Desarrollo de herramientas de programación y simulación para los agentes robóticos Pololu 3Pi+ dentro del ecosistema Robotat

Jonathan Emanuel Pu Aguilera

UVG UNIVERSIDAD DEL VALLE DE GUATEMALA

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Desarrollo de herramientas de programación y simulación para los agentes robóticos Pololu 3Pi+ dentro del ecosistema Robotat

Trabajo de graduación presentado por Jonathan Emanuel Pu Aguilera
para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

2024

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Desarrollo de herramientas de programación y simulación para los agentes robóticos Pololu 3Pi+ dentro del ecosistema Robotat

Trabajo de graduación presentado por Jonathan Emanuel Pu Aguilera
para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

2024

Vo.Bo.:

(f) _____

MSc. Miguel Enrique Zea

Tribunal Examinador:

(f) _____

MSc. Miguel Enrique Zea

(f) _____

Ing. Christopher Chiroy

(f) _____

Ing. Jonathan Mansilla

Fecha de aprobación: Guatemala, 20 de enero de 2024.

Prefacio

Quiero agradecer a toda mi familia por todo el apoyo brindado durante todo mi trayecto académico, en especial durante estos años de Universidad. En especial quiero agradecer a mis padres Byron Rolando Pu Lacán y María de los Ángeles Aguilera de Pu por su guía, sacrificio y esfuerzo por sacarme adelante, entre tantas cosas más que les debería agradecer. A mi hermana Allyson Pu por su ejemplo de dedicación en su vida académica y profesional y por su apoyo incondicional. Sin la ayuda de Dios y de mi familia, nada de esto sería posible. Gracias a todos ellos por creer en mí. Este logro alcanzado es de todos.

También quiero agradecer a mis amigos de carrera, por todo su apoyo de diferentes maneras, ellos fueron parte fundamental de mi desarrollo académico y personal, gracias por ese aliento mutuo en los momentos que más difícil se tornaba la situación y por esos consejos técnicos también. También esos amigos de años, que siempre estuvieron apoyándome y tendiéndome una mano de ánimos. También gracias a mis mentores dentro y fuera de la UVG, y a aquellos catedráticos que siempre me extendieron la mano. Del mismo modo agradezco a mi asesor y catedrático Miguel Zea, ya que siempre estuvo dispuesto a apoyarnos no solo en el proyecto de graduación, sino a compartir su conocimiento y recursos durante la carrera.

Índice

Lista de figuras

Resumen

Las plataformas de experimentación robótica desempeñan un papel fundamental en la validación de pruebas para agentes robóticos. En este contexto, el ecosistema del Robotat de UVG ha facilitado la continua integración de herramientas experimentales. El presente trabajo se enfoca en el desarrollo de herramientas de software diseñadas para la simulación, programación y monitoreo de los agentes robóticos de dos ruedas modelo Pololu 3Pi+ en la mesa de pruebas del Robotat.

El objetivo general consistió en crear herramientas que permitieran la simulación y programación de diversas aplicaciones en los agentes robóticos Pololu 3Pi+ dentro del entorno del Robotat de la UVG. Para lograrlo, se utilizó el lenguaje de programación Python y se estableció la infraestructura necesaria para ejecutar simulaciones en una ventana de animación mediante la librería Pygame. Esto permitió visualizar múltiples agentes ejecutando distintos controladores. Además, se aprovecharon las capacidades de actualización vía Wi-Fi de los microcontroladores ESP32, utilizados por los agentes robóticos, en las tareas de programación. Para ello, se empleó el entorno de desarrollo PlatformIO y se integró con la infraestructura de software de Python para automatizar el proceso de carga. Finalmente, se realizó una adaptación de funciones previamente existentes en MATLAB a Python para monitorear la pose de los robots en la mesa de pruebas del Robotat, utilizando el sistema de captura OptiTrack.

Abstract

Robotic experimentation platforms play a crucial role in validating tests for robotic agents. In this context, the UVG Robotat ecosystem has facilitated the continuous integration of experimental tools. This work focuses on the development of software tools designed for the simulation, programming, and monitoring of two-wheeled robotic agents, specifically the Pololu 3Pi+, on the Robotat test bench.

The main objective was to create tools that would enable the simulation and programming of various applications on Pololu 3Pi+ robotic agents within the UVG Robotat environment. To achieve this, the Python programming language was used, and the necessary infrastructure was established to execute simulations in an animation window using the Pygame library. This allowed for the visualization of multiple agents running different controllers. Additionally, the WiFi update capabilities of the ESP32 microcontrollers, used by the robotic agents, were leveraged in the programming tasks. For this purpose, the PlatformIO development environment was employed and integrated with the Python software infrastructure to automate the uploading process. Finally, a conversion of previously existing MATLAB functions to Python was carried out to monitor the pose of the robots on the Robotat test bench, utilizing the OptiTrack motion capture system.

CAPÍTULO 1

Introducción

La experimentación robótica en mesas de pruebas es algo que ha tomado bastante relevancia para implementar buenas prácticas en el desarrollo de robots móviles. Ejemplo de esto son los exitosos casos del ecosistema de robótica de la Universidad Georgia Tech [1], la plataforma DOTS [2], la plataforma de la Universidad de West Virginia [3], entre otros. En el caso del Robotat de la UVG, que es el ecosistema de robótica que se ha desarrollado en los últimos años, ha permitido una versatilidad en las pruebas que se pueden realizar. Sin embargo, ya que es un ecosistema en constante desarrollo, da la posibilidad a la integración de nuevas herramientas que agreguen funciones útiles al proceso de experimentación robótica.

La UVG cuenta con diversos agentes robóticos: manipuladores seriales, móviles de dos ruedas, drones, entre otros. Se decidió comenzar con los robots móviles de dos ruedas para agregar nueva infraestructura que facilite la experimentación y análisis de los mismos resultados. Considerando que es uno de los robots que tiene más campo para aprovechar el sistema de captura OptiTrack con el que cuenta la mesa de pruebas, se plantearon cuatro objetivos para estas herramientas que se desarrollaron: desarrollar una interfaz que permita la simulación de varios agentes robóticos Pololu 3Pi+ dentro de la plataforma de pruebas del Robotat, implementar una infraestructura de software que aproveche las capacidades de recibir actualizaciones remotas de los ESP32 que utilizan los robots, y monitorear y generar información útil a partir de la información dada por el sistema de captura instalado en la mesa de pruebas. Finalmente, se planeó abordar el sistema operativo ROS2 en los agentes robóticos Pololu 3Pi+.

Este documento se organiza de la siguiente manera: se comienza presentando un capítulo de requisitos que cada función debía cumplir, y así tener estos de guía antes de realizar la programación. Seguido por un capítulo que describe los módulos creados para la herramienta de simulación y las decisiones de librerías de Python que se usaron para lograr una animación de los agentes robóticos, aquí también se describen las decisiones y estructura general de la interfaz, y la integración del simulador en la misma. Posteriormente en cada capítulo siguiente solamente se aborda la integración de la herramienta específica dentro de la interfaz y sus funciones básicas para cada pestaña.

En el siguiente capítulo se describe el proceso de validación de las actualizaciones WiFi de los ESP32. Se muestran las configuraciones a seguir para el proceso exitoso de estas actualizaciones y las alternativas de entorno evaluadas. Se muestran los resultados que se deben obtener al lograr una conexión y actualización de firmware exitosa en los ESP32 asignados en la red del Robotat. Para finalmente proponer la estructura que debe seguir el código de los ESP32 para el funcionamiento de las actualizaciones OTA y un control punto a punto ejecutado desde el microcontrolador.

Finalmente, se describe la adaptación que se hizo de funciones preexistentes en MATLAB a Python para obtener información del sistema de captura OptiTrack. Se muestra el proceso para desplegar la información de la pose de los robots en una ventana de animación y se describe el proceso para generar archivos CSV de esta información como reportes.

El desarrollo de herramientas de comunicación para los agentes robóticos Pololu 3Pi+ utilizando ROS2 se aborda principalmente en la fase de investigación en este trabajo. En la Sección 6.4 se presentan los conceptos fundamentales de ROS2, así como otras consideraciones importantes acerca de sus requisitos mínimos. En el último capítulo del trabajo se presenta la ejecución de una transacción de información entre un microcontrolador con micro-ROS y un ordenador con ROS2. Con estos conceptos base, finalmente se presenta una propuesta inicial para la implementación de estas herramientas en el ecosistema del Robotat.

CAPÍTULO 2

Antecedentes

En años anteriores, se inició con el desarrollo de un ecosistema robótico en UVG, denominado Robotat. Este ecosistema cuenta con una variedad de agentes robóticos, como robots móviles sobre ruedas y manipuladores seriales, además de ofrecer una infraestructura en hardware y software que da la capacidad de interactuar con los agentes y realizar experimentos con fines de investigación. Anteriormente, Perafán describe en su tesis la implementación del sistema de captura con las cámaras OptiTrack, una red de comunicación WiFi dentro del ecosistema, y el desarrollo de una antena inteligente para permitir la interacción de agentes no robóticos en el Robotat [4]. Actualmente, sin embargo, la red de comunicación emplea el protocolo TCP/IP, el cuál es uno de los cambios más signficativos del protocolo MQTT implementado en [4]. Además, el ecosistema ha integrado principalmente agentes robóticos de dos ruedas de modelo Pololu 3Pi+, junto con un servidor que permite intercambio de información entre microcontroladores ESP32 que se integraron a los Pololu. Existen herramientas de software que permiten la obtención de la pose de los marcadores del OptiTrack desde diferentes medios, incluidos MATLAB y el ESP32.

Para tener una mejor idea de lo que se quiere alcanzar con el ecosistema Robotat en la Universidad del Valle de Guatemala, y como referencia, se tiene el ecosistema de robótica de Georgia Tech, el Robotarium [1]. El Robotarium es una plataforma dedicada a investigación de robótica de enjambres principalmente, abierta a todo público de manera remota. Cuenta con un simulador disponible para instalar y ejecutar con MATLAB o Python, y se integra con el laboratorio de robótica de Georgia Tech para que no solo se realice la simulación, sino que después de desarrollado el experimento pueda ser aprobado por la administración del Robotarium y ejecutarse en los agentes reales, obteniendo información útil de los agentes robóticos después del experimento.

Existen también otras plataformas de experimentación robótica, con características similares al Robotarium. En [5] se propone un acercamiento utilizando robots construidos con los kits de Lego Mindstorm para realizar simulaciones y experimentos remotos. [6] propone un simulador utilizando LabView, MATLAB e Easy Java Simulations junto con la operación de agentes reales para su posterior análisis de datos. Por último, se puede mencionar también

Figura 1: Robotarium de Georgia Tech [1].

a SyRoTek [7], el cual también provee acceso a robots móbiles en una arena con obstáculos reconfigurables, permitiendo variedad de experimentos, donde se pueden desarrollar algoritmos propios y experimentar el control de los robots y analizar los datos capturados por las interfaces disponibles.

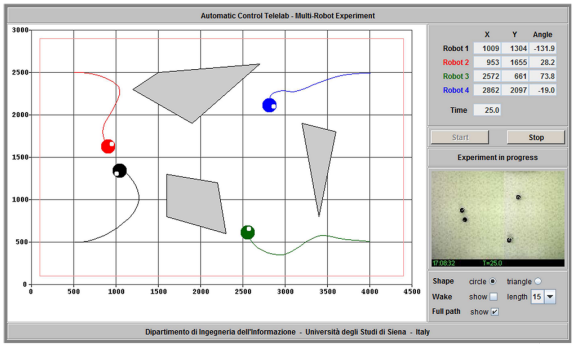

Figura 2: Sistema de simulación y captura de video para los Lego Mindstorm [5].

Actualmente las demás plataformas que podrían compararse, en cierto sentido, al Robotarium de Georgia Tech, son únicamente simuladores basados en la nube que permiten simular trayectorias, tareas y experimentos con una amplia variedad de agentes robóticos que se encuentran dentro de su sistema, ofreciendo variedad de librerías y sistemas enfocados a robótica. En [8] se proveen ejemplos de laboratorios remotos en diversas disciplinas académicas, como los iLab del Instituto Tecnológico de Massachusetts [9], los laboratorios remotos de la Universidad de Hagen [10] y del Instituto Tecnológico de Blekinge [11]. Estos son ejemplos de simuladores no robóticos para el aprendizaje eléctronico, principalmente de microelectrónica, que ofrecen diseñar experimentos controlados de circuitos, o aprendizaje básico de la materia. El laboratorio remoto más similar al Robotarium sería el laboratorio remoto de la Universidad de Deutso, WebLabs, el cual entre sus opciones tiene el programar un robot con Arduino de manera remota y ejecutar los experimentos en el robot real [12]. Hay otras plataformas de robótica de enjambres que se han desarrollado con objetivos similares al Robotat. La Universidad de West Virginia junto con su departamento de investigación de pregrado implementó una mesa de pruebas para robótica de enjambres, utilizando también cámaras OptiTrack y desarrollando sus propias plataformas interactivas para las pruebas de los agentes robóticos [3]. Puede visualizarse la configuración de la plataforma en la Figura 4.

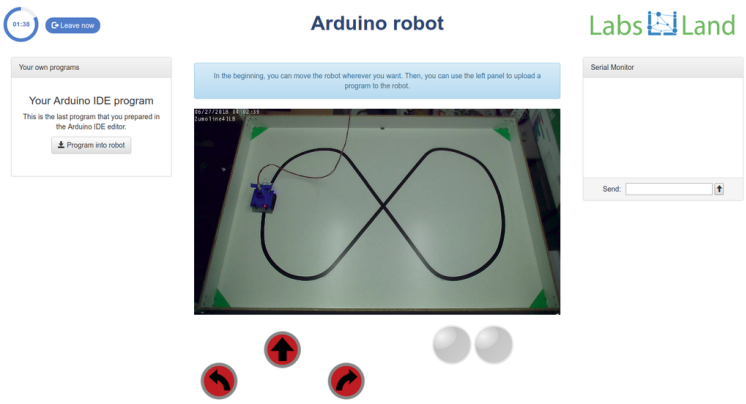

Figura 3: LabsLand de la Universidad Deutso [12].

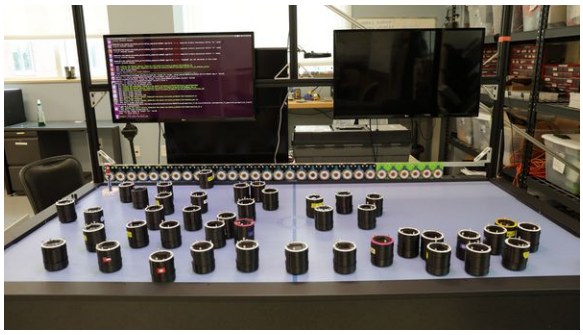

Figura 4: Plataforma robótica de enjambres Universidad West Virginia [3].

El instituto Wyss de la Universidad de Harvard también ha desarrollado mesas de prueba para los Kylobots, llegando a realizar pruebas con 1024 de estos agentes en configuración de robótica de enjambres [13]. Por otro lado la plataforma DOTS ofrece una mesa de pruebas abierto para pruebas de agentes robóticos en enjambres en ambiente industrial, diferenciándose de los mencionados antes que se limitan a investigación académica. Este cuenta con 20 agentes, sistemas de captura por cámara, red de comunicación 5G privada, entre otras características [2]. En la Figura 5 se puede ver la estructura de experimentación implementada en esta plataforma.

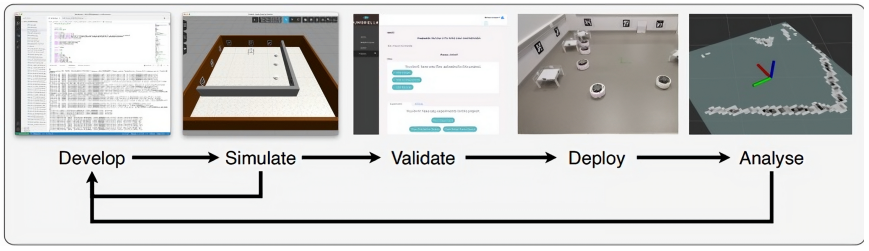

Figura 5: Sistema propuesto en la plataforma DOTS [2].

Vale la pena mencionar también al Smart Lab de la Universidad Purdue, el cual ha desarrollado investigaciones de robots múltiples que realizan tareas en conjunto, ejecutándose en simulaciones y sistemas que permiten la interacción de los agentes en ambientes de la vida real, con el objetivo de implementarlos en misiones de búsqueda y rescate, así como

en monitoreo ambiental [14]. En la Figura 6 se puede ver algunos de los ambientes que han usado para probar los robots en conjunto que, aunque no es una mesa de pruebas como las otras plataformas mencionadas, igual implementa protocolos de comunicación y localización de los agentes para desarrollar este concepto de enjambres de robots.

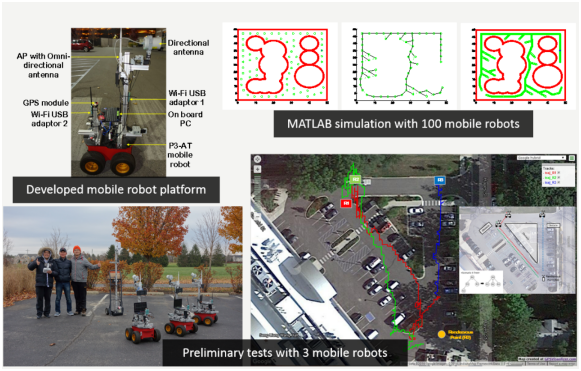

Figura 6: Pruebas desarrolladas por la Universidad Purdue, SmartLab [14].

CAPÍTULO 3

Justificación

El ecosistema Robotat en UVG ha experimentado notables avances con el desarrollo de herramientas específicas para sus diversos agentes robóticos, como manipuladores seriales, drones y robots móviles con ruedas. Este trabajo se enfocó especialmente en crear herramientas para los agentes Pololu 3Pi+.

La introducción de herramientas de simulación y programación en el Robotat ha demostrado ser esencial en el proceso de experimentación. Como se detalla en los antecedentes, la posibilidad de diseñar y probar experimentos a un nivel virtual antes de llevarlos a cabo en el mundo real ha demostrado ser una práctica muy efectiva en otras plataformas de experimentación robótica. La incorporación de esta funcionalidad en el Robotat ha simplificado significativamente la realización de experimentos y el análisis de los datos obtenidos.

La actualización de la red de comunicación del Robotat de un protocolo MQTT a TCP/IP, junto con las mejoras en hardware y software de los Pololu 3Pi+ mediante la inclusión de microcontroladores ESP32, ha proporcionado una base sólida para la programación y control simultáneo de múltiples agentes.

Las herramientas de simulación, programación y monitoreo, cuya implementación se detalla en el resumen, desempeñan un papel esencial en el avance y desarrollo continuo del ecosistema Robotat. La capacidad de simular situaciones de experimentación antes de llevarlas a cabo en la realidad, como se logra con las herramientas de simulación, crea un entorno controlado y seguro para probar algoritmos y controladores, lo que reduce la posibilidad de fallos en pruebas físicas y acelera el proceso de mejora en el diseño de algoritmos.

Además, las herramientas de programación diseñadas para aprovechar la actualización vía WiFi de los ESP32 permiten una programación más eficaz y flexible de los agentes Pololu 3Pi+. Esta capacidad de programación remota no solo ahorra tiempo, sino que también facilita la implementación y prueba de múltiples algoritmos simultáneamente, lo que conduce a una mayor agilidad en el proceso de desarrollo.

Por último, las herramientas de monitoreo, al integrar el sistema de captura OptiTrack,

ofrecen información detallada sobre el comportamiento de los robots en el entorno del Robotat, dando posibilidades a un mejor análisis de resultados y agregar información útil a la experimentación con los agentes robóticos.

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Desarrollar herramientas que permitan la simulación y programación de aplicaciones varias en los agentes robóticos Pololu 3Pi+ dentro del ecosistema Robotat.

