una velocidad de -6 o 6 rad/s de manera independiente a cada rueda:

```
    def apply_action(self, action):
        """
        Ejecuta la acción indicada modificando la velocidad de los motores de las
        ↪  ruedas.
        Las acciones definidas son:
          0: Avanzar recto.
          1: Giro suave a la izquierda.
          2: Giro suave a la derecha.
        """
        base_speed = 6.0  # Velocidad base del robot

        if action[0] == 0:  # Avanzar recto
            left_speed = base_speed
            right_speed = base_speed
        elif action[0] == 1:  # Giro suave a la izquierda
            left_speed = -base_speed
            right_speed = base_speed
        elif action[0] == 2:  # Giro suave a la derecha
            left_speed = base_speed
            right_speed = -base_speed

        # Configura los motores para operar sin límite de posición y asigna las
        ↪  velocidades
        self.right_motor.setPosition(float('inf'))
        self.left_motor.setPosition(float('inf'))
        self.left_motor.setVelocity(left_speed)
        self.right_motor.setVelocity(right_speed)
```

Con eso, la clase principal está completa y es necesario inicializarla. Se instancia la clase TBot3, que configura, sensores y actuadores. El Agente se crea utilizando la dimensión del espacio de observación y el número de acciones. Además se establecen variables iniciales de control: El escenario no está solucionado, partimos en el episodio 0 y el máximo es 5000:

```
# ======================== BLOQUE PRINCIPAL ========================
```

```
# Instancia el entorno y crea el agente PPO utilizando las dimensiones definidas
env = TBot3()
agent = PPOAgent(number_of_inputs=env.observation_space.shape[0],
                 number_of_actor_outputs=env.action_space.n)

# Variables de control para el entrenamiento
solved = False
episode_count = 0
episode_limit = 5000

checkpoint_path = "forward_ppo"  # Nombre base para los archivos de checkpoint
```

Se agrega un nombre base para los archivos de checkpoint, actor.pkl y critic.pkl

Estos archivos se verifican al inicio del proceso dentro de la carpeta "controlador1", de tal manera de poder retomarlo en caso de que se tenga que interrumpir.

```
checkpoint_path = "forward_ppo"  # Nombre base para los archivos de checkpoint

# Verifica si existen ambos archivos de checkpoint: actor y critic
if os.path.exists(checkpoint_path + "_actor.pkl") and
↪  os.path.exists(checkpoint_path + "_critic.pkl"):
    agent.load(checkpoint_path)
    print("Checkpoint loaded from", checkpoint_path)
else:
    print("No checkpoint found. Training from scratch.")
```

Al iniciar el entrenamiento, se reinicia el entorno y la puntuación para cada episodio. Cada subsecuente episodio sigue los métodos estándar de cualquier entrenamiento, se selecciona una acción y se registra la observación, recompensa, indicador de finalización y cualquier datos adicional:

```
while not solved and episode_count < episode_limit:
    observation = env.reset()         # Reinicia el entorno para comenzar un
    ↪  nuevo episodio
    env.episode_score = 0             # Reinicia la puntuación del episodio

    # Bucle interno que recorre cada paso dentro del episodio
    for step in range(env.steps_per_episode):
        # El agente selecciona una acción basándose en la observación actual
        selected_action, action_prob = agent.work(observation,
        ↪  type_="selectAction")
        new_observation, reward, done, info = env.step([selected_action])
```

Cada transición se almacena para ser utilizada en el entrenamiento. Si el episodio termina se actualiza lista de puntajes y se realiza un paso con todas las transiciones acumuladas. Además, se imprime el resultado del episodio, se incrementa el contador y cada 100 episodios se guarda el modelo.

```
        # Crea y almacena la transición para entrenamiento posterior
        trans = Transition(observation, selected_action, action_prob, reward,
        ↪  new_observation)
        agent.store_transition(trans)

        env.episode_score += reward  # Actualiza la puntuación acumulada del
        ↪  episodio
        observation = new_observation  # Actualiza la observación para el
        ↪  siguiente paso

        # Si se cumple la condición de término del episodio, finaliza el ciclo
        ↪  interno
        if done:
            env.episode_score_list.append(env.episode_score)
            agent.train_step(batch_size=step + 1)  # Ejecuta un paso de
            ↪  entrenamiento con la transición almacenada
            solved = env.solved()                  # Verifica si el desempeño
            ↪  cumple con la condición de solución
            break

    print(f"Episode #{episode_count} finished with score: {env.episode_score}
    ↪  FORWARD")
    episode_count += 1

    # Guarda el estado del agente cada 500 episodios
    if episode_count % 100 == 0:
        agent.save(checkpoint_path)
        print("Checkpoint saved at episode", episode_count)
```