4.2. Objetivos específicos

- Desarrollar una interfaz de usuario amigable que permita realizar simulaciones de uno o más agentes Pololu 3Pi+ dentro de la plataforma del ecosistema Robotat.
- Implementar la infraestructura de software para poder programar y monitorear los Pololu 3Pi+ de forma remota, desde el simulador.
- Generar reportes útiles y robustos con la información obtenida durante los experimentos realizados con los agentes robóticos.
- Abordar la implementación de un paquete que permita la comunicación con los agentes Pololu 3Pi+ dentro del sistema operativo ROS2.

CAPÍTULO 5

Alcance

Este trabajo de graduación se centró en el uso y desarrollo del ecosistema Robotat de la UVG, tomando como punto de partida la tesis de Camilo Perafán. El objetivo principal fue ampliar las capacidades de esta plataforma mediante la creación y estructuración de herramientas de software que facilitaran la experimentación.

El proyecto se centró en tres funciones fundamentales: simulación, programación y monitoreo, todas orientadas a agilizar la experimentación con los componentes del Robotat, incluyendo los Pololu 3Pi+, los microcontroladores ESP32, el sistema de captura OptiTrack y la red WiFi asociada.

El alcance de este trabajo se vio limitado por el tiempo de uso del espacio del Robotat, ya que al estar en un espacio de laboratorio de uso general para clases de la Universidad, el horario de trabajo era escaso. Las herramientas de simulación se limitaron a estructurar los objetos principales de forma modular, y a definir la forma de crear los escenarios de simulación para que en un futuro se puedan agregar más funcionalidades. Ya que el objetivo de las herramientas de simulación no era implementar un algoritmo específico de control, se realizaron pruebas con controladores de punto a punto para validar el funcionamiento de las mismas. Además, al estar implementando nuevas herramientas para los agentes Pololu 3Pi+ y sus microcontroladores, la infraestructura de software se vio limitada por las capacidades existentes de los agentes. Para las funciones de programación vía WiFi de los ESP32 se utilizaron librerías preexistentes del entorno Arduino y PlatformIO. El monitoreo de datos de los agentes robóticos se vio limitado a la información que provee el servidor del Robotat al momento de realizar este trabajo, siendo la pose de los marcadores, la información útil que se pudo obtener.

Por la coyuntura nacional de octubre de 2023, el espacio de trabajo del Robotat se vio aún más restringido, lo que incurrió en atrasos en el desarrollo y validación de las herramientas. Esto instó a que el enfoque del trabajo fuera el desarrollo del simulador (simulación, programación y monitoreo), y que el cuarto objetivo (relacionado a ROS2) se limitará a una fase de investigación, estableciendo un punto de partida para futuras integraciones de estas herramientas a los Pololu 3Pi+.

CAPÍTULO 6

Marco teórico

6.1. Pololu 3Pi+

Los Pololu 3Pi+ son robots de alto rendimiento, de fácil programación, con 9.7 cm de diámetro de tamaño. Están equipados con un microcontrolador AVR ATmega32U4, compuesto de sensores variados para desenvolverse en experimentos con diversas condiciones. En la Figura 7 se pueden ver algunos de los componentes principales con los que cuenta. Existen diferentes versiones de estos robots con respecto a los motores que incluye, la Standard Edition, Turtle Edition y la Hyper Edition. En el ecosistema Robotat se cuenta con la Standard Edition, que según el manual de usuario provisto por los fabricantes tiene una buena combinación de velocidad y controlabilidad.

La placa del Pololu 3Pi+ incluye una conexión USB Micro-B que permite establecer el intercambio de datos con una computadora y así programar la placa de manera sencilla. Esta conexión por cable USB también puede proveer energía a los componentes del robot exceptuando los motores. El ATmega32U4 está también precargado con un bootloader compatible con Arduino para ser capaz de programarse usando el Arduino IDE, también está la opción de usar un programador externo por medio de los headers de 6 pines SPI. [15]

6.2. ESP32

El ESP32 es un microcontrolador que incluye un módulo Wi-Fi y Bluetooth, siendo capaz de desempeñarse como un sistema individual o como esclavo de otro microcontrolador maestro, sirviendo como interfaz para brindarle conectividad Wi-Fi y Bluetooth a través de sus capacidades de comunicación [16]. Es adecuado para aplicaciones móviles, IoT, dispositivos portátiles, entre otros, al alcanzar un consumo de energía ultra bajo. La variante de ESP32 que se está implementando en los Pololu 3Pi+ es el ESP32-WROOM-32E, cuyas especificaciones más relevantes para la aplicación son :

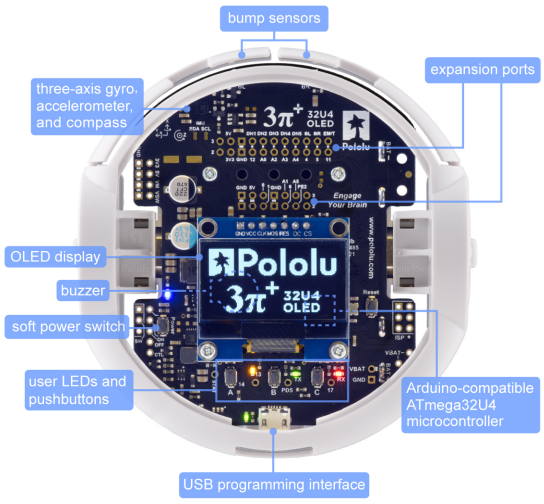

Figura 7: Componentes principales de los Pololu 3Pi+ [15].

- Rango de frecuencia central del canal operativo de 2.4 GHz 2.5 GHz.
- Estándar de Wi-Fi IEEE 802.11b/g/n.
- Protocolos de red IPv4, TCP/UDP/HTTP/FTP.

6.2.1. Protocolo TCP/IP

Un protocolo es un conjunto de normas que dictan la forma en que diferentes unidades de software van a establecer una comunicación. En estos se define la forma en que se entrega la información, cómo se envía para que llegue a su destino y la vía que va a utilizar. También se pueden coordinar mensajes y notificaciones de recepción de mensaje. En la Figura 8 se observan las capas del protocolo TCP/IP. La capa de transporte de Internet puede ser UDP (User Datagram Procol) o TCP (Transmission Control Protocol). Estos son los protocolos básicos en esta capa para conectar a los sistemas principales de Internet entre sí [17] [18]. Cada uno de estos dividen los datos en paquetes después de recibir los datos de la aplicación, añaden la dirección de destino y pasan los paquetes a la capa de red de Internet.

La capa de red de Internet coloca el paquete en un datagrama (paquete de transferencia) de IP (Internet Protocol), termina de poner información útil como una cabecera y pasa el datagrama a la interfaz de red. La interfaz de red es la que recibe el paquete IP y lo transmite a través de una red de hardware como una red Ethernet, red de anillo, entre otras [18]. En la Figura 9 se describe el proceso de transmisión y recepción de datos en ambas vías.

El ecosistema Robotat implementa una capa de transporte TCP para una comunicación confiable, y UDP para velocidad. TCP se caracteriza para la fiablidad de entrega de datos, el protocolo se asegura que los datos no se dañen, pierdan, dupliquen, o se entreguen desordenados en el receptor. TCP permite que en un solo sistema se usen recursos de la comunicación en simultáneo, combina el número de puerto con la dirección de red y la del sistema principal e identifica de forma exclusiva cada socket de conexión. Además, es capaz

de mantener la conexión entre un sistema y otro [19]. El protocolo UDP, por otro lado, no garantiza la entrega de datagrama ni protege contra la duplicación de datos. Tampoco establece una conexión entre sistemas y la entrega de datos puede ser desordenada [20] [21].

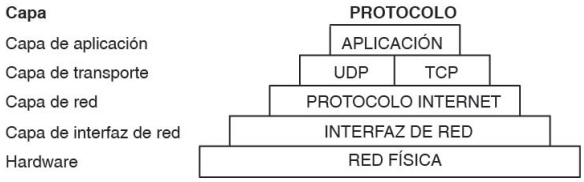

Figura 8: Conjunto de protocolos TCP/IP [18].

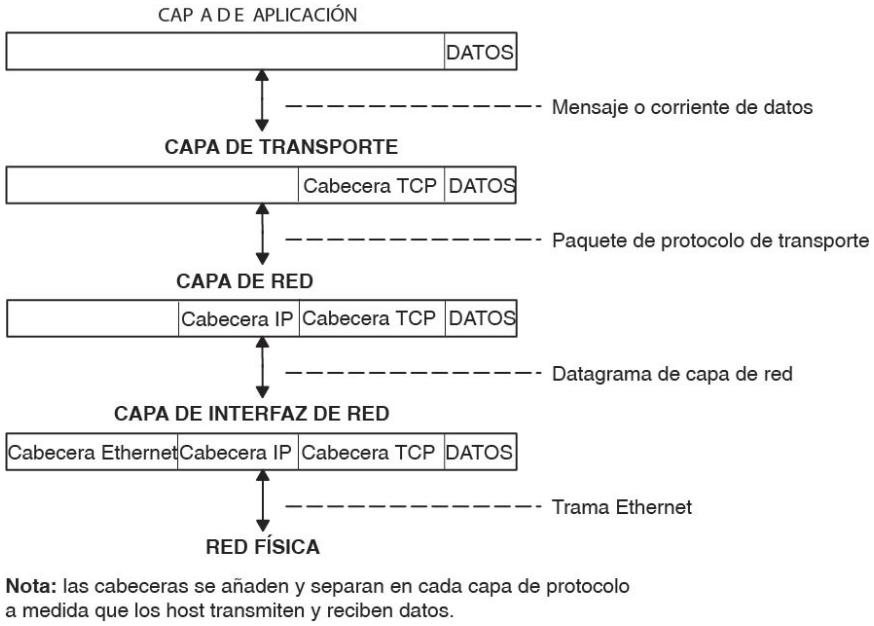

Figura 9: Transmisiones y recepciones de datos de sistema [18].

6.2.2. Over-the-air programming

Over-the-air (OTA) se refiere a distribuir información de forma inalámbrica, el término se usa en muchos contextos de dispositivos, especialmente en IoT. El término completo suele encontrarse como over-the-air update/programming, donde se refiere a actualizaciones en los sistemas que ocurren de manera remota, sin conexión física con el dispositivo final. En el contexto de dispositivos, permite actualizar de forma constante aspectos de seguridad, avances significativos del producto, entre otros. La posibilidad de programar o actualizar cierto sistema, trae de ventaja la eficiencia y practicidad en estas tareas, en vez de tener que conectarse físicamente a cada dispositivo o tecnología en general que implemente este método de programación.

Algo importante a considerar con OTA es la robustez de la red por la que se transmitirá la información. Las especificaciones técnicas de esta red va a regir sobre aspectos como velocidad, confiabilidad y seguridad de la transmisión [22].

6.2.3. OTA en ESP32

El ESP32 tiene soporte para actualizaciones OTA, aprovechando la función de Wi-Fi del microcontrolador, que le permite conectarse a una red para cumplir con esta tarea. Este requiere configurar una tabla de partición con al menos dos ranuras o accesos, para que luego las funciones de operación OTA en el ESP32 reciban nuevo firmware y lo escriban en cualquiera de las dos ranuras que no esté siendo utilizada en otra operación de carga. En [23] se dan más detalles del proceso a seguir para crear las particiones de forma correcta. Una característica crítica para OTA en ESP32 es la reversión de la aplicación, esto permite que el dispositivo siga trabajando de forma correcta aunque la nueva carga tenga errores, al regresar a la aplicación previa a la actualización.

6.3. Robots móviles con ruedas

Entre los tipos de agentes robóticos de base móvil, los robots móviles con ruedas pueden ser definidos y controlados a partir de su cinemática sin tener que recurrir a técnicas de dinámica de cuerpos rígidos. Los robots móviles con ruedas se dividen en **unidireccionales** y **no holonómicos**. La diferencia entre estos es que la restricción de velocidad en los no holonómicos no puede ser integrada en una restricción de configuración que forme una relación entre sus posiciones y ángulos. El movimiento de estos robots no holonómicos va a estar definido por restricciones de rodamiento y no deslizamiento. Este último tipo va a ser el enfoque de la teoría en cuánto a robots móviles con ruedas ya que los Pololu 3Pi+ entran bajo esta categoría por el tipo de ruedas que utiliza, convencionales cuya dirección se determina con un ángulo de giro y tiene las restricciones mencionadas [24].

6.3.1. Modelado de robots móviles con ruedas

El robot móvil más sencillo es una rueda individual giratoria de radio r, que se le denomina *uniciclo*. La configuración del uniciclo está dada por $q = (\phi, x, y, \theta)$, donde (x, y) es el punto de contacto, ϕ es la orientación, y θ es el ángulo de balanceo de la rueda. Usualmente el ángulo de balanceo de la rueda no es relevante para el sistema de control, por lo que las ecuaciones cinemáticas de movimiento quedan expresadas como (1) [24].

$$\begin{aligned} \dot{x} &= u_1 r \cos\phi, \\ \dot{y} &= u_1 r \sin\phi, \\ \dot{\phi} &= u_2. \end{aligned} \tag{1}$$

Al pasar a una arquitectura más completa de robots móviles con ruedas, se puede definir el robot diferencial. Un robot de accionamiento diferencial se compone de dos ruedas de radio r independientes, rotando sobre el mismo eje, y una rueda rueda giratoria, o de fricción baja en general, solamente para mantener el robot horizontal. Se define la configuración como $q = (\phi, x, y, \theta_R, \theta_L)$, en la cual los últimos dos términos corresponden a los ángulos de balanceo de la rueda derecha e izquierda respectivamente. Nuevamente, estos ángulos no

suelen ser de interés en el control, por lo que se omiten llegando al modelo en (2) [24].

$$\dot{q} = \frac{r}{2} \begin{bmatrix} \cos\phi & \cos\phi \\ \sin\phi & \sin\phi \\ \frac{1}{l} & -\frac{1}{l} \end{bmatrix} \begin{bmatrix} u_L \\ u_R \end{bmatrix}. \tag{2}$$

En el modelo, u_R es la velocidad angular de la rueda izquierda, u_L la velocidad de la derecha y l es la distancia del centro del robot a una de las ruedas.

6.3.2. Control de robots móviles con ruedas

Para el control de robots móviles con ruedas se suele utilizar una arquitectura jerárquica de control, en el cuál cada capa de control entrega referencias a la capa siguiente y más baja. La primera suele ser un sistema dinámico simple pero que es representativo del modelo completo, en este caso el uniciclo. Se van a denominar estas referencias como v_{ref} y w_{ref}. La capa que recibe estas referencias entonces es la del sistema dinámico completo con actuadores ideales, siendo este el modelo cinemático del robot móvil, dando referencia $\phi_{R,ref}$ y $\phi_{L,ref}$ a la última capa, que en el caso de los robots móviles con ruedas, son los motores con las ruedas unidas a ellos y estos ya se pueden controlar mediante PIDs de velocidad [25].

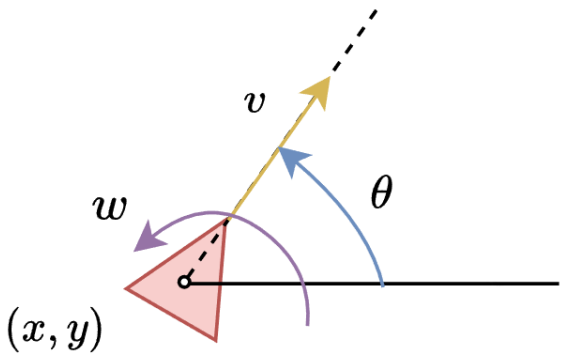

Figura 10: Modelado del uniciclo para control

El modelo del uniclo, partiendo de la Figura 10 queda definido como (3). Con este modelo, ya se tienen disponibles técnicas de control PID o enfoque de control moderno. Una vez mapeado el modelo del robot al uniciclo, se controla con alguno de los métodos mencionados y se encuentran las velocidades de control v_{ctrl} y w_{ctrl}. Finalmente, se mapean las velocidades de control al robot real como se muestra en (4) [25].

$$\begin{aligned} \dot{x} &= v\cos\theta, \\ \dot{y} &= v\sin\theta, \\ \dot{\theta} &= w. \end{aligned} \tag{3}$$

$$\begin{aligned} \dot{\phi}_{R,ctrl} &= \frac{v_{ctrl} + l w_{ctrl}}{r}, \\ \dot{\phi}_{L,ctrl} &= \frac{v_{ctrl} - l w_{ctrl}}{r}. \end{aligned} \tag{4}$$

6.4. ROS2

El Robot Operating System (ROS) es un conjunto de librerías y paquetes dedicado al desarrollo de aplicaciones robóticas, de código abierto. ROS se lanzó en 2007 y desde entonces se ha actualizado regularmente, creciendo en soporte y cada vez con más proyectos implementados por la comunidad [26]. Uno de los últimos lanzamientos fue ROS2, del que existen diferentes distribuciones y de las cuáles las que tienen soporte en 2023 son la Humble Hawksbill, Foxy Fitzroy, y más recientemente Iron Irwini [27]. Actualmente ROS2 es la versión del sistema operativo oficial, lo que implica que todas las versiones de ROS1 pronto dejaran de tener actualizaciones y soporte oficial de Open Robotics. Algunas de las herramientas que ofrece ROS son para propósito de desarrollo y simulación, como visualizaciones de robots, animaciones, gráficos; y otros son para desarrollo de aplicaciones para el núcleo de un agente robótico, como librerías de manejo de motores, drivers, entre otros. A pesar de que las siglas de ROS indican que es un sistema operativo, pero al ser en realidad un set de librerías y paquetes para el desarrollo de aplicaciones robóticas, requiere de un sistema operativo anfitrión, y Ubuntu Linux es el sistema de preferencia. ROS2 puede ser instalado en una computadora que funcione con ese sistema operativo, y pensando en nodos que se ejecuten desde un agente robótico, se puede hacer uso de Ubuntu 20.04 LTS Server OS para que se ejecute ROS2 en un dispositivo más portátil como una Raspberry Pi.

ROS2 aplica un mecanismo de publicación/suscripción, funcionando como software intermedio en las aplicaciones robóticas. Los conceptos más importantes a conocer del núcleo del funcionamiento de ROS2 (y ROS en general) son nodos, tópicos, interfaces, servicios, acciones y parámetros [28]. El componente base que se encuentra en cualquier distribución de ROS es el "gráfico de ROS", este hace referencia a los nodos y los demás componentes del sistema, y establece la forma en que se comunican entre sí [27]

6.4.1. Nodos

Los nodos son los procesos más básicos, deben ser diseñados para que tengan un propósito único, y sencillo, esto facilita la implementación de la modularidad que ofrece ROS. Ejemplos de actividades que puede realizar un nodo son, el control de los motores de una rueda, el manejo de un sensor, localización, planificar una ruta, pero no varias de estas actividades a la vez. Un nodo se puede comunicar con otro a través de tópicos, servicios, acciones, o un servidor de parámetros [29].

6.4.2. Interfaces

Son los medios de comunicación para las aplicaciones desarrolladas en ROS. Existen tres tipos de interfaces en ROS: tópicos, servicios y acciones. Un elemento clave de la forma en que se comunican las aplicaciones en ROS es el uso de un lenguaje de descripción simplificado, el lenguaje de definición de interfaz (IDL por sus siglas en inglés) [30]. Este IDL describe un lenguaje que permite la programación de objetos en un lenguaje que da la versatilidad a comunicarse con otros módulos que no están escritos necesariamente en el mismo lenguaje [31].

Tópicos

Estos son especialmente para flujos de data continuos. Ya que ROS es un sistema de tipo publicación/suscriptor en el cual el editor produce data y el suscriptor la adquiere. La forma en que ambas partes pueden intercambiar esta información es a través de los tópicos. Pueden haber desde cero a más editores y desde cero a más suscriptores en un mismo tópico. En el caso que haya un tópico con varios suscriptores, cuando el editor emita un mensaje todos los suscriptores en el sistema van a recibir la data. En la Figura 11 se puede ver un ejemplo de un tópico siendo el intermediador entre un nodo que es editor y otro nodo que actúa como el suscriptor [32].

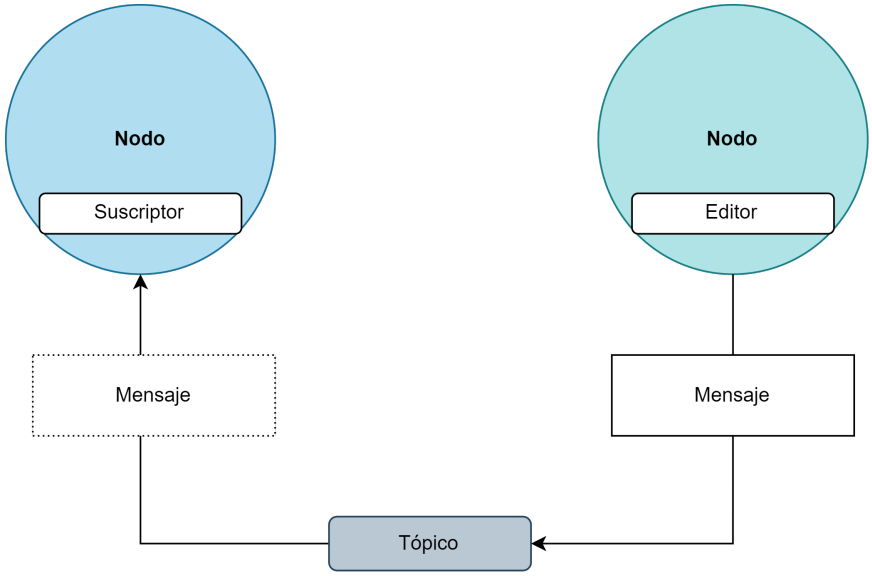

Figura 11: Diagrama de tópicos en ROS2 con un único suscriptor y editor en el tópico. [32]

Servicios

Los servicios se refieren a una instanciación de un proceso remoto, estos se componen de una solicitud y de una respuesta. Una característica importante de los servicios es que deben ser de ejecución rápida, es decir parar procesos que no requieran de mucho tiempo para dar una respuesta, ya que el cliente está esperando la respuesta para poder seguir con la ejecución del programa. Esta solicitud y respuesta se manejan con un servidor de servicio y un cliente de servicio. El primero es el que acepta las solicitudes, realiza los cálculos, envía la respuesta. El cliente es el que envía la solicitud y espera la respuesta del servidor. Los tópicos están "hechos" para ser llamados constantemente sin necesidad de que se requieran, pero los servicios solo deben ser llamados cuando se requiera. Si un servicio es llamado más de lo que se necesita puede reducir la velocidad en la ejecución del programa [33]. En la Figura 12 se puede observar este intercambio de información entre un cliente y un servidor, los cuales son dos nodos aparte, reforzando al nodo como unidad básica de ROS.

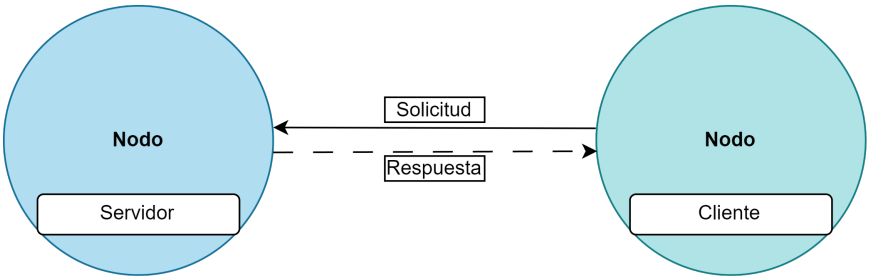

Figura 12: Diagrama de servicios en ROS2.
[33]

Acciones

Una acción a diferencia de un servicio, sí está pensada para ejecutarse durante un tiempo determinado que no debe ser corto necesariamente. Es una instanciación asíncrona a la función de otro nodo, lo que implica que no se debe esperar a la respuesta para poder seguir con la ejecución del programa. Similar a los servicios, habrá un nodo que sea el servidor y uno que sea el cliente. Otras características de las acciones es que tiene realimentación constante del nodo servidor, y en el proceso se puede cancelar el objetivo por petición del cliente. El cliente por otro lado, envía la orden de solicitud/cancelación, y espera a que el servidor termine de ejecutar la secuencia, mientras el mismo cliente sigue ejecutando otras tareas [34]. En la Figura 13 se muestra el intercambio entre los dos nodos (cliente y servidor), el cliente puede enviar la información del procedimiento que quiere solicitar, o cancelarlo, mientras que el servidor envía constantemente el estado del proceso, y el finalmente envía el resultado. La acción sigue siendo el intermediario, y dentro de la acción están contenidos los servicios con la solicitud y respuesta de la meta y del resultado. También hay un tópico que sirve para enviar la realimentación.

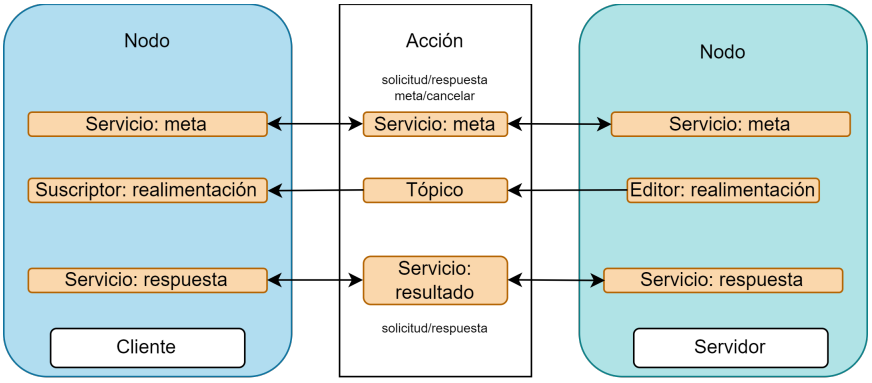

Figura 13: Diagrama de acciones en ROS2.
[34]

6.4.3. Parámetros

Los parámetros en ROS2 funcionan de igual forma que los atributos en cualquier lenguaje de programación en el que sirven para configurar una función, en este caso un nodo.

Estos parámetros permiten la inicialización de ciertos valores en los nodos, y su modificación durante la ejecución del código. Cada parámetro consiste de una llave, un valor y un descriptor. Existen también servidores de parámetros, que son como diccionarios a los cuales los nodos pueden acceder para almacenar y extraer parámetros durante la ejecución de un programa.

6.4.4. micro-ROS

Dados los requisitos de sistema de ROS2, existe un proyecto en desarrollo de micro-ROS, que es específico para poder ejecutar ciertas tareas de ROS en microcontroladores. Este proyecto está enfocado a conseguir aprovechar las capacidades de los microcontroladores en agentes robóticos, como el fácil acceso, consumo de energía, y tiempos de latencia. La arquitectura que sigue micro-ROS es la misma que la de ROS2, en la Figura 14 se muestran en azul los componentes que son específicos para microcontroladores, y en celeste los que son solamente un port de las librerías de ROS2 [35].

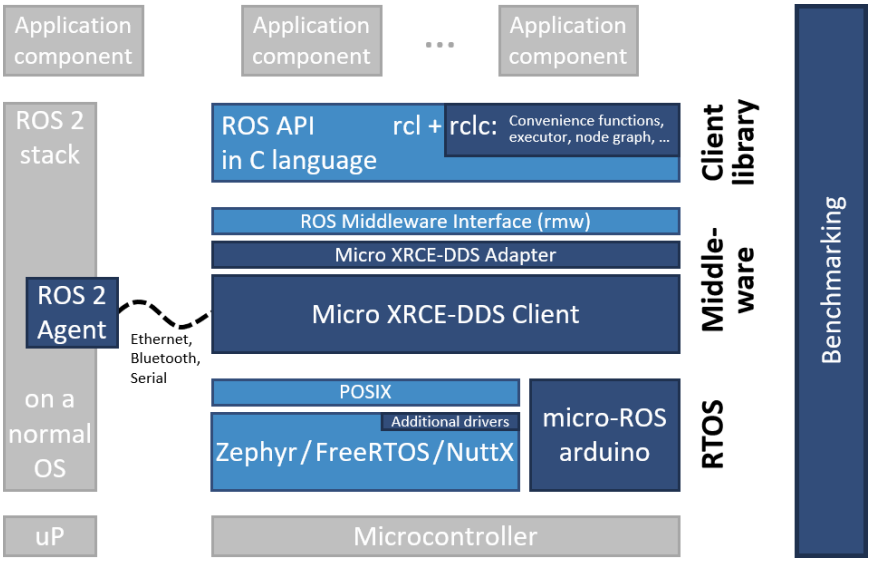

Figura 14: Arquitectura en desarrollo del port de ROS2 a microcontroladores, micro-ROS. [35]

En agosto del 2020, el ESP32 entró a la familia de microcontroladores con soporte en micro-ROS, siendo un punto de partida en microcontroladores con módulo WiFi y Bluetooth integrado. El port está desarrollado para un ESP32-DevKitC-32E [36], que tiene un módulo ESP32-WROOM-32E [37]. Además, aprovecha las capacidades de FreeRTOS [38], que es un sistema operativo en tiempo real para dispositivos embebidos, que ya utiliza esta familia de microcontroladores de forma nativa. Entre las capacidades que resalta la documentación de micro-ROS para el ESP32 son la memoria flash de 4 MB, la memoria ROM de 448 KB, la memoria SRAM de 520 KB, y los 16 KB de SRAM en RTC, que abre las puertas a ejecutar múltiples instrucciones, y hacer uso de los múltiples periféricos que ofrece el microcontrolador [39].

La documentación de micro-ROS indica que el flujo de compilación sigue una serie de cuatro pasos [40]:

- Creación: en este paso se descargan las librerías necesarias, específicas para el microcontrolador a utilizar.
- Configuración: se debe especificar la forma en que se va a comunicar el microcontrolador con el otro dispositivo, dirección IP o puerto, identificador, entre otros.
- Compilación: se generan los archivos binarios, ya sea por comandos o algún otro entorno editor que permita compilar el archivo.
- Carga: se actualiza el firmware del microcontrolador para que la aplicación de micro-ROS pueda ejecutarse.

6.5. Python

Python es un lenguaje de programación de alto nivel, interpretado y orientado a objetos. Este se ha caracterizado por tener una sintaxis relativamente sencilla. Con el lenguaje viene incluido una variedad de librerías entre las que destacan el procesamiento de cadenas, protocolos de internet, ingeniería de software, e interfaces de sistemas operativos. Esto y su sintaxis, hace que sea un lenguaje versátil de aplicar en diferentes aplicaciones [41]. En el contexto de este trabajo, se hace uso de diversas librerías de Python que desempeñan un papel fundamental en el desarrollo de las herramientas de software propuestas.

6.5.1. Programación orientada a objetos (*OOP*)

La programación orientada a objetos, OOP por sus siglas en inglés, es un paradigma de programación que permite crear aplicación más robustas, flexibles y fáciles de mantener. La idea base de este paradigma es que el código se puede organizar como un grupo de objetos, cada uno con sus propios atributos y funciones (métodos). Es decir, se utilizan *clases*, que son piezas simples y reutilizables de código, para crear instancias de estas clases, llamadas *objetos*. La perspectiva de este paradigma es buscar las relaciones de los objetos en el programa. Los cuatro principios que sigue la *OOP* son:

- Encapsulación: toda la información de un objeto esta definida en la clase, y no se puede acceder a ella de forma directa, sino a través de las funciones asociadas a la clase.
- Abstracción: solo las propiedades relevantes se pueden observar para alguna función específica. Es decir, el usuario solo interactúa con los atributos y métodos seleccionados de un objeto.
- Herencia: define relaciones de jerarquía entre diferentes clases, esto permite que atributos y métodos comunes puedan ser reutilizados. Se pueden crear clases a partir de otra, agregando características y funciones.
- Polimorfismo: permite que los objetos de diferentes clases respondan a cierta entrada de diferentes maneras, un mismo método puede responder diferente según la clase del que está recibiendo [42].

El trabajar con este paradigma ofrece ventajas como la reutilización de código, se tienen estructuras más simples de código, evita al programador duplicar código, da la posibilidad de proteger información de alguna clase. Un ejemplo básico de OOP se puede ver en la Figura 15, en el cual la *clase* es el círculo con atributos simples que componen a cualquier círculo en general, y métodos que en este caso son para definir u obtener características geométricas. En la parte inferior se definen tres objetos de esta clase, es decir tres círculos diferentes, cada uno con diferente radio o color.

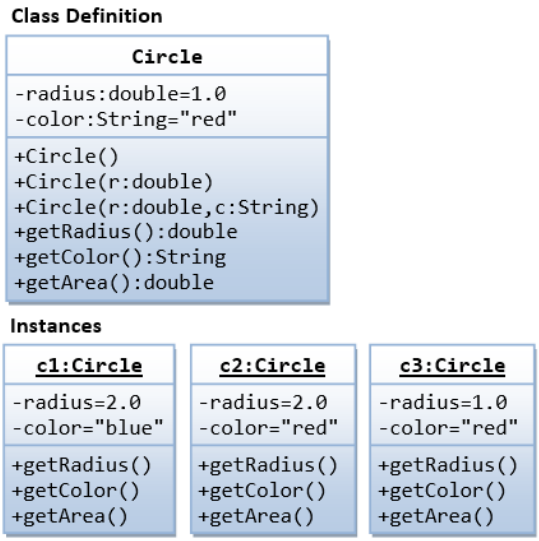

Figura 15: Ejemplo básico de programación orientada a objetos [43]

6.5.2. Ambientes virtuales

Dado que los proyectos creados con Python usualmente van a utilizar módulos y paquetes que no son parte de las librerías estándar con las que se instala, el manejo de estas adiciones será esencial para el correcto funcionamiento de cada aplicación y que sea posible su distribución. La solución a este manejo de dependencias es la creación de ambientes virtuales (*virtual environments*). Estos ambientes permiten que los usuarios de Python instalen y actualicen paquetes y módulos de cierto proyecto sin interferir en otros proyectos que estén en el mismo sistema. Es decir cada proyecto de Python creado puede tener dependencias diversas e incluso versiones de Python diferentes, y funcionar en el mismo sistema [44]. El módulo para crear y manejar estos ambientes virtuales se llama *venv* [45], por defecto, al crear un ambiente virtual se crea con la versión más reciente instalada en el sistema, sin embargo, es posible utilizar una versión diferente si se tiene instala en el sistema.

Una parte fundamental de los ambientes virtuales es el activarlos previo a trabajar en algún proyecto, en Windows se debe ejecutar el comando mostrado en 6.1. Esto permitirá que cada dependencia instalada, actualizada y utilizada durante el desarrollo del proyecto, se guarde en este ambiente específico y no en el sistema. Cada vez que se regrese a trabajar a este proyecto, se debe activar el ambiente virtual [45].

```
"nombre del ambiente"\Scripts\activate.bat
```

Código 6.1: Activación de ambientes virtuales en Python

6.5.3. Pygame

Pygame es un set de módulos de Python diseñado para la creación de videojuegos. Sin embargo, sus características permiten la creación de programas multimedia también, en el lenguaje Python. Es altamente portátil y tiene soporte en la mayoría de sistemas operativos y plataformas. Entre sus propiedades destaca el control de las funciones y manejo de eventos, esto quiere decir que da un buen control de los eventos de Pygame incluso integrando otras librerías de Python [46].

6.5.4. Sockets

La librería de *socket* en Python permite la creación de objetos del mismo nombre. Estos son utilizados para enviar mensajes a través de una red, la cual puede ser lógica, local a la computadora, o una red que esté físicamente conectada a una red externa. Un socket es una representanción en software del punto final local de una ruta de comunicación de red. Las aplicaciones más comunes de sockets son las de tipo cliente-servidor, el servidor espera conexiones de los clientes [47]. Cabe resaltar que el módulo de Python hace una interfaz a la API de sockets de Berkeley (una interfaz de programación para sockets de internet utilizados para comunicación entre procesos [48]).

El módulo permite especificar el tipo de socket que se va a utilizar. Al especificarlo como `socket.SOCK_STREAM` utiliza el *Transmission control Protocol* (TCP). Este ofrece las ventajas del protocolo mencionado antes, como que es confiable, y recibe datos en forma ordenada. En la Figura 16 se puede observar el flujo de funciones que se llama para establecer el intercambio de datos entre cliente y servidor, el lado izquierdo de la imagen representa el servidor y el derecho el cliente [47].

6.5.5. PyQt5

PyQt5 es una integración de las librerías Qt versión 5 en Python. Qt es un set de librerías implementadas en C++, compuesta por APIs de alto nivel que permiten accesar a varias características del sistema en que estén operando, incluidas funciones de desarrollo de interfaces de usuario [50]. Las dos versiones más recientes de PyQt son PyQt5 y PyQt6, sin embargo, la versión con más soporte en documentación sigue siendo PyQt5 al momento de redactar este documento. Una de las mayores ventajas de PyQt es que al estar construido sobre el marco Qt, que es multiplataforma, las aplicaciones pueden construirse en casi cualquier sistema operativo, haciendo que sus Widgets se sientan y parezcan nativos del sistema operativo en cuestión [50]. Agregado a los componentes estándar de GUIs, Qt provee de herramientas como las siguientes:

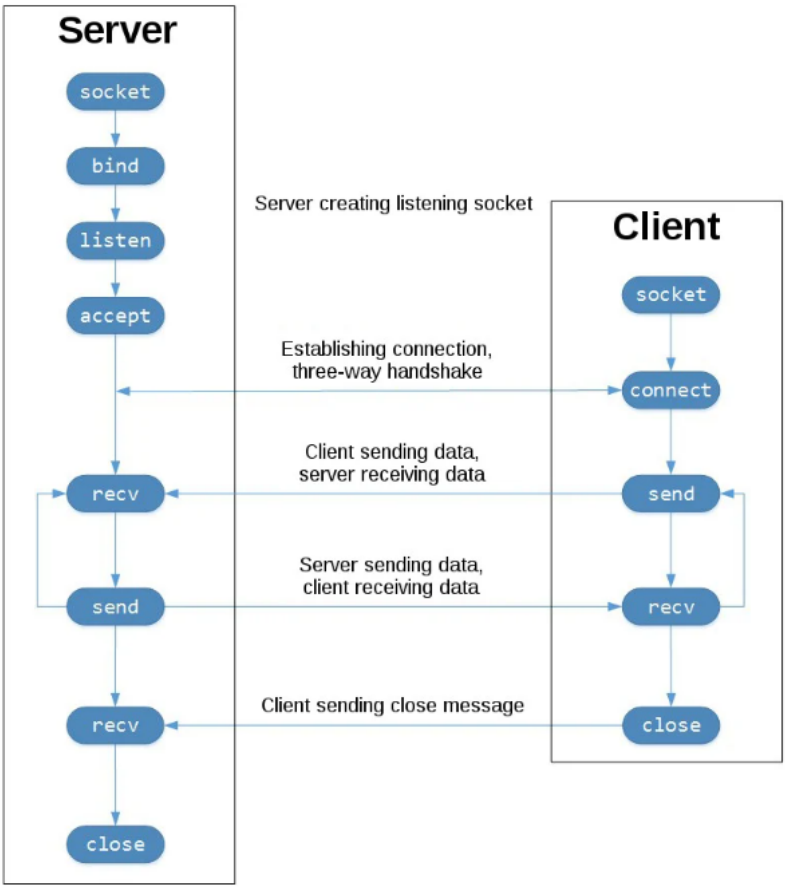

Figura 16: Secuencia de funciones del socket API y flujo para TCP [49]