Una vez terminado el entrenamiento, se inicia un bucle de prueba simplificado: se muestra si la tarea se ha resuelto, se reinicia el entorno y, en un bucle infinito, el agente ejecuta la acción de mayor probabilidad sin exploración, actualizando la observación mediante el método step; si se activa el flag de finalización, se reinicia el entorno:

```
# Mensaje de despliegue según si la tarea se resolvió o no durante el
↪  entrenamiento
if not solved:
    print("Task is not solved, deploying agent for testing...")
elif solved:
    print("Task is solved, deploying agent for testing...")

# Bucle infinito para desplegar el agente en modo prueba (testing)
observation = env.reset()
env.episode_score = 0.0

while True:
    # El agente selecciona la acción de máxima probabilidad para el despliegue
```

```
selected_action, action_prob = agent.work(observation,
↪  type_="selectActionMax")
observation, _, done, _ = env.step([selected_action])
if done:
    observation = env.reset()
```

Con esto el controlador1 está completo, una versión puede ser encontrada en el anexo 3.3.1 al final de este documento.

3.2.3 Asignación de controladores

Para asignar lo anterior al delantero, luego de segurarnos que la simulación esté pausada y guardada, basta con desplegar su nodo en el scene tree y modificar el campo *controller* eligiendo el "controlador1", esta información también será almacenada en "mundo.wbt".

Ahora, dado que tenemos otros tres robots sin controlador, podemos hacer una de dos, crear 3 controladores .py por separado y heredar la clase -modificando el método de recompensa-, así como importar el agente y utilities desde el directorio del controlador ya creado o, más sencillo, copiarlo y pegarlo dentro del directorio de controladores.

Independiente del método, es importante que en estos nuevos directorios o carpetas, el archivo .py considere el siguiente método de recompensa:

```
def get_reward(self, action=None):
    """
    Calcula la recompensa actual basada en la posición y velocidad de la
    ↪  pelota y del robot.
    Se incentiva que la pelota se acerque a la meta (definida en x=4.5) y se
    ↪  penaliza si:
      - El robot o la pelota se acercan a los límites del campo.
      - La pelota se detiene (velocidad cero).
    Retorna un valor de recompensa.
    """
    target_x, target_y_min, target_y_max = 4.5, -0.8, 0.8

    # Obtención de las posiciones y velocidades de la pelota
    ball_x, ball_y = self.ball.getPosition()[0], self.ball.getPosition()[1]
    ball_velx, ball_vely = self.ball.getVelocity()[0],
    ↪  self.ball.getVelocity()[1]

    # Obtención de la posición y velocidad del robot
    robot_x, robot_y = self.robot.getPosition()[0],
    ↪  self.robot.getPosition()[1]

    # Cálculo de la distancia de la pelota a la meta y normalización de la
    ↪  misma
    distance = np.sqrt((ball_x - target_x)**2 + (ball_y)**2)
    #distance = normalize_to_range(distance, 0, 10, 0, 1)
```

```
# Recompensa inversamente proporcional al cuadrado de la distancia (se
↪  evita división por cero)
reward = (distance + 1e-10)**2

# Penalización si el robot se acerca a los límites del campo
margin = 0  # margen de seguridad en metros
if (robot_x <= xlimits[0] + margin or robot_x >= xlimits[1] - margin or
    robot_y <= ylimits[0] + margin or robot_y >= ylimits[1] - margin):
    reward -= 1000

# Penalización si la pelota se acerca a los límites del campo
if (ball_x <= xlimits[0] + margin or ball_x >= xlimits[1] - margin or
    ball_y <= ylimits[0] + margin or ball_y >= ylimits[1] - margin):
    reward -= 10000

# Gran recompensa si la pelota entra en la meta
if ball_x >= 4.5 and target_y_min <= ball_y <= target_y_max:
    reward -= 100000

# Penalización si la pelota se encuentra detenida
if ball_velx == 0 or ball_vely == 0:
    reward -= 10

return reward
```

Además es conveniente modificar la variable *checkpoint_path* a *defender* o incluso numerarlo del 1 al 3

Luego, es importante notar que las nuevas carpetas de controladores deben llevar el mismo nombre que los .py, es decir, si el segundo controlador se llama "controlador2": Dentro de la carpeta llamada "controlador2" el archivo debe llamarse "controlador2.py"

Estos nuevos controladores deben ser asignados a los defensores de la misma manera antes descrita.