- Vistas de datos impulsadas por MVC (modelo-vista-controlador) (hojas de cálculo, tablas)
- Interfaces y modelos de bases de datos
- Trazado de gráficos
- Visualización de gráficos vectoriales
- Reproducción de multimedia
- Efectos de sonido y listas de reproducción
- Interfaces para hardware

Otra de las ventajas de PyQt5 es que dado la forma en que se estructura, permite manejar aplicaciones complejas en componentes más pequeños, permitiendo arquitecturas más robustez y mantenibles. A diferencia de otros módulos para crear interfaces gráficas, provee de funcionalidades completas con sus componentes, y no se limita a desplegar objetos

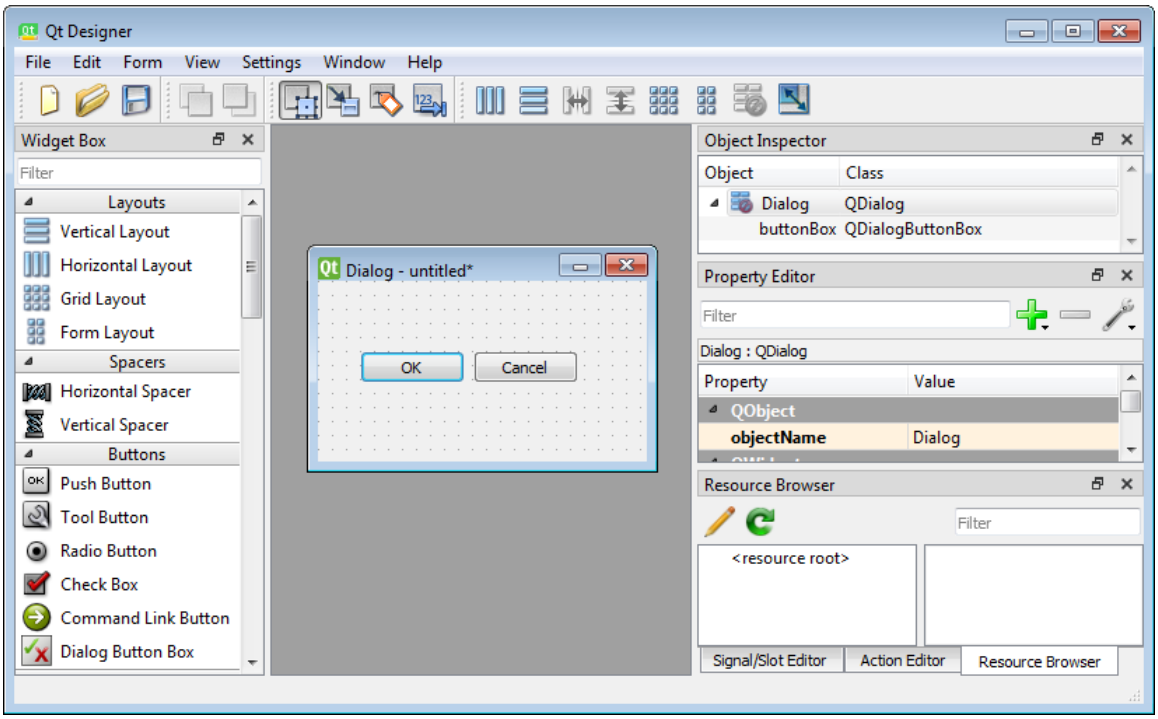

Figura 17: Ventana de Qt Designer en Windows
[51]

en una pantalla teniendo que manejar sus funciones en métodos aislados. PyQt5 provee de una versión de diseño de interfaces tipo *drag-and-drop* llamado *QtDesigner*, se puede ver en la Figura 17. Este acelera la creación de la estructura de la interfaz, facilitando añadir funciones posteriormente, y también permite realizar cambios que se verán reflejados en la ejecución del *script* que importe estas ventanas. También permite diseñar widgets, cajas de diálogo y ventanas completas [51].

CAPÍTULO 7

Definición de requisitos de las herramientas de software a desarrollar

A continuación, se describen los requisitos que se definieron para las tres funciones principales programadas para los Pololu 3Pi+. Estos se establecieron con base en los objetivos específicos planteados. Las tres funciones principales son: las herramientas de simulación, las herramientas de programación y el monitoreo de datos de los Pololu 3Pi+ en la plataforma del Robotat.

De forma general, la estructura del código debe cumplir con ser modular, mantenible, replicable y editable. Para integrar estas tres funciones principales, se propuso incluirlas en una interfaz gráfica que fuera amigable para el usuario y que permitiera la integración fluida de las funciones entre sí. Esto seguiría un flujo completo de experimentación con estos agentes robóticos.

7.1. Herramientas de simulación

Los requisitos de simulación se han definido con el objetivo de proporcionar resultados precisos con respecto a los parámetros proporcionados y, al mismo tiempo, ofrecer una manera sencilla de simular el comportamiento de los agentes robóticos Pololu 3Pi+ en la plataforma del Robotat, bajo diferentes condiciones.

- Modularidad del contexto de simulación:

  El contexto de simulación debe ser modular para permitir la integración sencilla de nuevas condiciones en el escenario de simulación.

- Ventana de visualización representativa:

  La simulación debe incluir una ventana de visualización que represente con precisión

las condiciones reales de la plataforma del Robotat. Esto, junto con el requisito de modularidad, asegurará que la simulación sea adaptable a cambios en la vida real, que se reflejarán en la ventana de animación de la simulación.

- Múltiples robots:

  Es importante que la simulación permita la adición de una cantidad variable de robots al entorno, cada uno con parámetros de simulación distintos. Esto proporcionará la flexibilidad necesaria para llevar a cabo diversos experimentos y evaluar diferentes algoritmos de control.

- Parámetros de controlador y características personalizables:

  Cada robot debe tener la capacidad de recibir y aplicar una variedad de parámetros de controlador, objetivos y características físicas, y simular con base en estos. Esto añade flexibilidad al modelado de las condiciones en cada simulación.

- Estructura estándar de mundo a simular:

  Establecer un estándar para la estructura del mundo que se simula facilitará la modificación de diversos escenarios. Además, contribuirá a la adaptabilidad a diferentes contextos de simulación.

- Interfaz gráfica amigable al usuario:

  Debe ser posible integrar la simulación en una interfaz gráfica que resulte intuitiva y accesible para el usuario.

7.2. Herramientas de programación

Esta función se plantea con requisitos basados en las capacidades investigadas de los ESP32 para recibir actualizaciones *over-the-air* (OTA). Su objetivo principal es aprovechar estas capacidades para implementar un algoritmo de actualización de firmware para múltiples agentes robóticos de manera inalámbrica.

- Replicable:

  Es fundamental definir una estructura y una secuencia de pasos que se deban seguir cada vez que se quiera llevar a cabo una actualización vía WiFi a los ESP32 en la red del Robotat. Esto garantizará que el proceso sea consistente y pueda repetirse en diferentes situaciones, independientemente de la cantidad de ESP32 que se deseen actualizar.

- Integración con la interfaz :

  Los pasos que se definan para llevar a cabo la programación OTA deben poder integrarse en la interfaz de usuario. Además, se debe asegurar que el proceso sea lo más automático posible para facilitar su implementación.

- Integración con el firmware de los ESP32 de los Pololu 3Pi+:

  Para que los Pololu 3Pi+ sean capaces de ejecutar nuevas rutinas surgidas a partir de una actualización de firmware realizada de forma remota, se debe definir una estructura

que permita modificar el código de los ESP32 basándose en las decisiones tomadas en la interfaz. Esto asegurará la compatibilidad entre el hardware y el nuevo software.

- Robustez:

  El proceso de carga de nuevos programas debe incluir pasos de verificación previos antes de intentar realizar el proceso de carga. Además, se debe garantizar la robustez en la infraestructura de software para la conexión con la red.

- Verificable:

  Debe ser sencillo verificar que los ESP32 a utilizar cuenten con la capacidad de recibir actualizaciones OTA. Esto es esencial para asegurarse de que los dispositivos cumplen con los requisitos necesarios para este proceso.

7.3. Monitoreo de datos

La función de monitoreo de datos tiene como objetivo principal registrar el movimiento real de los agentes robóticos, haciendo uso del sistema de captura OptiTrack sobre la plataforma. Esto posibilita la obtención de la pose de los marcadores colocados en los Pololu 3Pi+, permitiendo su visualización en una ventana de animación. Además, se busca la generación de reportes a partir de los datos capturados.

- Conexión con el Robotat:

  Las funciones de conexión al sistema de captura del OptiTrack deben ser capaces de establecer una conexión TCP/IP con el servidor del Robotat y, además, obtener la pose de los marcadores en la plataforma del Robotat. La pose debe poder actualizarse conforme evoluciona el experimento y capturarse en tiempo real.

- Generación de reportes:

  Los datos obtenidos por el sistema de captura deben poder exportarse a un formato que sea útil para el usuario.

- Integrable:

  Las funciones y la captura de datos deben ser integrables tanto en la interfaz de usuario como en una ventana de animación. Esta ventana facilitará la visualización de lo que está ocurriendo con los agentes sobre la plataforma del Robotat. La integración con la interfaz de usuario debe ser capaz de modificar algunos parámetros de la captura que se desea realizar.

- Estabilidad:

  Deberá asegurarse, en la medida de lo posible, una conexión estable al sistema de captura del OptiTrack para una captura íntegra de la pose de los agentes robóticos y los marcadores de interés.

CAPÍTULO 8

Desarrollo de herramientas de simulación para los Pololu 3Pi+ en el Robotat

En este capítulo se describen los pasos tomados para desarrollar un algoritmo de simulación para los agentes robóticos Pololu 3Pi+ dentro del ecosistema de Robotat. Este algoritmo tuvo como objetivo poder simular uno o varios agentes robóticos en una ventana de animación que permitiera visualizar el comportamiento resultado de aplicar un controlador al modelo de un robot móvil.

8.1. Resultados

Con el propósito de crear herramientas de simulación para los agentes robóticos Pololu 3Pi+ de manera modular, mantenible y reutilizable, se decidió utilizar Python. Esta elección se basa en su capacidad para integrar librerías y módulos de forma gratuita, así como en la posibilidad de trabajar en un ambiente virtual que encapsule todos los elementos necesarios. Además, Python añade un nivel de mantenibilidad que facilita futuras actualizaciones. A lo largo del capítulo se describen las librerías que fueron necesarias por cada módulo de la simulación de los agentes robóticos.

8.1.1. Desarrollo de clases

Se utilizó el paradigma de programación orientada a objetos, considerando las características de los objetos a representar en el simulador, que tendrían múltiples funciones que realizar. Conforme se vaya desarrollando la interfaz, se podrían añadir funcionalidades e incluso otro tipo de agentes robóticos siguiendo la misma estructura propuesta para cada clase.

En la herramienta de simulación, se tienen tres clases principales: la ventana de animación, la de tipo robot y la de personaje robot.

Ventana de animación

El módulo de Python utilizado para la visualización de las animaciones es Pygame. Dado que está especializado en la creación de juegos en Python, cuenta con varios métodos integrados para animar imágenes. Esto fue fundamental para actualizar la posición (x, y) y la orientación θ de la representación de los Pololu 3Pi+. Aunque esta es la librería principal, se necesitó importar la librería `os` que ayuda a obtener la ruta relativa de archivos, en este caso de imágenes que se instancian desde esta clase. La librería `ctypes`, permitió a la ventana de animación estar consciente de las propiedades DPI del ordenador en el que se está ejecutando la aplicación, para escalar con respecto a esa resolución, con el siguiente comando: `ctypes.windll.user32.SetProcessDPIAware()`.

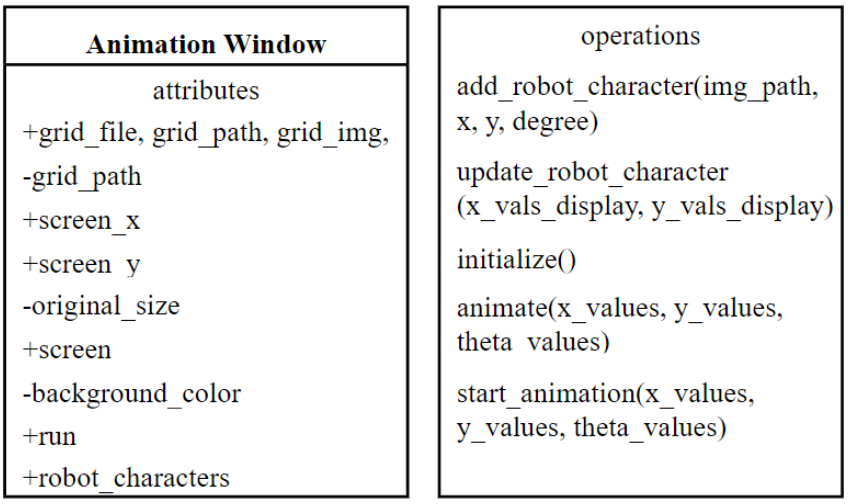

Figura 18: Diagrama UML tipo clase: Ventana animación.

En la Figura 18 se muestran los atributos de la ventana de animación, los cuales se centran en el dimensionamiento de la ventana (`screen_x, screen_y, screen`). Durante la inicialización de la clase, se establecen únicamente los valores numéricos, y posteriormente, en el método `initialize()`, se instancian los métodos proporcionados por la librería Pygame para crear la ventana con esos parámetros. Para una mejor visualización de la animación en relación a la plataforma del Robotat, se ha implementado una cuadrícula con cuadros de 20×20 píxeles, representando divisiones de 10 cm en ambos ejes coordenados (x, y), abarcando un total de 380×480 cm, respectivamente. Además, para una mejor visualización de todos los elementos que se necesiten representar en la ventana de animación, se ha establecido el color blanco de fondo, con el atributo `background_color`.

El método de inicialización, como se mencionó anteriormente, crea la ventana con los parámetros establecidos en los atributos de la clase. Utiliza los métodos `pygame.init()` para crear una instancia de tipo Pygame, `surface.fill` y `surface.blit` para llenar la ventana con el color establecido, y para colocar el fondo cuadriculado desde el origen de la ventana.

El método `animate` invoca al método previamente mencionado, `initialize`, y posteriormente llama a `start_animation`. Los parámetros que reciben estos métodos se pueden observar en la Figura 18. Este conjunto de acciones, al haber llamado a los otros dos métodos, resulta en la apertura de la ventana de animación y en la ejecución de la animación con los valores proporcionados para (x, y, θ) en un determinado *script* en el que se incluya esta clase.

El método `add_robot_character` permite añadir una cantidad variable de robots a la ventana de animación. Estos se almacenan en una lista con los parámetros de la imagen del personaje del robot, así como su valor de posición y orientación.

Finalmente, el método de la clase de animación que ejecuta todos los resultados es `start_animation(x_values, y_values, theta_vaues)`. La estructura principal de este código se puede observar en el Código 8.1. Se comienza con una variable de índice que sirve para iterar sobre los resultados de (x, y, θ) Hay una variable de tipo booleano llamada `animation_running` que comienza en Falso y se vuelve Verdadero cuando la animación se está ejecutando. Sirve como bandera para evitar que el ciclo de simulación entre en un bucle infinito, y permite repetir la simulación varias veces cambiando el valor de esta bandera. Dentro del bucle `while` que se ejecuta, se utilizan funciones del módulo de Pygame para manejar el evento de "cerrar ventana", el cual cambia la bandera `self.run` a Falso, lo que lleva al final del bucle y termina el programa. Luego, se actualiza y redibuja la cuadrícula en pantalla. Si se presiona el botón de reproducción y la animación no está en curso, se inicia y el índice se restablece a cero. Si la animación está en curso y el índice es menor que la longitud de la lista `x_values`, el código itera sobre los personajes de robot y actualiza sus posiciones y orientaciones en pantalla. Después, se actualiza la pantalla con `pygame.display.flip()`, se incrementa el índice y se vuelve a actualizar la pantalla. Esto se repite hasta que se alcanza el final de la lista `x_values`. En este momento, la variable `animation_running` se cambia a Falso para detener la animación.

Cabe resaltar que, como se puede observar, se realiza una constante actualización de la imagen de fondo (la cuadrícula) y el color de fondo (blanco). Esto es necesario ya que sin esta actualización recurrente, la imagen del robot/s quedaría impresa en todos los frames de simulación por los que haya pasado, es decir, dejaría un rastro con la imagen y la cuadrícula no estaría presente todo el tiempo, lo que dificultaría la verificación del alcance del objetivo establecido para el robot.

En la Figura 19 se puede observar la ventana de animación con la cuadrícula de fondo, cabe resaltar que se agregó también un par de ejes coordenados en el centro de lo que representa la plataforma, ya que en la plataforma física del Robotat el origen se sitúa en el centro con sus ejes positivos en la dirección en que se muestran. También se puede observar un botón de reproducción que se definió en una clase separada para poder realizar acciones al cambiar de un estado booleano a otro. Esta definición se encuentra en el mismo módulo de la clase de animación. La orientación de la ventana de animación y de las demás herramientas desarrolladas se presenta desde una perspectiva que representa la plataforma física en una visualización inferior, considerando la entrada del laboratorio del Robotat como referencia. La parte inferior de la imagen corresponde a la vista desde la entrada del laboratorio.

El constructor de la clase `button_pygame` recibe cuatro argumentos: `x, y` que representan las coordenadas de la esquina superior izquierda del botón, `image` que es la imagen

que representará el botón y `screen` que es la pantalla donde se va a dibujar el botón. `self.clicked` es una bandera que indica si el botón ha sido presionado, y `self.action` guarda si se ha llevado a cabo una acción sobre el botón. Se utiliza la función de Pygame `pygame.mouse.get_pose()` para obtener la posición del *mouse*. Esto junto con la función `.collidepoint` y `get_pressed` verifica si el mouse está sobre alguna posición en específico y se ha presionado el click derecho para ejecutar una acción. En este caso la acción es hacer el cambio entre Verdadero y Falso y devolver ese valor booleano.

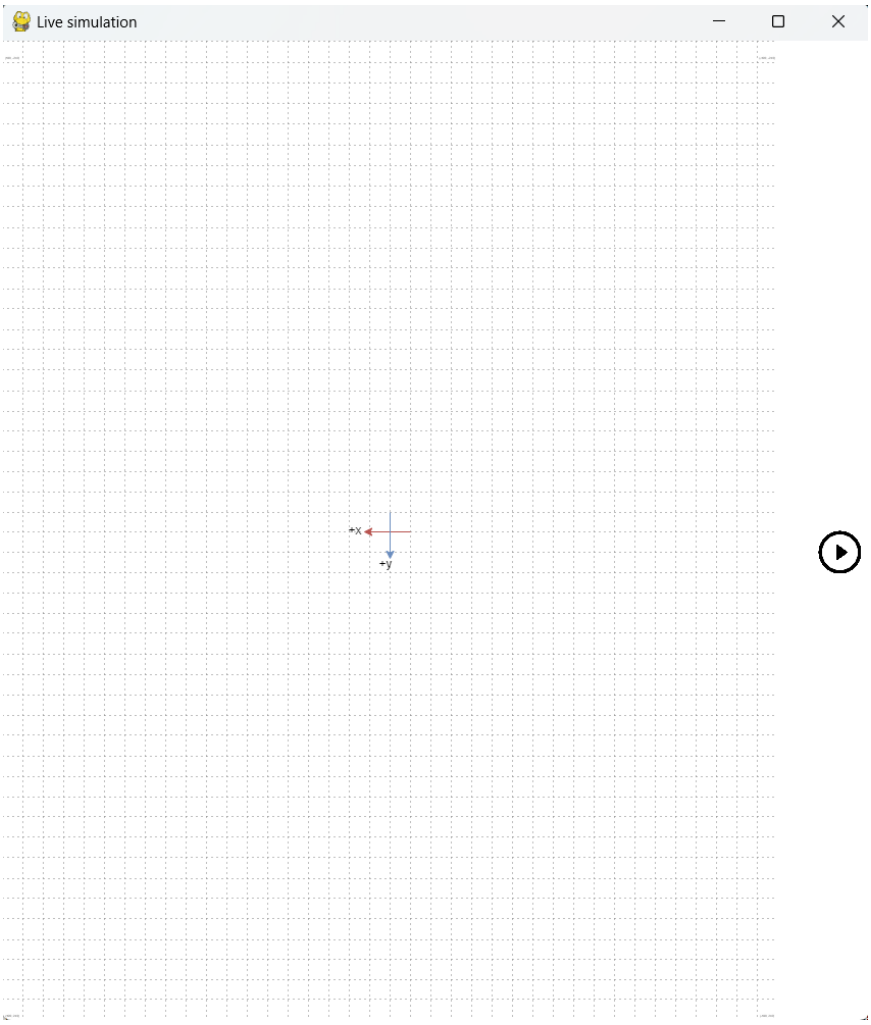

Figura 19: Ventana de animación previo a la simulación.

Otro detalle a resaltar es la representación del Pololu 3Pi+ que se puede ver en la Figura 20. Esta imagen tiene una pequeña flecha roja representando el frente del robot, para una mejor visualización del movimiento esperado en la ventana de animación. También se orientó a propósito en el mismo sentido que el eje positivo $+x$ del eje coordenado de la plataforma siguiendo la regla que el frente de los robots móviles con ruedas va en x positivo. Esto facilita la animación al no tener que hacer ajustes extras de orientación para la actualización de la imagen.

```
1 def start_animation(self, x_values, y_values, theta_values):
2     index = 0
```

```
    animation_running = False
    while self.run:
        ...
        self.screen.fill(self.background_color)
        self.screen.blit(self.grid, (0, 0))

        if self.play.action and not animation_running:
                ...
                animation_running = True
                index = 0
        if animation_running and index < len(x_values):
                for i in range(len(self.robot_characters)):
                    x_robot = x_values[index][i]
                    y_robot = y_values[index][i]
                    theta_robot = theta_values[index][i]
                    robot = self.robot_characters[i]
                    robot.update(theta_robot, x_robot, y_robot)
                    robot.rotate_move()
```

Código 8.1: Flujo de animación en el método start animation

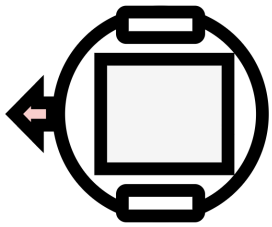

Figura 20: Representación del Pololu 3Pi+ en la ventana de animación.

Dado que la ventana de animación tiene el origen $(0, 0)$ está en la esquina superior izquierda, y el sistema de coordenadas funciona diferente en la plataforma del Robotat, se tuvo que realizar un mapeo. Este se definió en un módulo aparte llamado `map_coordinates.py`, con una función llamada `inverse_change_coordinates`. Esto facilitaría el arreglo de los valores para la cantidad variable de objetos que se necesitarán representar en la plataforma en la ventana de animación. Esta función recibe de parámetros el valor de (x, y) y la altura y ancho de la ventana en que se está trabajando. Luego entra a una serie de condiciones dependiendo del cuadrante en que se encuentre el robot, esto ayuda a establecer también un límite según el cuadrado que ocupa cada robot, haciendo que si está en alguno de los bordes de la plataforma, haga un desfase de forma que se siga viendo el robot completo ya que la traslación la hace con el centro del cuadrado. Finalmente, devuelve los valores de (x, y) listos para desplegar en la animación. Estas funciones ya están tomando en cuenta la escala duplicada de la plataforma para que tenga una correcta visualización.

Robot Pololu

La clase de tipo Pololu contiene los atributos y métodos que se muestran en la Figura 21. Esta clase se creó en un módulo separado en la ruta `src/robots/robot_pololu.py`.

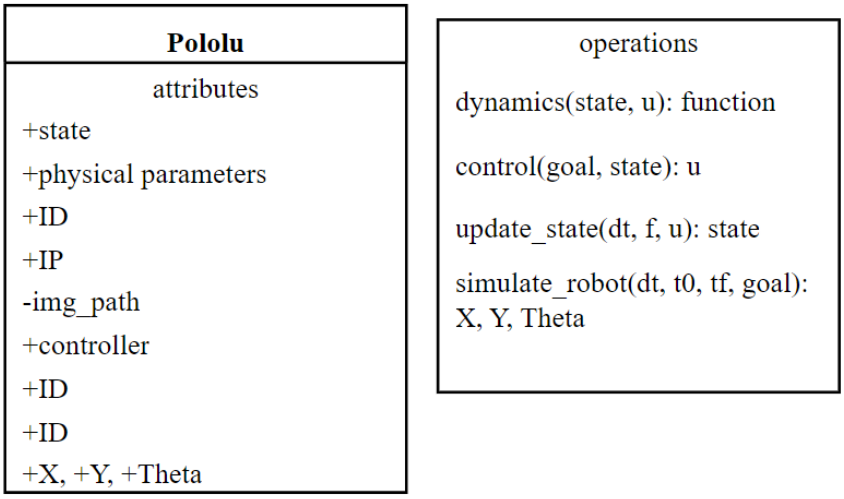

Figura 21: Diagrama UML tipo clase: Pololu.

Fue necesario importar el módulo *NumPy* de Python el cual es un paquete de computación científica [52]. También se utilizó el módulo de Pygame, para poder inicializar la imagen que representa a los Pololu 3Pi+ dentro de la clase. Los atributos de esta clase abarcan el estado actual, los parámetros físicos del robot, un identificador único ("ID"), la dirección IP del ESP32 que controla el robot (lo cual es esencial para la programación remota), la ruta del archivo de imagen que representa al Pololu 3Pi+, el controlador a aplicar y la pantalla en la que se mostrará.

En el constructor de la clase, se realizan las inicializaciones necesarias para los arreglos que contendrán las iteraciones del ciclo de simulación, así como para el estado, las entradas y los valores de posición y orientación (x, y, θ).

El método `dynamics` acepta como argumentos el estado y la entrada de control `u`, y utiliza estos valores para calcular la dinámica del sistema del uniciclo. En cuanto a los controladores, se han implementado en el método `control`. Para mantener la flexibilidad y generalidad, se ha optado por la inclusión del atributo *controller*, como se aprecia en el Código 8.2. Este atributo toma como parámetros tanto la meta como el estado del robot, y devuelve el conjunto de velocidades $u = [v, w]$.

```
1 def control(self, goal, state):
2     u = self.controller(state, goal)
3     return u
```

Código 8.2: Atributo para el controlador

Esto habilita el uso de cualquier tipo de controlador definido en otro módulo (por ejemplo, *controllers.py*) al instanciar los métodos de la clase Pololu. Esto es posible siempre y cuando el controlador esté definido de la misma manera que espera el método y se pase como un atributo de la clase. Un ejemplo de su uso se describe en el Código 8.4.

Los controladores se definen dentro de una carpeta llamada *controllers*. Esta carpeta contiene todos los *scripts* de Python con la definición tipo función de los controla-

dores. Para validación de las herramientas desarrolladas se escribieron dos controladores: `exponential_pid.py` y `pid_controller.py`, regresando la velocidad `u`. Estas funciones son las que se utilizan en la simulación de cada robot actualizando sus estados. Estos controladores son la base para la parte de programación, pero no el mismo tipo de definición, es decir, estos controladores contienen valores de constantes para la simulación y dependencias de NumPy que hacen que trabaje de forma correcta con la simulación.

El método `update_state` toma como parámetros el periodo de muestreo, la función de dinámica y las velocidades contenidas en `u`. Su función es actualizar el estado utilizando el método de aproximación Runge Kutta 4. Aunque no devuelve un nuevo valor, actualiza el atributo de estado que ya ha sido inicializado en la clase tipo Pololu. Finalmente, se lleva a cabo la simulación del robot para obtener tanto la posición y orientación, como las velocidades lineales y angulares.

El método `initialize_image` se encarga de cargar la imagen del robot Pololu desde la ruta especificada en `img_path`. Primero, utiliza la función `pygame.image.load` para cargar la imagen en formato alfa, que soporta transparencias. Luego, redimensiona la imagen a un tamaño adecuado mediante `pygame.transform.scale`, estableciéndola en 50×50 píxeles. Finalmente, se obtiene el rectángulo de la imagen con `self.img.get_rect()`, que será útil para posteriores operaciones de posicionamiento.

El método `simulate_robot` lleva a cabo la simulación del robot Pololu. Recibe como argumentos *dt* (el intervalo de tiempo entre cada paso de simulación), *t0* (el tiempo inicial), *tf* (el tiempo final) y *goal* (el objetivo de la simulación). Primero, se calcula el número de pasos de simulación N con base en los tiempos proporcionados. Se inicializan diversos arreglos para almacenar las trayectorias de las variables de estado, las entradas y las salidas del sistema. Luego, se inicia la trayectoria con el estado inicial en `self.XI[:, 0] = self.state`.

Se procede a cargar la imagen del robot utilizando el método `initialize_image`. Después, se ejecuta un bucle que itera a través de los pasos de simulación. En cada iteración, se calcula la entrada de control `u` utilizando el método `self.control(goal, self.state)`. Luego, se actualiza el estado con el método `self.update_state(dt, self.dynamics, u)`. Los valores de posición y orientación se almacenan en los arreglos correspondientes.

Al final del bucle, se registra el número total de pasos tomados durante la simulación en `self.steps`, y se marca la simulación como completada. Además, se almacenan los resultados de la simulación en `self.simulation_results`. El método devuelve las trayectorias de posición X, Y y orientación $Theta$.

Personaje robot

La clase `robot_character` es fundamental en la representación gráfica de los robots en la animación. Sus atributos se pueden observar en la Figura 22. Al ser instanciada, requiere la ruta de la imagen que servirá como apariencia del robot (`img_path`), así como sus coordenadas iniciales (x, y), su orientación inicial en grados (`degree`), la pantalla donde será visualizado (`screen`), y el tamaño de la pantalla (`size`).

Una vez inicializada, la clase carga la imagen del robot desde la ruta proporcionada y la

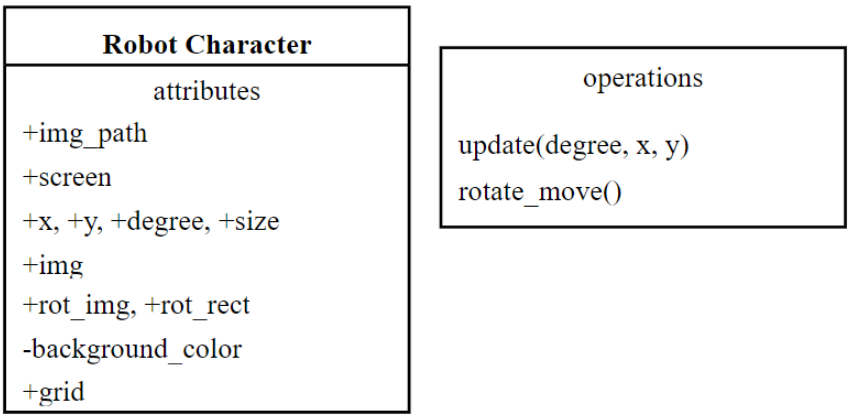

Figura 22: Diagrama UML tipo clase: Personaje.

escala a 40×40 píxeles si es necesario. Asimismo, crea una copia de la imagen que puede rotar para reflejar los cambios de orientación. También establece un rectángulo de rotación para asegurar la correcta visualización del robot. Además, la clase define el color de fondo (`background_color`) y carga una imagen de fondo de cuadrícula que servirá como referencia de coordenadas.

El método `update` permite modificar el estado del robot en términos de orientación y posición. Esta actualización es crucial para que el método `rotate_move` pueda operar sin necesidad de recibir estos valores como argumentos adicionales.

El método `rotate_move` se encarga de rotar la imagen del robot según su orientación actual y la posición en la pantalla. Esta rotación se realiza a partir de la imagen original y se aplica antes de ser mostrada en la pantalla.

8.1.2. Flujo de simulación

La forma original en que se había estructurado el código era colocando opciones en una primera versión de una interfaz de usuario para que se construyera la simulación a partir de estas selecciones, la mayoría dadas en forma de listas desplegables. Estas contenían opciones como: número de robots, tipo de control a implementar, control a implementar, entre otras. Sin embargo, esta forma de estructurarlo lo volvería más complejo de manejar de forma ordenada mientras más robots existieran, y las condiciones bajo la cual se ejecutaría la simulación serían menos editables. Por ello se migró a una forma de manejar las simulaciones como *mundos*, estos mundos son un archivo de extensión JSON, en el que se definen las condiciones completas de la simulación, desde características de la animación, hasta los parámetros modificables de los robots. Esto permitió que en el código principal de la simulación, se instanciaran las clases antes mencionadas, creando los objetos a partir de lo definido en el JSON, siendo un código más general y modular.

La estructura del JSON se muestra en la Figura 23 en el que los primeros parámetros corresponden a las características de la simulación, como el tiempo que durará, el período de muestreo, el número de robots, y las dimensiones de la plataforma (se está utilizando un

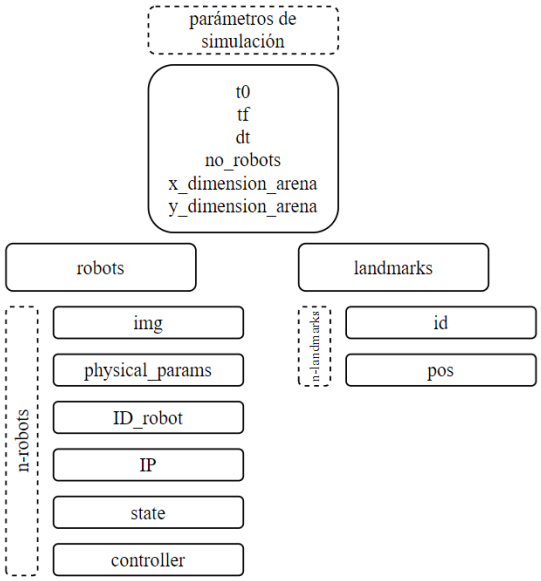

Figura 23: Definición del *mundo* a simular en el archivo JSON.

valor de 850×960 para mantener la relación de la plataforma original $380 \times 480cm$ siendo el doble, excepto el valor de x que debe dar espacio para el botón de reproducción. Después de esto se define un arreglo denominado "robots", este puede contener una cantidad variable de robots, en el que cada uno tiene un *string* con la ruta de la imagen que lo representa, un objeto que contiene las características físicas del robot (diámetro, radio de las llantas, velocidad máxima izquierda y derecha), seguido de esto tiene un identificador único *ID* que están previamente definidos por el robot físico y la IP estática que está asignada en la red del Robotat en los ESP32 correspondientes. También se agrega un arreglo con el valor inicial del estado, y un string con el nombre de controlador a implementar. Esta misma estructura se repite para todos los robots que se deseen agregar.

Para los *landmarks* se define también un arreglo que contiene el identificador del landmark, y un arreglo definiendo su posición (x, y). Estos landmarks con su identificador, que coincide con el identificador del robot, se utilizaron para definir la meta que el robot debía alcanzar para fines de la simulación únicamente.

Con estos cambios que se implementaron, el flujo de simulación se puede describir con el Algoritmo [1]. Primero, se importan las librerías necesarias (JSON, NumPy) y se importan los módulos correspondientes a las demás clases involucradas que son las descritas con anterioridad (ventana de animación, robot Pololu, personaje robot), además se importan los controladores de interés y el mapeo de coordenadas.

Se carga el archivo JSON, desde la carpeta llamada *worlds* que contiene todos las definiciones de los mundos que se quieren simular, para que posteriormente sea más práctica la selección del mundo que se quiere probar. Una vez se haya importado el JSON utilizando la librería del mismo nombre, se crean objetos a partir de lo definido en el mundo. Se llama a cada uno con su *key* (por ejemplo, `robots = world['robots']`). En este paso es importante obtener todos los valores para que Python los pueda utilizar, se recomienda que se utilice el mismo nombre de variable para poder identificarlos de la misma forma en que se definió

Algoritmo 1: Flujo de simulación en código principal

Data: world_definition.json
Result: Animación de Robots
Cargar *world_definition.json*;
Inicializar ventana de animación;
for *cada robot en robots* **do**
    Obtener controlador y objetivo asociados;
    Crear instancia de Pololu con parámetros específicos;
    Agregar personaje a la animación;
Obtener *dt*, *t0* y *tf* del mundo;
for *cada robot en robots* **do**
    Simular robot y almacenar resultados;
Convertir coordenadas a formato de visualización;
Animar la ventana con los resultados de simulación;

el mundo, y poder pasarlos de argumentos a la creación de los objetos a partir de las clases que se definieron.

Para los controladores, ya que en el JSON están en formato de string y se tienen que pasar como una función a la clase Pololu, se hace un diccionario con los nombres correspondientes y el nombre de la *key* del JSON. Esto se puede observar en el Código 8.3.

```
controller_map = {
    'exponential_pid': exponential_pid,
    'pid_controller': pid_controller,
}
```

Código 8.3: Diccionario para mapear de *strings* a funciones definidas en Python

En el Código 8.4 se puede ver la función *lambda* de la forma en que se crean en el código principal de simulación. Los valores de la meta y el controlador seleccionado se obtienen previamente del archivo JSON.

```
lambda state, goal=desired_goal,
ctrl_func=controller_function: ctrl_func(state, goal)
```

Código 8.4: Función lambda para los controles

Luego de inicializar la ventana de animación, se crean los objetos de la clase Pololu. Se itera a través de la información de los robots del archivo JSON, creando instancias para cada uno de los que estén ahí definidos, y agregándolos a una lista. En esta misma iteración se agrega la información de los personajes de tipo robot. Se añaden los personajes a la ventana de animación utilizando el método `add_robot_character`.

En la última sección del código, se están preparando los datos para la animación de los robots. Primero, se inicializan tres listas vacías: `x_vals_display`, `y_vals_display` y

`theta_vals_display`. Estas listas se utilizarán para almacenar las coordenadas de posición y orientación de los robots durante la simulación.

A continuación, hay un bucle que itera sobre cada robot en la lista de robots. Para cada robot, se obtiene su `ID_robot` y se busca el `landmark` más cercano que tenga el mismo ID. Si se encuentra un landmark, se obtiene su posición. Si no se encuentra, se establece un objetivo predeterminado en [0, 0].

Luego, se realiza una simulación del robot con el tiempo de muestreo (dt), tiempo inicial (t0), tiempo final (tf) y el objetivo actual `current_goal`. Los resultados de la simulación, que representan las coordenadas (x, y) y la orientación del robot, se almacenan en las variables `x_results`, `y_results` y `theta_results`, respectivamente. Estos resultados se agregan a las listas `X_sim`, `Y_sim` y `Theta_sim`.

Posteriormente, se itera nuevamente sobre cada robot. Para cada uno, se toman los resultados de la simulación y se realizan transformaciones de coordenadas. Se convierten las coordenadas (x, y) utilizando la función llamada `inverse_change_coordinates`. Además, se convierte la orientación de radianes a grados. Los resultados se agregan a listas específicas para cada tipo de valor.

Finalmente, las listas de valores se organizan en arreglos de NumPy para facilitar su manipulación. A continuación, se utiliza la función *animate* del objeto `animation_window` para iniciar la animación en la ventana.

8.1.3. Interfaz gráfica

El prototipo de la interfaz gráfica se realizó utilizando el set de librerías de PyQt5. Ejecutando la línea de comando `designer.exe` en el cmd de Windows, se abre el diseñador *drag-and-drop* (siempre estando con el ambiente virtual activo) con el que se escogieron los Widgets del prototipo de la interfaz de usuario. Para mantener la modularidad del código desarrollado, la clase de la ventana principal, únicamente fija el tamaño de la ventana y crea los objetos de cada pestaña conforme se vayan creando. En este caso se tienen tres clases más, una para cada una de las herramientas desarrolladas: Simulador, Programador, Monitoreo.