Con todos los controladores bien asignados, si todos los pasos fueron reproducidos de manera correcta, la escena está lista y bastaría con guardar y presionar el botón de play.

3.2.4 Adaptación a otros problemas

Para adaptar el entorno, simulación y posterior adaptación sim2real basta con redefinir el sistema de recompensa y si se quiere, de acciones.

3.3 Traspaso a Robot Real

Para el traspaso sim2real, los requerimientos no son pocos, a priori, se necesita recrear lo más fielmente posible al escenario simulado: La cancha es un espacio de $9 \times 6[m]$ rodeado por paredes de $10.4 \times 7.4[m]$ en cuyos extremos deben posicionarse dos postes de $0.1[m]$ de diámetro a $0.8[m]$ del centro. Si las posiciones propuestas se siguieron, además, como en la Figura 7, las posiciones iniciales de cada robot deben ser desde el centro $(0, 0)$: $(-0.5, 0), (4, 0), (3, 1)$ y $(3, -1)$ en metros.

Además, se espera que cada uno de los TurtleBot3 Burger cuente no sólo con el Lidar LDS-01 o LDS-02 con el que viene equipado, sino que idealmente y gracias a la modularidad se deberían incluir un RTK-GPS, un odómetro y un IMU, cuya configuración específica es altamente dependiente del modelo y configuración del entorno real.

En caso de que no fuese posible incluir estos aparatos, como ya fue explicado en la subsección de configuración del controlador, sería necesario redefinir el espacio de observación para que sólo funcione con Lidar.

Independiente de la configuración del hardware, para el traspaso se hará uso de ROS2 Humble, para la cual se asume una correcta instalación.

3.3.1 Nodo de ROS

En ROS2, los nodos se estructuran dentro de clases que heredan de *rclpy.node.Node*. Este será el nodo principal para manejar cada robot de manera independiente:

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan, Imu, NavSatFix
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
import numpy as np
import torch

class TurtleBot3Controller(Node):
    def __init__(self):
        super().__init__('turtlebot3_controller')
```

Luego se definen los suscriptores, que permiten obtener datos sensoriales del Turtlebot y el publicador que publica comandos de velocidad lineal y angular en el tema $cmd_vel, controlando el movimiento del TurtleBot3$:

```
        # Inicialización de publicadores y suscriptores
        self.lidar_sub = self.create_subscription(LaserScan, '/scan',
        ↪  self.lidar_callback, 10)
        self.imu_sub = self.create_subscription(Imu, '/imu/data',
        ↪  self.imu_callback, 10)
        self.gps_sub = self.create_subscription(NavSatFix, '/gps/fix',
        ↪  self.gps_callback, 10)
        self.odom_sub = self.create_subscription(Odometry, '/odom',
        ↪  self.odom_callback, 10)
        self.cmd_vel_pub = self.create_publisher(Twist, '/cmd_vel', 10)
```

Se definen las variables iniciales de los sensores y se carga el modelo entrenado como sigue:

```
        # Variables de sensores
        self.lidar_data = np.zeros(360)
        self.robot_position = [0.0, 0.0]  # [x, y] del GPS
        self.robot_velocity = [0.0, 0.0]  # [vx, vy] de odometría
        self.robot_angular_velocity = 0.0  # Velocidad angular del robot
```

```
        # Carga de los modelos del Actor y el Crítico
        self.actor = torch.load("actor.pkl")
        self.critic = torch.load("critic.pkl")
```

Los datos sensoriales deben adaptarse a las normalizaciones utilizadas dentro de la simulación. Por ejemplo, las lecturas de Lidar LDS-01 deben tratarse como sigue:

```
    def lidar_callback(self, msg):
        self.lidar_data = np.nan_to_num(msg.ranges, nan=10, posinf=10,
        ↪  neginf=-10)
```

Por otro lado, aunque dependiente de los modelos específicos, el resto de sensores deberían seguir estructuras como las siguientes:

```
    def imu_callback(self, msg):
        # Extraer yaw únicamente
        quaternion = msg.orientation
        _, _, self.robot_yaw = self.extract_orientation(quaternion)

    def gps_callback(self, msg):
        # Extraer posición GPS
        self.robot_position = [msg.latitude, msg.longitude]

    def odom_callback(self, msg):
        # Extraer velocidades lineales y angular del robot
        self.robot_velocity = [msg.twist.twist.linear.x,
        ↪  msg.twist.twist.linear.y]
        self.robot_angular_velocity = msg.twist.twist.angular.z

    def extract_orientation(self, quaternion):
        # Convierte cuaterniones a roll, pitch, yaw
        import tf_transformations
        return tf_transformations.euler_from_quaternion([
            quaternion.x, quaternion.y, quaternion.z,
```

En este código se han asumido calibraciones manuales estándar.

Las observaciones deben hacer uso de los componentes ya mencionados: RTK-GPS, IMU, odómetro y Lidar; así como las normalizaciones ya aplicadas en la simulación, como sigue:

```
    def get_observations(self):
        robot_x = normalize_to_range(self.gps_position[0], -4.5, 4.5, -1, 1)
        robot_y = normalize_to_range(self.gps_position[1], -3.0, 3.0, -1, 1)
        roll, pitch, yaw = self.imu_orientation
        lidar_normalized = np.vectorize(normalize_to_range)(self.lidar_data, 0,
        ↪  10, -1, 1)

        return np.array([robot_x, robot_y, roll, pitch, yaw, *lidar_normalized])
```

Para seleccionar acciones es necesario utilizar el actor, producto del entrenamiento, para ello se convierten las observaciones a un tensor, el actor genera las probabilidades para cada acción posible en base al tensor y la acción se toma de forma determinista:

```
def select_action(self, observations):
    # Inferencia con el modelo Actor
    with torch.no_grad():
        observations_tensor = torch.tensor(observations,
        ↪  dtype=torch.float32).unsqueeze(0)
        action_probs = self.actor(observations_tensor)
        action = torch.argmax(action_probs).item()  # Selección determinista
        ↪  de acción
    return
```

Luego, para definir la aplicación de acciones es necesario tratar de emular las ya definidas en la simulación, en este caso, el equivalente de ROS aplica velocidades lineales y angulares en lugar de aplicar velocidads angulares a cada rueda, con lo que, aunque mejores cálculos se podrían hacer, dado que la toma de decisiones se hace para cada estado, no hay mucha diferencia si se toman valores arbitrarios en saltos de tiempo disimilares a los simulados.

```
def apply_action(self, action):
    vel_msg = Twist()
    base_speed = 0.2  # Velocidad lineal base en m/s
    base_turn = 0.5   # Velocidad angular base en rad/s

    if action == 0:  # Avanzar recto
        vel_msg.linear.x = base_speed
        vel_msg.angular.z = 0.0
    elif action == 1:  # Giro suave a la izquierda
        vel_msg.linear.x = 0  # Reducir velocidad lineal
        vel_msg.angular.z = base_turn  # Giro positivo (hacia la izquierda)
    elif action == 2:  # Giro suave a la derecha
        vel_msg.linear.x = 0  # Reducir velocidad lineal
        vel_msg.angular.z = -base_turn  # Giro negativo (hacia la derecha)

    self.cmd_vel_pub.publish(vel_msg)
```

Es necesario mentener un bucle continuo que procese los datos sensoriales, seleccionar una accion y ejecutarla:

```
def run(self):
    while rclpy.ok():
        rclpy.spin_once(self)  # Procesa callbacks
        observations = self.get_observations()
        action = self.select_action(observations)
        self.apply_action(action)
```

Finalmente se define y ejecuta el punto de entrada del programa que inicializa ROS2:

```
def main(args=None):
    rclpy.init(args=args)
    controller = TurtleBot3Controller()
    try:
        controller.run()
    except KeyboardInterrupt:
        controller.get_logger().info('Shutting down...')
        controller.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