Pestaña del simulador

Siguiendo la misma estructura mencionada para el flujo de simulación, se colocaron los fragmentos de código correspondiente a cada operación general dentro de funciones en un script llamado `main_for_gui.py`. Las funciones que se definieron fueron:

- `load_world(file_path)`: carga el archivo JSON que contiene la información sobre el entorno y los robots. Devuelve un diccionario que representa el mundo.
- `initialize_animation(world)`: inicializa la ventana de animación gráfica según las dimensiones proporcionadas en el archivo JSON.

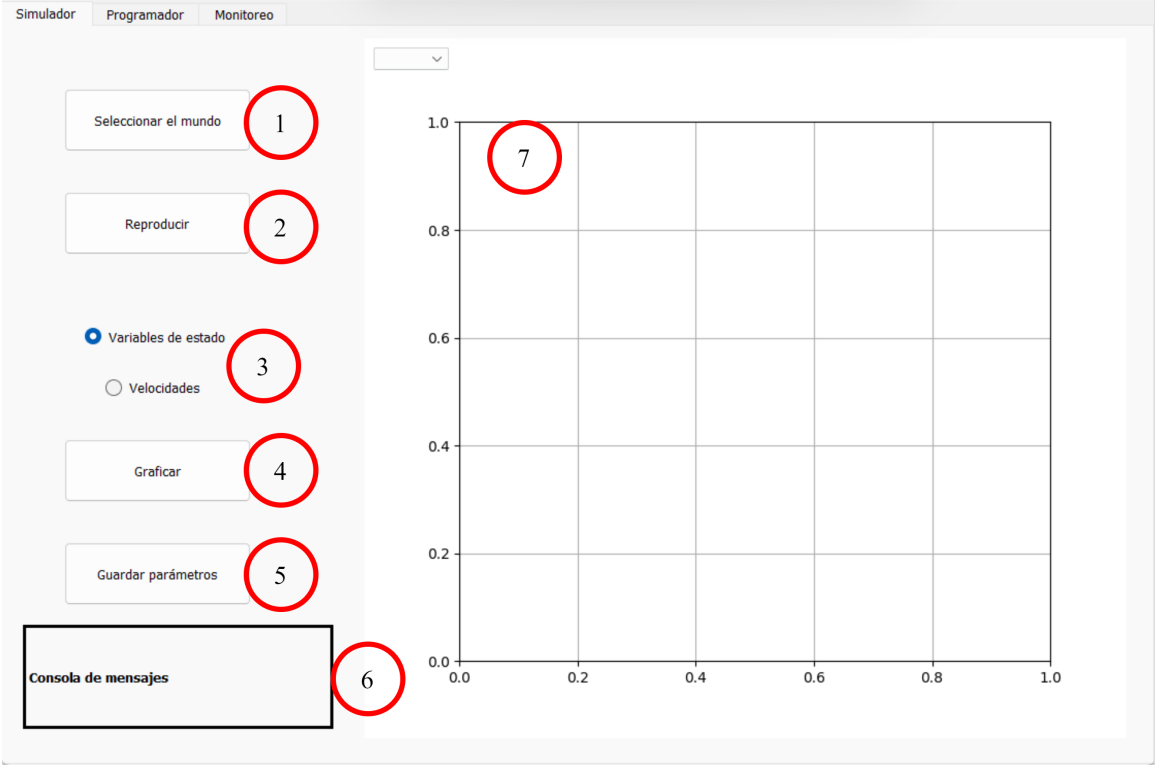

Figura 24: Pestaña de simulación en la interfaz gráfica de usuario.

- `create_objects(world, animation_window)`: crea los objetos de robots y controladores de estos, según la información del mundo y los agrega a la ventana de animación.
- `calculate_simulation(world, robots, pololu)`: realiza la simulación de los robots con los parámetros del controlador asignado y el objetivo del robot. Devuelve los resultados para su posterior visualización.
- `run_animation(animation_window, x, y, theta)`: inicia la animación con los resultados de la función anterior.

El objetivo de encapsular el código principal en funciones fue darle la capacidad a los Widgets de la GUI para instanciarlos y ejecutarlos en el orden adecuado según la interacción del usuario. El código de la GUI está directamente implementado en una clase, en la cual se inicializan sus atributos, en este caso los objetos que se crean a partir de las funciones del flujo principal de simulación. Una parte fundamental es la línea `uic.loadUi("<file_name>.ui", self)` la cual importa el ".ui" exportado del diseñador de interfaces de PyQt5, al importarlo de esta forma cada vez que se realice un cambio en el diseñador y se guarde, se actualizará en el código de Python. Los botones descritos previamente llaman a las funciones como se muestra en la Figura 27.

En la Figura 24 se puede observar una instancia del script *maingui.py*. Las opciones del usuario se describen a continuación, siguiendo la misma numeración que en la Figura mencionada:

1. Esta opción permite escoger un archivo con extensión JSON, automáticamente dirige a la carpeta de archivos de mundos de prueba del repositorio. Sin embargo, se puede escoger otro mundo creado siempre y cuando tenga la misma estructura definida anteriormente y que tenga extensión JSON para que el buscador de archivos lo despliegue entre las opciones.

2. Al presionar este botón se calculan la velocidades y los arreglos con las posiciones calculadas. Se abre automáticamente la ventana de animación de Pygame y se puede presionar el botón dentro de la ventana de animación para que comience la visualización.

3. En este paso se puede escoger si se quieren ver las velocidades o las variables de estado de la simulación (modelo del uniciclo).

4. Con este botón se grafica en el Widget de matplotlib con base en la selección anterior. Presionar este botón es opcional, ya que no tiene efecto en los resultados de la simulación, pero se recomienda para verificar que la simulación tenga congruencia con los resultados que se esperan de aplicar cierto controlador.

5. Este botón sirve para relacionar los parámetros de simulación con la pestaña de programación, se guardan en una lista los parámetros de la IP, el TAG que utiliza el robot, el controlador aplicado y punto meta a alcanzar.

6. Aquí se pueden ver mensajes útiles al usuario, entre ellos se muestra si algún botón fue presionado en orden incorrecto lo cual no permite que se ejecute la animación correctamente.

7. Finalmente, en esta ventana se grafican las velocidades y las variables de estado para cada robot según la selección de la lista desplegable (relacionados con el botón 3 y 4).

Para la parte derecha de la ventana se creó un Widget (`mplwidget.py`) que permitiera integrar gráficos de los librería `matplotlib` dentro de la pestaña de simulación, esta ventana es en la que se pueden visualizar las variables de estado y velocidades $[v, w]$ por cada robot. Ya que puede haber una cantidad variable de robots, se creó una lista desplegable que se actualizara con cada mundo, indicando el número de robot, y que al cambiar el robot seleccionado se grafiquen automáticamente sus variables de estado.

En la Figura 25 y 26 se puede ver el ejemplo de haber ejecutado una simulación para un robot, en el cual al presionar el botón de Graficar, la selección por defecto es comenzar con las variables de estado, se muestra (x, y, θ). Y en la segunda imagen se muestran las variables de la velocidad lineal y angular para el modelo uniciclo (sin haberle hecho el mapeo a rueda izquierda y derecha aún).

La implementación actual de la pestaña del simulador exige que se siga un orden para presionar los botones, de lo contrario se imprime un mensaje de error en el Widget "text label" indicando el paso que se debió haber realizado antes. El orden a seguir es como se muestra el orden de botones en la pestaña, con las descripciones mencionadas anteriormente de la Figura 24.

El caso de presionar en diferente orden se maneja en bloques *try-except* de Python dentro del mismo script que maneja las clases de la interfaz `maingui.py`. Estos son únicamente los

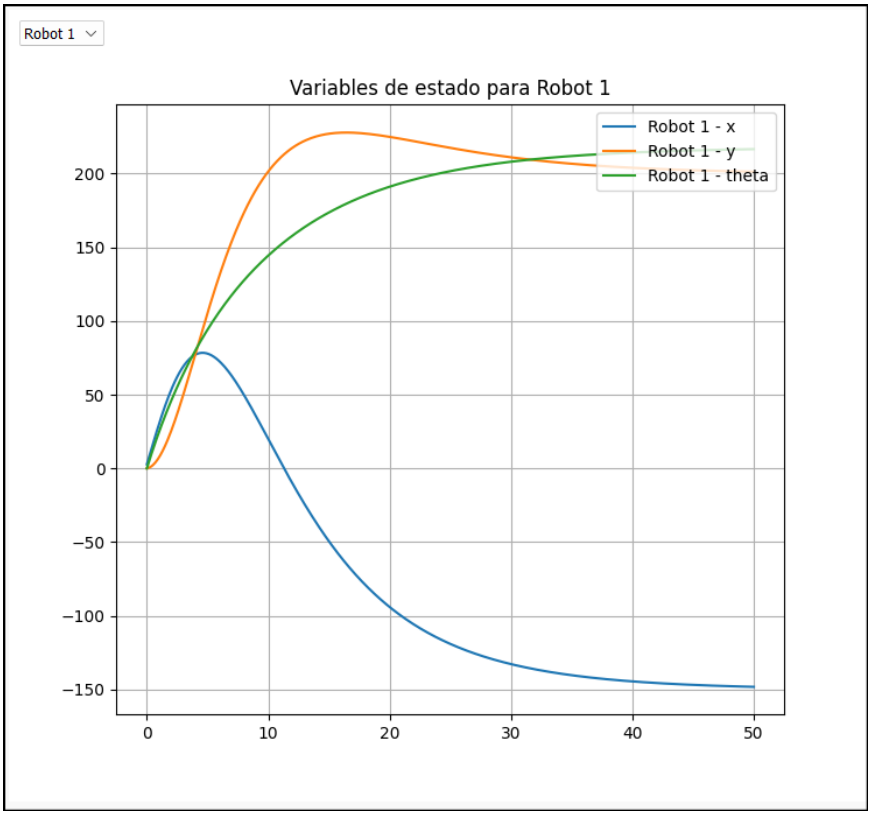

Figura 25: Variables de estado para un robot ejecutando un controlador PID.

primeros pasos a una programación defensiva, con fines de mantener el flujo de simulación, y mostrar las consideraciones que se deben tener al estar manejando múltiples posibles entradas de los usuarios.

8.1.4. Validación de simulador

En la Figura 28 y 29 se puede ver un ejemplo de haber seguido el flujo de simulación mencionado. En el caso de las figuras se simularon dos robots, uno con un control PID con acercamiento exponencial, con una meta de $[+100, +100]cm$ y el segundo con un PID y una meta de $[-50, -50]cm$. Se puede apreciar el comportamiento distinto para cada robot de forma simultánea en la animación a pesar que ambos tienen parámetros distintos.

En los vídeos Test 1 - Demostración herramientas de simulación con un agente [1] y Test 2 - Demostración herramientas de simulación con dos agentes [2] se puede ver el comportamiento de haber simulado dos mundos distintos con las funciones de la interfaz gráfica. En el primer vídeo se puede ver el caso de haber utilizado las funciones de simulación para un escenario con un solo robot ejecutando un control PID. Mientras que el segundo vídeo muestra el caso para dos robots, cada uno con un control y una meta diferente.

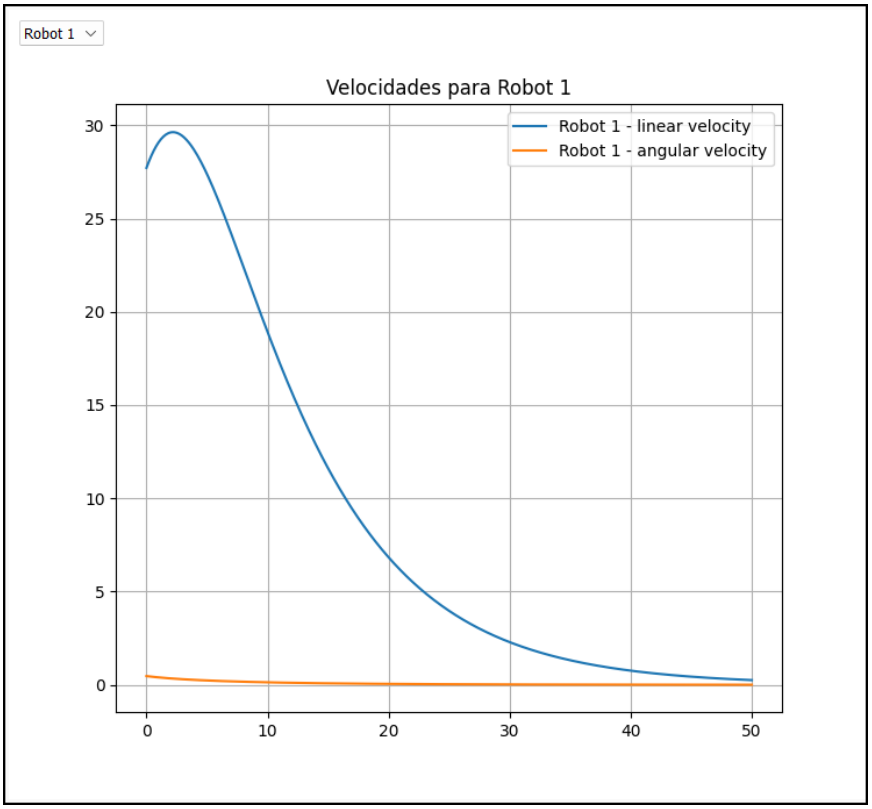

Figura 26: Velocidad lineal y angular para un robot ejecutando un controlador PID.

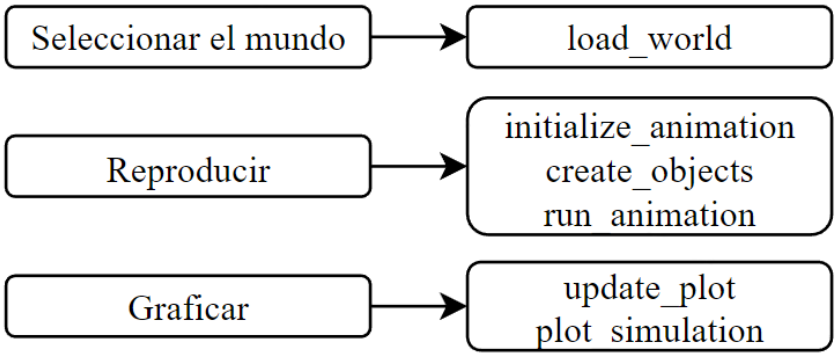

Figura 27: Funciones de GUI con funciones de flujo de simulación principal.

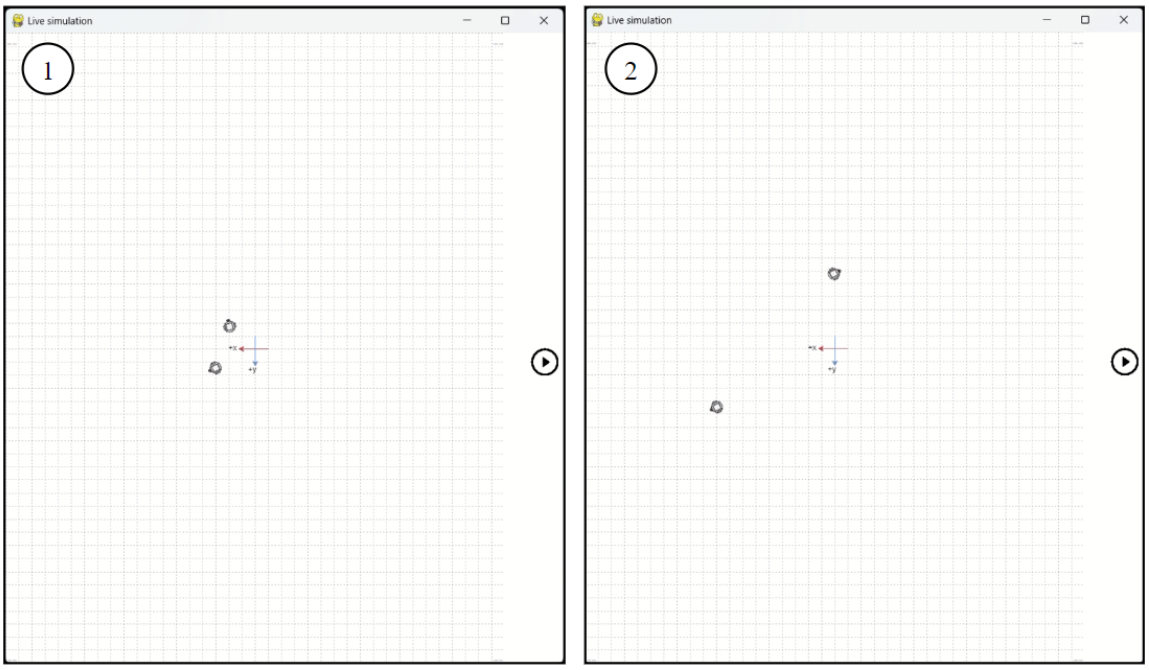

Figura 28: Ventana de animación con un escenario de ejemplo Parte 1.

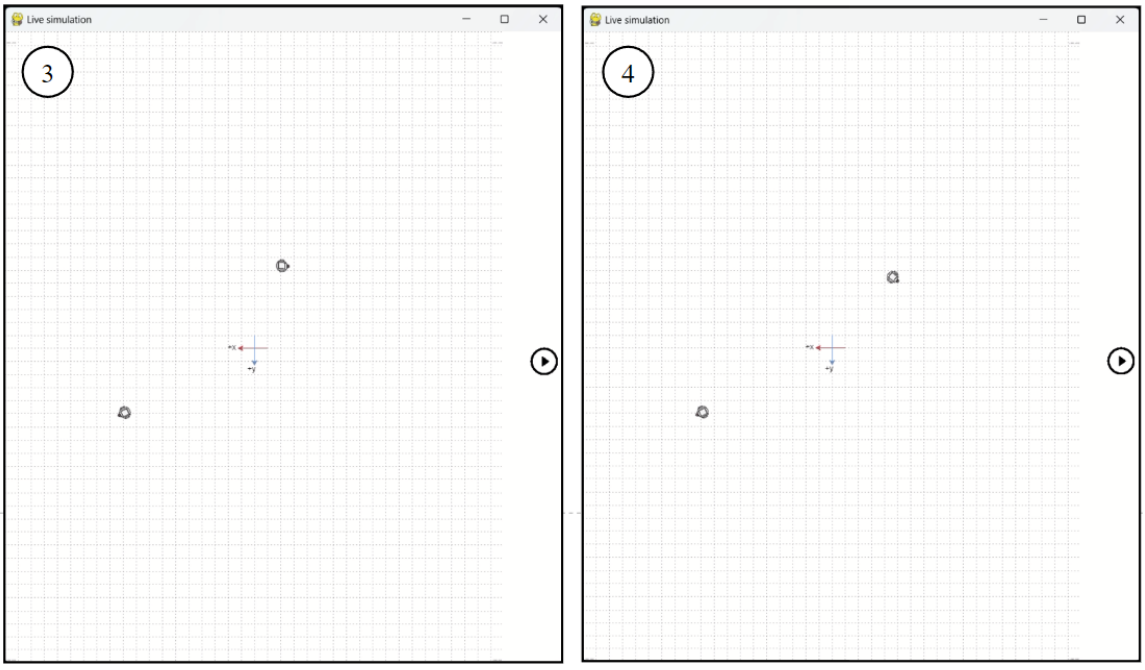

Figura 29: Ventana de animación con un escenario de ejemplo Parte 2.

CAPÍTULO 9

Implementación de OTA programming en los ESP32 de los Pololu 3Pi+

En este capítulo se describe el proceso realizado para configurar correctamente los ESP32 y que puedan recibir actualizaciones remotas utilizando la red WiFi del Robotat y librerías existentes. Se describen las alternativas exploradas para conseguir un proceso exitoso y que además, se pudiera integrar en comandos en un código de Python.

9.1. Resultados

9.1.1. Verificación de actualización de firmware

Ya que los ESP32 tienen soporte para poder recibir actualizaciones vía WiFi, se deseaba validar primero esa funcionalidad. La Arduino IDE, dentro del paquete de instalación de las placas ESP32 dentro del entorno, incluye ejemplos para probar esta característica. Primero, se debe cargar el primer código a la placa utilizando la forma convencional con cable y puerto serial COM, este primer código únicamente contiene las configuraciones de OTA utilizando la librería ArduinoOTA.h y la librería WiFi.h. En este código de configuración se escribe el identificador y la contraseña de la red WiFi en las que se estarán trabajando las actualizaciones remotas. Una vez esté cargado el programa, los ESP32 ya deberían de ser capaces de recibir actualizaciones de su firmware vía WiFi, siempre y cuando en el nuevo *sketch* a cargar mantenga los fragmentos de configuración y manejo de OTA cargados en el primer paso. Al momento de querer hacer las actualizaciones OTA, es importante que el ordenador desde donde se está enviando el programa, esté conectado a la misma red WiFi que los dispositivos ESP32 a actualizar. En el primer código cargado se puede hacer una verificación de la IP del ESP32 abriendo el monitor serial, esto es especialmente útil cuando se está trabajando con una red que asigna a una IP dinámica a los dispositivos en la red.

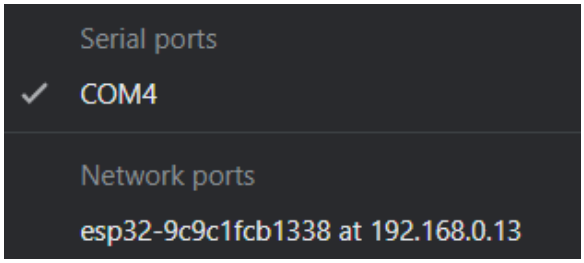

Figura 30: ESP32 identificados por su IP en Arduino IDE.

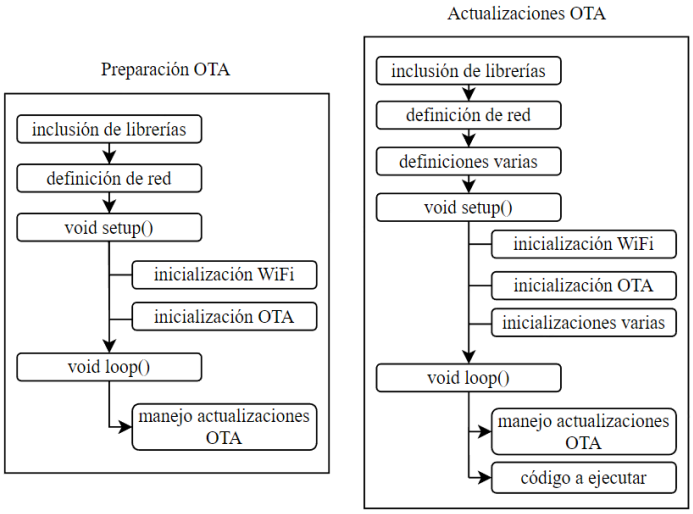

Figura 31: Estructura a seguir pre y post actualizaciones OTA con Arduino IDE

En la IDE de Arduino, el proceso de carga de un nuevo programa vía WiFi se hace bastante sencillo, ya que en la ventana en donde se selecciona el puerto de carga, aparecerá como se ve en la Figura 30, una opción de *Network ports* con los dispositivos ESP32 identificados por su IP (previamente se pudo verificar con el monitor serial que este identificador corresponda al mismo ESP32). Luego de seleccionar el dispositivo a cargar, se vuelve a presionar el botón de *Upload* en la interfaz, y hará la actualización del firmware vía WiFi. La estructura del código en Arduino, para las pruebas, se puede ver en la Figura 31. También se pueden encontrar dos sketches de Arduino en el Anexo 15.1, en el repositorio siguiendo la ruta `ota/tests/ota_prepare.ino` y `/ota_blinkingLed.in` para la configuración OTA en los ESP32 y realizar pruebas con un código básico de parpadeo de una LED con el ESP32 enviándolo por WiFi.

9.1.2. Implementación de librerías y PlatformIO

A pesar de poder validar que los ESP32 pudieran recibir actualizaciones de firmware desde la Arduino IDE y con las librerías estándar proporcionadas por Arduino, para poder integrar esta función en la interfaz y con Python para que fuera de forma autónoma, era necesario encontrar una alternativa para hacer las actualizaciones por comandos. La primera

opción analizada fue Arduino CLI que es la versión de Arduino IDE [53], sin embargo, luego de pruebas realizadas sin éxito, y revisando la documentación de Arduino, se descubrió que aún no tiene soporte para enviar por comandos las actualizaciones de firmware utilizando el protocolo identificado como *espota*. La otra alternativa que surgió fue PlatformIO, el cuál es una plataforma de desarrollo en múltiples entornos y con soporte para múltiples dispositivos embebidos, incluidos los ESP32 [54]. PlatformIO permite crear proyectos los cuales encapsulan los archivos necesarios para cada desarrollo, para los ESP32, se requería el archivo con el código en C, y las librerías para hacer OTA y la comunicación WiFi. Al estar todo en un mismo proyecto, también permite fácil acceso al archivo binario generado luego de compilar el código en C, y por lo tanto facilita cargar este binario vía WiFi al ESP32 en cuestión. PlatformIO incluía la librería de WiFi que Arduino provee de forma directa, y en [55] se tiene la versión compatible con PlatformIO de la librería ArduinoOTA.h.

Para validar la misma funcionalidad que se probó con la Arduino IDE para hacer las actualizaciones OTA pero vía comandos, se procedió a crear un proyecto nuevo de PlatformIO, se utilizó el mismo ejemplo que se había utilizado antes (`ota_prepare.ino`), haciendo el cambio de la inclusión de la librería ArduinoOTA.h por la versión compatible [55]. El proceso y la estructura se mantienen igual a lo mencionado para Arduino IDE, primero cargar el código con las funciones de manejo de actualizaciones OTA vía serial con cable, y posteriormente, agregar el código de nuevo, a las configuraciones previas, para hacer las actualizaciones vía WiFi. Las mayores diferencias a notar al utilizar PlatformIO son: ahora el código principal del ESP32 se encuentra dentro de la carpeta creada al inicializar un proyecto de PlatformIO `src/main.cpp`, la forma de compilar y hacer *Upload* al ESP32 depende de la versión de PlatformIO que se esté utilizando (PlatformIO IDE con VSCode, o PlatformIO Core CLI). Ya que lo que se deseaba era ejecutar todo desde comandos para poder integrarlo a un script de Python, se utilizó el comando `pip install platformio` lo que da acceso a las herramientas de PlatformIO como la creación de proyectos, compilación y carga de código, a través de comandos [56].

Teniendo las herramientas para compilar y cargar el firmware vía comandos instaladas, se debe preparar el archivo de configuración `platformio.ini` para que se ejecute el protocolo y el ambiente deseado durante el proceso de carga. Este archivo de configuración, permite preparar un ambiente de desarrollo, con el cual se sobre escriben las configuraciones por defecto de carga de cierto dispositivo, y declarar las que se requieren para el proyecto en específico, como protocolos de carga, baudaje, puerto de carga, entre otros [57]. De igual forma que para los sketches probados en Arduino IDE, se hicieron dos proyectos en PlatformIO, en este paso se mezcló el uso de PlatformIO IDE en VSCode para crear los proyectos y modificar el `platformio.ini` y de los comandos para hacer el proceso de compilación y carga. Mientras la estructura se mantiene igual que en la Figura 31, la diferencia yace en el archivo de configuración. Como se puede ver en la Figura 32, cada proyecto tiene un ambiente diferente. Del lado izquierdo, se tiene la configuración para hacer la carga vía serial con cable, definido como `[env:esp32dev]` y del lado derecho, con las configuraciones para utilizar el protocolo `espota`, la IP del ESP32 y la autorización (contraseña) de la red, este ambiente está definido como `[env:esp32ota]`.

Antes de hacer la integración de los comandos provistos por PlatformIO, a Python, para hacer la escritura del firmware al ESP32, se utilizó la línea de comandos mostrada en el Código 9.1 directo en la terminal de Windows. Esta línea es el análogo de lo que haría

Figura 32: Archivos de configuración pre y post actualizaciones OTA utilizando PlatformIO

Arduino IDE al presionar el botón de *Upload*, primero indica que utilizará comandos de PlatformIO `platformio`, con la directriz `run` se inicia el proceso de compilación y carga del proyecto, con `--target upload` se indica que la acción final a realizar es cargar al dispositivo, finalmente `--environment esp32dev` toma el ambiente especificado en `platformio.ini` para tomar esas configuraciones bajo las que compilará y cargará el programa. Este ambiente cambiaría de `--environment esp32dev` a `--environment esp32ota`, el primero para preparar el ESP32, y el segundo para el resto de actualizaciones que se realicen. Cabe resaltar que al ejecutar esta línea de comando 9.1, PlatformIO buscará el proyecto dentro del directorio actual, y el archivo de configuración está por defecto en la raíz del proyecto. Los proyectos de PlatformIO para realizar la prueba de preparación y posteriormente hacer una actualización OTA, se encuentran en el Anexo 15.1 en el repositorio en la ruta `ota/tests/esp32dev_ota_prepare` y `ota/tests/esp32ota_ota_update` respectivamente.

```
1 platformio run --target upload --environment esp32dev
```

Código 9.1: Upload al ESP32 vía comandos en la terminal de Windows

Para abordar el tema de buscar el directorio correcto para cargar el binario deseado en las actualizaciones OTA y su automatización, se procedió a hacer la integración con un código de prueba en Python. En este código se utilizó el módulo `os.py` para encontrar el directorio especificado, y `subprocess.py` el cual permite ejecutar comandos del sistema operativo, desde un script de Python; da las capacidades de ejecutar programas externos, controlar las entradas y salidas, entre otros [58]. Con este módulo, se pudo integrar la línea mostrada en el Código 9.1, con la mostrada en el Código 9.2.

```
1 subprocess.run(["platformio", "run", "--target", "upload", "--environment",
    "esp32dev"], cwd=sketch_directory, capture_output=True, text=True)
```

Código 9.2: Upload al ESP32 vía comandos en un script de Python

9.1.3. Robustez al proceso de conexión y programación

La integración de las actualizaciones OTA desde un script de Python permitió la inclusión de robustez al proceso previo a la carga del firmware a los ESP32. Durante los

experimentos realizados para verificar la función de OTA en los ESP32, se observó que en algunos momentos, la computadora utilizada para hacer las actualizaciones, cambiaba de red WiFi para buscar una que tuviera internet, a diferencia de la del Robotat. O que dependiendo la estabilidad de la red WiFi también se hacía el cambio. Ya que uno de los pasos principales para hacer estas actualizaciones es que ambos dispositivos (ordenador y ESP32) estén conectados en la misma red, se agregó una verificación al código de Python, para que corrobore la conexión WiFi del ordenador; en caso esta no coincida con la SSID especificada, en este caso la del Robotat, que se conecte automáticamente a esta red WiFi, y luego ya puede hacer el intento de actualizar el ESP32.

Las funciones para hacer esta verificación se encuentran en el módulo `wifi_connect.py` en la clase `NetworkManager` en el directorio `ota/wifi_connect.py` y el script de Python para realizar estas pruebas en `ota/cmd_otaTest.py`.

La función `is_connected_to_network(SSID)` permite verificar si el sistema está actualmente conectado a una red específica identificada por su SSID. Para ello, el script primero determina el sistema operativo actual utilizando `platform.system()`. Luego, busca el nombre de la red (SSID) en la salida. Esta función devuelve *True* si la red está disponible y *False* en caso contrario.

La función `createNewConnection(name, SSID, key)` es responsable de crear una nueva conexión a una red WiFi con un nombre (name), un SSID (SSID) y una clave (key). Primero, se genera un perfil XML con la configuración de la red proporcionada. Luego, se utiliza el nombre, SSID y clave proporcionados para configurar el perfil. Se ejecuta un comando para agregar el perfil de red, para Windows, utiliza `netsh wlan add profile`.

La función `connect(name, SSID)` se encarga de conectar el sistema a una red específica (SSID) utilizando un perfil de red previamente configurado con un nombre (name). Por último, la función `displayAvailableNetworks()` muestra las redes WiFi disponibles.

9.1.4. Nueva estructura del código de los Pololu 3Pi+

Como se mostró en las estructuras anteriores, hay cuatro partes claves para las actualizaciones remotas a los ESP32: mantener el código de manejo de actualizaciones OTA en todos los códigos para que pueda recibir una siguiente actualización, el archivo de configuración `platformio.ini`, la clave y SSID de la red desde el paso de preparación, y estar conectado a la misma red en ambos dispositivos. Ya que el código que ejecutan los ESP32 en el ecosistema implementan múltiples tareas en simultáneo con freeRTOS, se modificó la estructura mostrada en la Figura 31 pero únicamente es necesario para la parte de las actualizaciones. la preparación puede seguir teniendo una estructura lineal de programación sin múltiples hilos. La nueva estructura se muestra en la Figura 33 considerando también los *tasks* que los Pololu 3Pi+ deben ejecutar por implementaciones preexistentes del ecosistema, como el manejo de velocidades de ruedas, conexiones, entre otros.

En el vídeo Test 1 - Demostración actualizaciones OTA [1] se puede observar las actualizaciones OTA desde la interfaz de PlatformIO, en este caso se tiene la visualización de la ventana de monitoreo que se describe en el siguiente capítulo, la visualización del robot moviéndose sobre la plataforma, y el proceso de carga desde PlatformIO. Aquí se observa la

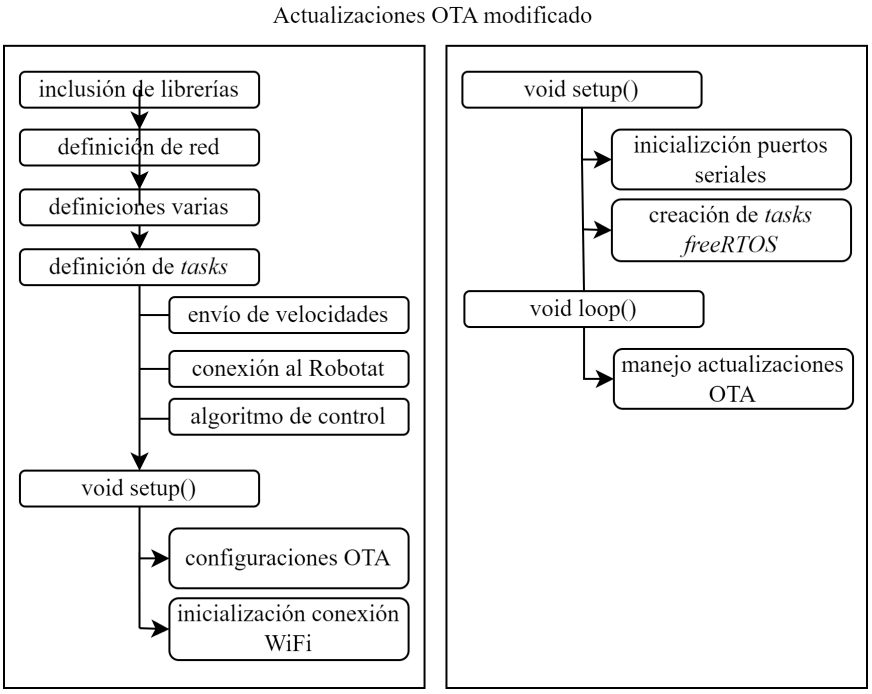

Figura 33: Nueva estructura para ejecutar un controlador en el ESP32 y manejar actualizaciones OTA.

capacidad de los ESP32 a recibir actualizaciones vía WiFi, en este caso se comenzó con una velocidad inicial en las ruedas para que el Pololu se moviera en círculos. Mientras se estaba moviendo el robot, se cambió la velocidad de las ruedas para que tuviera un radio de giro más grande. Se puede ver en el vídeo del robot físico y en la ventana de monitoreo que el robot cambia la velocidad de las ruedas al recibir la actualización. Además, con este pequeño ejemplo se demuestra que siempre y cuando el ESP32 esté encendido (tenga alimentación de voltaje) puede recibir las actualizaciones sin necesidad de estar conectado físicamente a la computadora.

9.1.5. Auto-generación de código de Python a C

Para implementar los controladores simulados en el ESP32, fue necesario desarrollar una herramienta que permitiera ajustar algunos parámetros del código **main.cpp** según las características del entorno de simulación. Para lograr esto, se empleó el módulo de Python Sympy [59]. Este módulo ofrece un paquete de funciones especializado en la "generación" de código a partir de expresiones en Python. Cabe destacar que genera el código compilable, pero no lo compila en sí mismo. Estas funciones devuelven cadenas de texto que pueden contener tanto el código como el encabezado del archivo.

En el caso de los controladores, se crearon dos archivos de Python, uno para cada controlador probado en este trabajo: el control PID y el control PID exponencial. Estos difieren de los controladores utilizados en la simulación, ya que para esta se emplean módulos numéricos para los cálculos como NumPy, mientras que la librería SymPy requiere que se utilicen

expresiones lo más simples posible para evitar dependencias o conflictos con la importación de módulos externos. Adicional a esto, las constantes en estos archivos nuevos son resultado de pruebas físicas, recomendadas como punto de partida para la calibración del controlador.

En el caso de los cálculos con funciones trigonométricas, la misma librería SymPy permite importar símbolos para las funciones **atan2, sin, cos**, que son las que se utilizan en ambos controladores implementados. En cuanto al cálculo de la norma y del exponencial, se debieron calcular de forma explícita.

La implementación de este módulo posibilitó la modificación exclusiva de las constantes relevantes para llevar a cabo pruebas en un entorno físico. Esto permitió obtener automáticamente resultados para la velocidad lineal y angular del modelo del uniciclo. De esta manera, en el código principal del ESP32, en la rutina de control, solo fue necesario realizar el mapeo a las velocidades de la rueda izquierda y derecha del Pololu. Entonces, al implementar un controlador, las dos expresiones de interés como resultado de cierto algoritmo son únicamente dos líneas correspondientes a estas velocidades. Esto hace que no se tenga que cambiar el controlador completamente en el código en C sino que al tener una estructura definida en Python, únicamente se debe calibrar el control, y para ambos controles en este caso, se obtendrán expresiones útiles.

La función del control tiene la estructura mostrada en el Código 9.3. Al tener el puntero hacia las velocidades de las ruedas en el código principal únicamente se tienen que tomar estos valores y mapearlos como se muestra en el Código 9.4.

```
#include "codegen.h"
#include <math.h>
#include <stdio.h>

volatile float v_result;
volatile float w_result;
void control(double goal_x, double goal_y, double x0, double y0, double
    theta0, double *wheel_speeds)
{
    v_result = expresion auto-generada por Sympy;
    w_result = expresion auto-generada por Sympy;

    wheel_speeds[0] = v_result;
    wheel_speeds[1] = w_result;
}
```

Código 9.3: Función de control en el ESP32 para los Pololu 3Pi+

```
control(goal_x, goal_y, x, y, yaw, temp);
v = temp[0];
w = temp[1];
phi_ell = (v - w*DISTANCE_FROM_CENTER) / WHEEL_RADIUS;
phi_r = (v + w*DISTANCE_FROM_CENTER) / WHEEL_RADIUS;
```

Código 9.4: Mapeo de las ruedas con base en la función de control

9.1.6. Validación de un controlador punto a punto ejecutándose en un ESP32 con función OTA

Para ejecutar el controlador en el mismo ESP32, se utilizó la información de la pose del marcador sobre el robot, proporcionada por el servidor a los ESP32 mediante la función `connect2robotat_task`. Esta función ofrece la pose del robot con una representación de orientación en cuaterniones unitarios. Para integrar esta información en la función de control y corregir la orientación del robot de manera adecuada, se convirtió a ángulos de Euler.