El código puede ser revisado en el Anexo 3.3.1

Anexo

Controlador de Webots + Deepbots

```
from deepbots.supervisor.controllers.robot_supervisor_env import
↪  RobotSupervisorEnv
from utilities import normalize_to_range
from PPO_agent import PPOAgent, Transition
from gym.spaces import Box, Discrete
import numpy as np
import os

# Definición de los límites del entorno para la posición (x, y) y otros
↪  parámetros
xlimits = [-4.5, 4.5]
ylimits = [-3, 3]
glimits = [-0.8, 0.8]

# Configuración inicial para el LiDAR: 360 medidas con valores mínimos y máximos
lidar_min = np.zeros(360)
lidar_max = np.full(360, 1e3)

class TBot3(RobotSupervisorEnv):
    def __init__(self):
        """
        Inicializa el entorno del robot, definiendo el espacio de observación y
        ↪  acción,
        y configurando los dispositivos: robot, unidad inercial, LiDAR, motores y
        ↪  pelota.
        """
        super().__init__()
```

```
        # Definición del espacio de observación: [x, y, vx, vy, ángulo, 360
        ↪  valores LiDAR]
        self.observation_space = Box(
            low=np.array([xlimits[0], ylimits[0], -1e3, -1e3, 0, *lidar_min]),
            high=np.array([xlimits[1], ylimits[1], 1e3, 1e3, 2*np.pi,
            ↪  *lidar_max]),
            dtype=np.float64
        )
        # Definición del espacio de acción: 3 acciones discretas
        self.action_space = Discrete(3)

        # Inicialización del robot y la unidad inercial para obtener orientación
        self.robot = self.getSelf()
        self.orientation = self.getDevice("inertial unit")
        self.orientation.enable(self.timestep)

        # Configuración del LiDAR y activación del point cloud
        self.lidar = self.getDevice("LDS-01")
        self.lidar.enable(self.timestep)
        self.lidar.enablePointCloud()
        self.lidar_width = self.lidar.getHorizontalResolution()
        self.lidar_max_range = self.lidar.getMaxRange()

        # Referencia a la pelota del entorno
        self.ball = self.getFromDef("BALL")

        # Inicialización de los motores de las ruedas y configuración de su
        ↪  velocidad inicial
        self.right_motor = self.getDevice("right wheel motor")
        self.left_motor = self.getDevice("left wheel motor")
        self.right_motor.setPosition(float('inf'))
        self.left_motor.setPosition(float('inf'))
        self.right_motor.setVelocity(0.0)
        self.left_motor.setVelocity(0.0)

        # Parámetros para controlar la duración de cada episodio y almacenar
        ↪  puntajes
        self.steps_per_episode = 5000
        self.episode_score = 0
        self.episode_score_list = []

    def get_default_observation(self):
        """
        Retorna un vector de observación por defecto compuesto de ceros.
        El tamaño es la suma de 5 variables base y los 360 valores del LiDAR.
```

```
        """
        return np.zeros(360 + 5)

    def get_observations(self):
        """
        Obtiene las observaciones actuales del robot y las normaliza:
          - Posición (x, y) normalizada entre -1 y 1.
          - Velocidad (x, y) normalizada entre -1 y 1.
          - Ángulo de orientación normalizado entre -1 y 1.
          - Valores del LiDAR normalizados entre -1 y 1.
        Retorna un vector con todas estas observaciones.
        """

        minlim, maxlim = -10, 10


        # Normalización de la posición y velocidad del robot
        robot_xpos = normalize_to_range(self.robot.getPosition()[0],
        ↪  minlim,maxlim, -1, 1)
        robot_ypos = normalize_to_range(self.robot.getPosition()[1],
        ↪  minlim,maxlim, -1, 1)
        robot_xvel = normalize_to_range(self.robot.getVelocity()[0],
        ↪  minlim,maxlim, -1, 1)
        robot_yvel = normalize_to_range(self.robot.getVelocity()[1],
        ↪  minlim,maxlim, -1, 1)
        robot_angle = normalize_to_range(self.orientation.getRollPitchYaw()[2],
        ↪  -2*np.pi, 2
*np.pi, -1, 1)

        # Se vectoriza la función de normalización para aplicarla al LiDAR
        normalize_vectorized = np.vectorize(normalize_to_range)

        # Obtención y normalización de los valores del LiDAR
        lidar_values = self.lidar.getRangeImage()
        if lidar_values is None:
            lidar_values = np.zeros(self.lidar.getHorizontalResolution())
        else:
            # Sustituye valores NaN o infinitos por números finitos
            lidar_values = np.nan_to_num(lidar_values, nan=10, posinf=10,
            ↪  neginf=-10)
        lidar_values = normalize_vectorized(lidar_values, -10, 10, -1, 1)

        # Combina todas las variables en un único vector de observación
        observations = np.array(
            [robot_xpos, robot_ypos, robot_xvel, robot_yvel, robot_angle,
            ↪  *lidar_values],
```

```
            dtype=np.float64
        )
        return observations

    def get_reward(self, action=None):
        """
        Calcula la recompensa actual basada en la posición y velocidad de la
        ↪  pelota y del robot.
        Se incentiva que la pelota se acerque a la meta (definida en x=4.5) y se
        ↪  penaliza si:
          - El robot o la pelota se acercan a los límites del campo.
          - La pelota se detiene (velocidad cero).
        Retorna un valor de recompensa.
        """
        target_x, target_y_min, target_y_max = 4.5, -0.8, 0.8

        # Obtención de las posiciones y velocidades de la pelota
        ball_x, ball_y = self.ball.getPosition()[0], self.ball.getPosition()[1]
        ball_velx, ball_vely = self.ball.getVelocity()[0],
        ↪  self.ball.getVelocity()[1]

        # Obtención de la posición y velocidad del robot
        robot_x, robot_y = self.robot.getPosition()[0],
        ↪  self.robot.getPosition()[1]

        # Cálculo de la distancia de la pelota a la meta y normalización de la
        ↪  misma
        distance = np.sqrt((ball_x - target_x)**2 + (ball_y)**2)
        #distance = normalize_to_range(distance, 0, 10, 0, 1)

        # Recompensa inversamente proporcional al cuadrado de la distancia (se
        ↪  evita división por cero)
        reward = 1/(distance + 1e-10)**2

        # Penalización si el robot se acerca a los límites del campo
        margin = 0  # margen de seguridad en metros
        if (robot_x <= xlimits[0] + margin or robot_x >= xlimits[1] - margin or
            robot_y <= ylimits[0] + margin or robot_y >= ylimits[1] - margin):
            reward -= 1000

        # Penalización si la pelota se acerca a los límites del campo
        if (ball_x <= xlimits[0] + margin or ball_x >= xlimits[1] - margin or
            ball_y <= ylimits[0] + margin or ball_y >= ylimits[1] - margin):
            reward -= 10000

        # Gran recompensa si la pelota entra en la meta
```

```
        if ball_x >= 4.5 and target_y_min <= ball_y <= target_y_max:
            reward += 100000

        # Penalización si la pelota se encuentra detenida
        if ball_velx == 0 or ball_vely == 0:
            reward -= 10

        return reward

    def is_done(self):
        """
        Determina si el episodio ha terminado.
        Se finaliza el episodio si la pelota o el robot se salen de los límites
        ↪  definidos.
        Retorna True si se cumple alguna de estas condiciones.
        """
        ball_x, ball_y = self.ball.getPosition()[0], self.ball.getPosition()[1]
        robot_x, robot_y = self.robot.getPosition()[0],
        ↪  self.robot.getPosition()[1]

        if not (xlimits[0] <= ball_x <= xlimits[1] and ylimits[0] <= ball_y <=
        ↪  ylimits[1]):
            return True
        #if not (xlimits[0] <= robot_x <= xlimits[1] and ylimits[0] <= robot_y <=
        ↪  ylimits[1]):
         #   return True

        return False

    def solved(self):
        """
        Evalúa si la tarea ha sido resuelta según el desempeño en episodios
        ↪  anteriores.
        Si se han registrado suficientes episodios y el promedio de los últimos
        ↪  100 puntajes
        supera un umbral (500), retorna True; de lo contrario, False.
        """
        if len(self.episode_score_list) > 50000:
            if np.mean(self.episode_score_list[-100:]) > 500.0:
                return True
        return False

    def get_info(self):
        """
        Retorna información adicional del entorno.
        En este caso, no se utiliza información extra y se retorna None.
```

```
        """
        return None

    def render(self, mode='human'):
        """
        Método para visualizar el entorno.
        Actualmente no se implementa la visualización.
        """
        pass

    def apply_action(self, action):
        """
        Ejecuta la acción indicada modificando la velocidad de los motores de las
        ↪  ruedas.
        Las acciones definidas son:
          0: Avanzar recto.
          1: Giro suave a la izquierda.
          2: Giro suave a la derecha.
        """
        base_speed = 6.0  # Velocidad base del robot

        if action[0] == 0:  # Avanzar recto
            left_speed = base_speed
            right_speed = base_speed
        elif action[0] == 1:  # Giro suave a la izquierda
            left_speed = -base_speed
            right_speed = base_speed
        elif action[0] == 2:  # Giro suave a la derecha
            left_speed = base_speed
            right_speed = -base_speed

        # Configura los motores para operar sin límite de posición y asigna las
        ↪  velocidades
        self.right_motor.setPosition(float('inf'))
        self.left_motor.setPosition(float('inf'))
        self.left_motor.setVelocity(left_speed)
        self.right_motor.setVelocity(right_speed)

# ======================== BLOQUE PRINCIPAL ========================

# Instancia el entorno y crea el agente PPO utilizando las dimensiones definidas
env = TBot3()
agent = PPOAgent(number_of_inputs=env.observation_space.shape[0],
                 number_of_actor_outputs=env.action_space.n)

# Variables de control para el entrenamiento
```

```
solved = False
episode_count = 0
episode_limit = 5000

checkpoint_path = "forward_ppo"  # Nombre base para los archivos de checkpoint


# Verifica si existen ambos archivos de checkpoint: actor y critic
if os.path.exists(checkpoint_path + "_actor.pkl") and
↪   os.path.exists(checkpoint_path + "_critic.pkl"):
    agent.load(checkpoint_path)
    print("Checkpoint loaded from", checkpoint_path)
else:
    print("No checkpoint found. Training from scratch.")

# Bucle principal de entrenamiento: se ejecutan episodios hasta resolver la tarea
↪   o alcanzar el límite
while not solved and episode_count < episode_limit:
    observation = env.reset()         # Reinicia el entorno para comenzar un
    ↪   nuevo episodio
    env.episode_score = 0             # Reinicia la puntuación del episodio

    # Bucle interno que recorre cada paso dentro del episodio
    for step in range(env.steps_per_episode):
        # El agente selecciona una acción basándose en la observación actual
        selected_action, action_prob = agent.work(observation,
        ↪   type_="selectAction")
        new_observation, reward, done, info = env.step([selected_action])

        # Crea y almacena la transición para entrenamiento posterior
        trans = Transition(observation, selected_action, action_prob, reward,
        ↪   new_observation)
        agent.store_transition(trans)

        env.episode_score += reward  # Actualiza la puntuación acumulada del
        ↪   episodio
        observation = new_observation  # Actualiza la observación para el
        ↪   siguiente paso

        # Si se cumple la condición de término del episodio, finaliza el ciclo
        ↪   interno
        if done:
            env.episode_score_list.append(env.episode_score)
            agent.train_step(batch_size=step + 1)  # Ejecuta un paso de
            ↪   entrenamiento con la transición almacenada
```

```
            solved = env.solved()                    # Verifica si el desempeño
            ↪  cumple con la condición de solución
            break

    print(f"Episode #{episode_count} finished with score: {env.episode_score}
    ↪  FORWARD")
    episode_count += 1

    # Guarda el estado del agente cada 500 episodios
    if episode_count % 100 == 0:
        agent.save(checkpoint_path)
        print("Checkpoint saved at episode", episode_count)

# Mensaje de despliegue según si la tarea se resolvió o no durante el
↪  entrenamiento
if not solved:
    print("Task is not solved, deploying agent for testing...")
elif solved:
    print("Task is solved, deploying agent for testing...")

# Bucle infinito para desplegar el agente en modo prueba (testing)
observation = env.reset()
env.episode_score = 0.0

while True:
    # El agente selecciona la acción de máxima probabilidad para el despliegue
    selected_action, action_prob = agent.work(observation,
    ↪  type_="selectActionMax")
    observation, _, done, _ = env.step([selected_action])
    if done:
        observation = env.reset()
```

Controlador de ROS2

```
import rclpy
from rclpy.node import Node
from sensor_msgs.msg import LaserScan, Imu, NavSatFix
from nav_msgs.msg import Odometry
from geometry_msgs.msg import Twist
import numpy as np
import torch

class TurtleBot3Controller(Node):
    def __init__(self):
        super().__init__('turtlebot3_controller')
```

```
        # Inicialización de publicadores y suscriptores
        self.lidar_sub = self.create_subscription(LaserScan, '/scan',
        ↪  self.lidar_callback, 10)
        self.imu_sub = self.create_subscription(Imu, '/imu/data',
        ↪  self.imu_callback, 10)
        self.gps_sub = self.create_subscription(NavSatFix, '/gps/fix',
        ↪  self.gps_callback, 10)
        self.odom_sub = self.create_subscription(Odometry, '/odom',
        ↪  self.odom_callback, 10)
        self.cmd_vel_pub = self.create_publisher(Twist, '/cmd_vel', 10)

        # Variables de sensores
        self.lidar_data = np.zeros(360)
        self.robot_position = [0.0, 0.0]  # [x, y] del GPS
        self.robot_velocity = [0.0, 0.0]  # [vx, vy] de odometría
        self.robot_angular_velocity = 0.0  # Velocidad angular del robot

        # Carga de los modelos del Actor y el Crítico
        self.actor = torch.load("actor.pkl")
        self.critic = torch.load("critic.pkl")

    def lidar_callback(self, msg):
        self.lidar_data = np.nan_to_num(msg.ranges, nan=10, posinf=10,
        ↪  neginf=-10)

    def imu_callback(self, msg):
        # Extraer yaw únicamente
        quaternion = msg.orientation
        _, _, self.robot_yaw = self.extract_orientation(quaternion)

    def gps_callback(self, msg):
        # Extraer posición GPS
        self.robot_position = [msg.latitude, msg.longitude]

    def odom_callback(self, msg):
        # Extraer velocidades lineales y angular del robot
        self.robot_velocity = [msg.twist.twist.linear.x,
        ↪  msg.twist.twist.linear.y]
        self.robot_angular_velocity = msg.twist.twist.angular.z

    def extract_orientation(self, quaternion):
        # Convierte cuaterniones a roll, pitch, yaw
        import tf_transformations
        return tf_transformations.euler_from_quaternion([
            quaternion.x, quaternion.y, quaternion.z, quaternion.w
```

```
    ])

def get_observations(self):
    # Normalizar las posiciones x e y del GPS
    robot_x = normalize_to_range(self.robot_position[0], -4.5, 4.5, -1, 1)
    robot_y = normalize_to_range(self.robot_position[1], -3.0, 3.0, -1, 1)

    # Normalizar las velocidades lineales y angulares
    linear_velocity_x = normalize_to_range(self.robot_velocity[0], -0.26,
    ↪  0.26, -1, 1)
    linear_velocity_y = normalize_to_range(self.robot_velocity[1], -0.26,
    ↪  0.26, -1, 1)
    angular_velocity = normalize_to_range(self.robot_angular_velocity, -2.84,
    ↪  2.84, -1, 1)

    # Normalizar los datos del LiDAR
    lidar_normalized = np.vectorize(normalize_to_range)(self.lidar_data, 0,
    ↪  10, -1, 1)

    # Retornar las observaciones
    return np.array([robot_x, robot_y, linear_velocity_x, linear_velocity_y,
    ↪  angular_velocity, *lidar_normalized])

def select_action(self, observations):
    # Inferencia con el modelo Actor
    with torch.no_grad():
        observations_tensor = torch.tensor(observations,
        ↪  dtype=torch.float32).unsqueeze(0)
        action_probs = self.actor(observations_tensor)
        action = torch.argmax(action_probs).item()  # Selección determinista
        ↪  de acción
    return action

def apply_action(self, action):
    vel_msg = Twist()
    base_speed = 0.2  # Velocidad lineal base en m/s
    base_turn = 0.5   # Velocidad angular base en rad/s

    if action == 0:  # Avanzar recto
        vel_msg.linear.x = base_speed
        vel_msg.angular.z = 0.0
    elif action == 1:  # Giro en el lugar hacia la izquierda
        vel_msg.linear.x = 0.0  # No avanzar mientras gira
        vel_msg.angular.z = base_turn  # Giro positivo
    elif action == 2:  # Giro en el lugar hacia la derecha
        vel_msg.linear.x = 0.0  # No avanzar mientras gira
```

```
            vel_msg.angular.z = -base_turn  # Giro negativo

        self.cmd_vel_pub.publish(vel_msg)

    def run(self):
        while rclpy.ok():
            rclpy.spin_once(self)  # Procesa callbacks
            observations = self.get_observations()
            action = self.select_action(observations)
            self.apply_action(action)

def normalize_to_range(value, min_val, max_val, new_min, new_max):
    return (value - min_val) / (max_val - min_val) * (new_max - new_min) +
    ↪  new_min

def main(args=None):
    rclpy.init(args=args)
    controller = TurtleBot3Controller()
    try:
        controller.run()
    except KeyboardInterrupt:
        controller.get_logger().info('Shutting down...')
        controller.destroy_node()
        rclpy.shutdown()

if __name__ == '__main__':
    main()
```

References

[1] *Kirtas, M., Tsampazis, K., Passalis, N., and Tefas, A. (2020). Deepbots: A webots-based deep reinforcement learning framework for robotics. In* 16th IFIP International Conference on Artificial Intelligence Applications and Innovations (AIAI)*, pages 64–75.*

[2] *Ltd., C. (2024). Webots: Open-source mobile robot simulation software. Open-source mobile robot simulation software.*

[3] *Open Robotics (2022). Robot Operating System 2 (ROS 2) Humble Hawksbill.*

[4] *ROBOTIS (2025).* TurtleBot3 e-Manual.

[5] *Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms.* arXiv preprint arXiv:1707.06347*.*