Es relevante destacar que para implementar esta función en PlatformIO (originalmente implementada en Arduino directamente), fue necesario importar la librería de ArduinoJson compatible con PlatformIO. Para esto, se modificó nuevamente el archivo **platformio.ini**, como se muestra en las dos últimas líneas del Código 9.5. Además, se observa otra librería en las dependencias, TinyCBOR, la cual corresponde a la función para enviar las velocidades de las ruedas (`encode_send_wheel_speeds_task`), ya validada en los Pololu.

```
lib_deps =
  https://github.com/JakubAndrysek/BasicOTA-ESP32-library.git
  soburi/TinyCBOR
  ArduinoJson
lib_archive = false
```

Código 9.5: Modificación del archivo de configuración del ESP32 con control

Una vez que se obtuvieron los datos correctamente formateados, se realizaron diversas pruebas con los controladores PID desde varios puntos de inicio hacia una meta $(1.0, -1.0)m$, así como con el PID con acercamiento exponencial. Estas pruebas se llevaron a cabo modificando manualmente la meta en el **main.cpp** del proyecto PlatformIO. En el vídeo Test 1 - Validación PID [1] y Test 2 - Validación PID [2] se pueden ver vídeos del ESP32 del Pololu ejecutando el control con la estructura de código mencionada anteriormente. En el primer vídeo se puede ver un control proporcional de orientación $Kp = 15.0$ con el que se ve al robot orientándose con dificultad, sin embargo, se valida con MATLAB que el agente llega a la meta especificada. En el segundo, el factor proporcional se modificó a $Kp = 10.0$ y se obtuvieron mejores resultados. Cabe resaltar que la posición de inicio aunque parece cercana en ambos casos, inicia desde un punto aleatorio en la plataforma según el espacio que había disponible, esto valida también que las constantes de control y el control mismo aplicado directamente en el ESP32 recibiendo datos del OptiTrack funcionan correctamente. Las constantes de control pueden seguirse ajustando para obtener mejores resultados. Independientemente del monitoreo de datos implementado, que se discutirá más adelante, se confirmó que los agentes alcanzaban la meta establecida, obteniendo la pose del marcador mediante funciones preexistentes de MATLAB y verificando su convergencia al punto deseado.

En el vídeo Test 3 - Validación PID con acercamiento exponencial [3] y Test 4 - Validación PID con acercamiento exponencial [4] se muestra también la validación del control PID con acercamiento exponencial con las velocidades lineal y angular generadas con SymPy, estos resultados muestran una mejor orientación como se espera de esta modificación del PID. Sin embargo, se puede observar en la posición (x, y) de MATLAB que el control aún se puede ajustar para que se acerque de forma más precisa.

9.1.7. GUI - Pestaña de programación

Para la pestaña de enviar actualizaciones OTA se dividió en dos funciones principales: cargar parámetros a partir de la información del mundo de simulación, y cargar un proyecto de forma independiente. Estas funciones cambian con base en la selección del usuario.

A partir de los parámetros de la simulación

A continuación, se muestran las funciones de la ventana de programación para la opción de cargar los parámetros a partir de algunos de los parámetros del mundo de simulación. Estos pasos se deben de realizar en el orden especificado por el mismo orden de los números en la Figura 34 y en el caso en que sea de más de un agente a programar, se debe hacer en orden uno por uno.

1. Permite seleccionar el robot al que se le quiere cargar los parámetros, esto es especialmente útil para el caso de los escenarios con más de un agente, ya que esta lista de selección cambiará según la cantidad de robots que haya en el mundo JSON de simulación, y según el número que se seleccione comenzando en cero, se cambian los parámetros mencionados en la sección 8.1.3 al presionar el siguiente botón.

2. Al presionar este botón, se modifican los parámetros de la IP, meta, y número de marker que utiliza ese agente. Para esto se definió un proyecto de PlatformIO estándar `src/ota/esp32ota_sim` el cuál se modificará con cada robot que se seleccione.

3. Este botón se dirige al archivo de preparación del ESP32 en el cual este debe estar conectado con cable para hacer la carga vía serial. En este proyecto de PlatformIO no se modifica nada ya que es un mismo archivo de preparación para todos los procesos.

4. Este botón envía la actualización vía WiFi al ESP32 con la IP que se modificó con el segundo botón. En este paso el dispositivo puede o no estar conectado, ya podría estar montado sobre el Pololu, siempre y cuando tenga alimentación podrá recibir las actualizaciones.

5. Aquí se despliegan algunos mensajes importantes y errores que se puedan dar principalmente en el orden de haber ejecutado los pasos anteriores. Sin embargo, la realimentación del proceso se da más detallada en el siguiente widget.

6. Dado que al hacer el proceso de compilación y carga de un microcontrolador en las IDEs que se suelen utilizar, como Arduino IDE o PlatformIO IDE, se obtienen mensajes del proceso, se decidió colocar esto mismo dentro de la interfaz. Al ejecutar las funciones de subprocess en los scripts de Python, se obtiene el *standard ouput* en la terminal desde donde se esté ejecutando el comando. Para tener esta información dentro de la misma interfaz se despliega en el Widget de tipo Text Browser la información que usualmente se obtiene en las IDEs, como estado de la compilación, y de subida. En el caso de la actualización OTA automáticamente se obtiene también el proceso de carga.

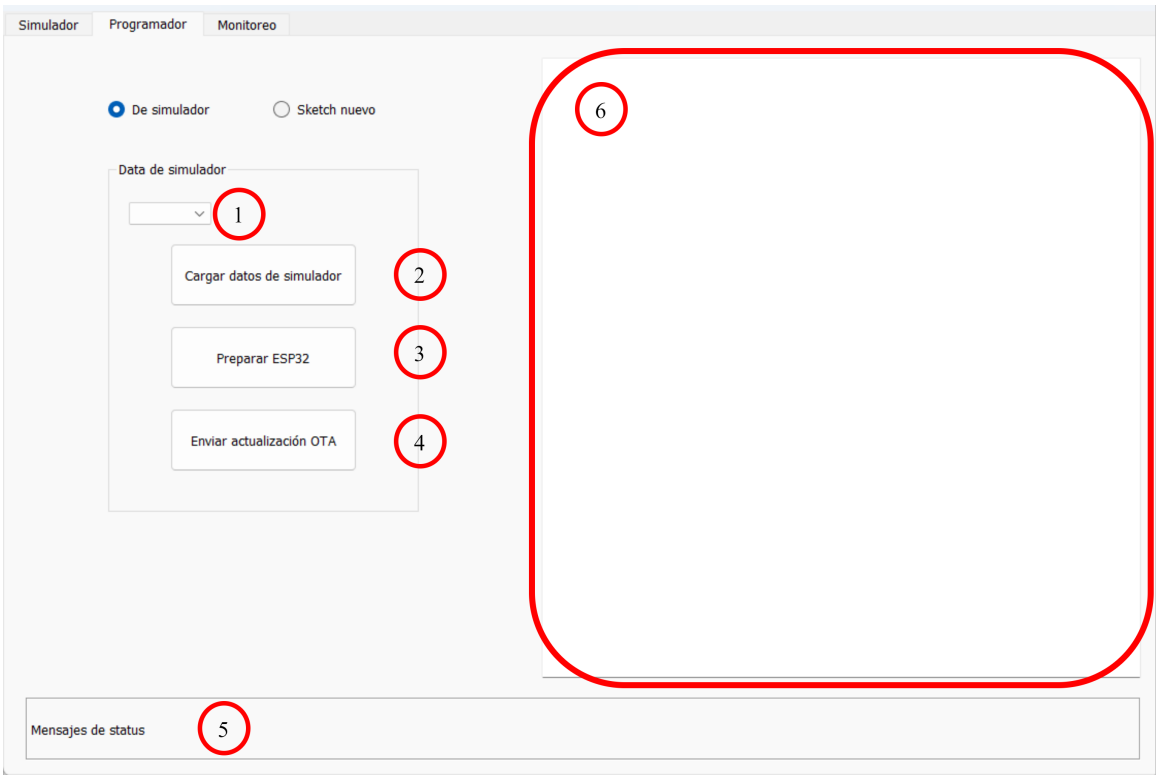

Figura 34: Ventana de programación a partir de parámetros de simulación.

A partir de un proyecto nuevo

Esta función permite seleccionar un archivo de PlatformIO con parámetros independientes de la simulación. Esto permite utilizar las herramientas de OTA en los ESP32 de los Pololu, sin tener que haber realizado una simulación previamente. Esto fue especialmente útil para realizar las pruebas de control que se mencionaron en la sección 9.1.6. En la Figura 35 se muestran las opciones para este bloque. Cabe resaltar que considerando que las opciones son para cargar el proyecto de forma independiente, se asume que los parámetros ya fueron modificados según la necesidad (velocidades, IP, tag, meta) sin embargo, aún se colocan dos parámetros para poder modificar en caso sea necesario según la disponibilidad de materiales en el Robotat:

1. Aquí se puede modificar la IP en caso se desee cambiar según el robot que se vaya a utilizar.
2. De la misma forma, si es necesario cambiar el número del marker que se va a utilizar se puede cambiar con esta lista. Idealmente, el marker debe coincidir con el número del Pololu que se va a utilizar, y a la vez con el ESP y su IP, sin embargo, por la disponibilidad de materiales, estos podrían llegar a diferir.
3. El botón de "Buscar" abre un buscador de carpetas, del cual se debe seleccionar la carpeta que contenga el proyecto de PlatformIO, sin entrar a ninguna sub-folder de la misma. Esto permite que detecte el archivo de configuración del proyecto de forma correcta.

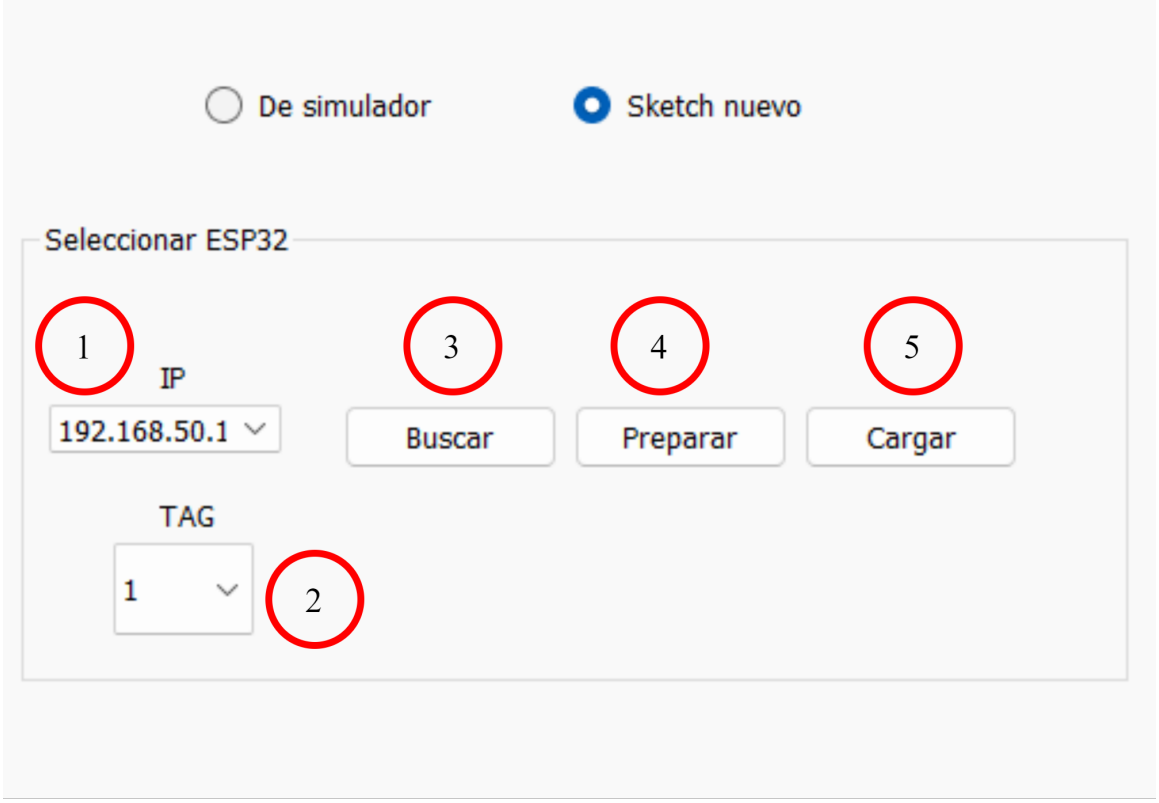

Figura 35: Ventana de programación a partir de un sketch nuevo.

4. El botón de "Preparar" se dirige a la misma carpeta por defecto que se usa en el bloque de cargar parámetros a partir del mundo de simulación. En este paso debe estar conectado el ESP32 de forma serial.

5. Hace el proceso de carga OTA. De la misma forma que el bloque de funciones descritas anteriormente, esta opción y la anterior despliegan el log en el mismo Text Browser para conocer el estado de la carga.

Las funciones para poder desplegar la información en el Widget de Text Browser se definieron en el archivo `src/ota/ota_main.py` como clases separadas. Para que se pudiera obtener el log de información en la ventana sin que se detuvieran los demás procesos de la interfaz se trabajó con QThreads [60] y la misma librería de subproccess de Python, pero esta vez en vez de enviar comandos a la terminal, para leer la salida de los procesos de la terminal. El log de las actualizaciones OTA, para conocer el proceso de carga, se encuentra en la ruta `.platformio/packages/framework-arduinoespressif32/tools/espota.py`. Este archivo fue necesario modificar el tipo de salida que tenía de *standard error* a *standard output* para que pudiera ser desplegado en el Widget.

En el vídeo Test 2 - Demostración actualizaciones OTA desde la interfaz [2] se puede ver el funcionamiento de las opciones de la interfaz de usuario para cargar un archivo de un proyecto nuevo y la información que se despliega en pantalla sobre el proceso de carga. En este caso se estaba programando un ESP32 para que parpadeara un LED, con el fin de demostrar la capacidad de recibir las actualizaciones y las opciones de usuario.

CAPÍTULO 10

Funciones en Python para el monitoreo de poses de los Pololu 3Pi+ en la plataforma del Robotat

A continuación, se describe el proceso de migrar funciones preexistentes en MATLAB a Python para conectarse al servidor del Robotat y obtener información del sistema de captura OptiTrack. Esta información permitió obtener la pose de los marcadores colocados sobre cada Pololu 3Pi+. Adicional a esto, se describe la forma de visualizar este monitoreo de datos y generar reportes a partir de la data capturada.

10.1. Resultados

10.1.1. Conexión con el sistema de captura OptiTrack con Python

Para poder monitorear el movimiento de los Pololu 3Pi+ dentro de la plataforma de pruebas del Robotat, se tiene el sistema de captura OptiTrack y el servidor del Robotat. Originalmente se tenían las funciones para conectarse al servidor del Robotat, y obtener las poses de los marcadores en la plataforma del Robotat, en MATLAB `robotat_connect/MATLAB`, a partir de estas se realizó el port a Python para su posterior integración con el resto de infraestructura. En el módulo `robotat_functions.py` se tienen las funciones `robotat_connect()` y `get_pose_continuous()`. En este mismo módulo está la migración de las demás funciones de MATLAB sin embargo, no se utilizan para el desarrollo de este trabajo.

La primera crea un socket utilizando el módulo del mismo nombre de Python, se especifica la IP de la red del Robotat para crear un cliente tipo TCP, esta función devuelve un objeto tipo `socket.socket`. Se especifica que sea de tipo TCP con los siguientes parámetros `socket.AF_INET, socket.SOCK_STREAM`.

La segunda función, recibe de parámetros el objeto tipo socket creado con la función anterior, los identificadores de los marcadores correspondientes a cada Pololu 3Pi+, y la representación de orientación que se desea obtener. La función itera sobre un rango de intentos para agregar manejo de posibles errores a la función, en cada intento se establece un tiempo de espera para la operación de recepción de información en el *socket*. Es importante establecer el tamaño del búfer de forma que se pueda capturar toda la información que reciba; se prepara un diccionario para enviar la información de solicitud al servidor y esperar la respuesta. Cuando se recibió la respuesta, se decodifica la información del sistema de captura, recibiéndolo en formato JSON, y se convierte a un arreglo de NumPy. Se puede seleccionar el formato en que se desean recibir los ángulos de la pose del sistema de captura, y se devuelven los datos obtenidos. Esta función tiene una variable que permite reintentar obtener datos del servidor, por defecto tiene diez intentos. Estos intentos se van dando cada vez que no se obtiene información del servidor por diferentes motivos, y en vez de abortar la comunicación, intenta recuperarla cada vez que haya un timeout.

10.1.2. Verificación de obtención de poses en Python

Para poder verificar los datos de las poses obtenidas en Python, con las poses que se obtienen con las funciones de MATLAB que ya estaban implementadas en el ecosistema, se colocó un marcador en la plataforma para capturar su pose. Se colocó el marcador en los cuatro cuadrantes de la plataforma. Primero se obtuvo la pose con la orientación representada en cuaterniones unitarios que es el formato que devuelve el OptiTrack por defecto, los resultados se pueden observar en el Cuadro 1. En él se muestran los resultados en MATLAB, los valores obtenidos con las funciones del port a Python, y el porcentaje de error para cada medición. Con estos resultados se pudo validar la infraestructura en Python.

Sin embargo, ya que la representación de orientación que serviría para las funciones de animación y monitoreo es en ángulos de Euler, se utilizó el módulo `squaternion` [61] para convertir los cuaterniones unitarios a ángulos de Euler. Esta conversión se aplica directamente en la función `get_pose_continuous` para que al cambiar el tipo de representación que se desea obtener, se obtenga la información ya con ese formato. Finalmente, se obtuvieron la posición (x, y) y el ángulo de rotación sobre el eje z, θ. Los resultados se pueden observar en el Cuadro 2, nuevamente con el valor de MATLAB, las obtenidas con la conversión en Python y el porcentaje de error.

10.1.3. Visualización de las poses de los Pololu 3Pi+ en una ventana de Pygame

El objetivo de monitorear continuamente la posición y orientación de un marcador en la plataforma del Robotat era poder desplegar una animación en "tiempo real", en lugar de utilizar datos precalculados como en la simulación, y obtener resultados útiles del experimento. Para lograrlo, se creó una clase llamada `py_game_monitoring` que hereda todos los atributos y métodos de la clase `py_game_animation`. Se aplicó polimorfismo en el constructor de la clase y en el método `start_animation`. Esta modificación implicó pasar una función que sirviera como fuente de los datos a desplegar en los objetos de tipo "clase robot", que son los mismos utilizados en la animación de la simulación. Para poder pasar esta función como

	x **(m)**	y **(m)**	z **(m)**	η	ϵ_x	ϵ_y	ϵ_z
Cuadrante: $(+x, +y)$							
MATLAB	1.0671	1.1934	0.0063	0.9688	0.0006	0.0049	-0.2479
Python	1.0666	1.1933	0.0064	0.9687	0.0006	0.0049	-0.2479
% error	0.04	0.008	1.58	0.96	0	0	0
Cuadrante: $(+x, -y)$							
MATLAB	1.1234	-0.8554	0.0141	0.965	-0.0059	0.0024	0.2623
Python	1.1233	-0.8553	0.014	0.9649	-0.0057	0.0027	0.2623
% error	0.008	0.01	0.7	0.01	3.38	12.5	0
Cuadrante: $(-x, +y)$							
MATLAB	-1.0809	0.8343	-0.0006	-0.0541	-0.0048	-0.0011	0.9985
Python	-1.0808	0.8343	-0.0005	-0.054	-0.0048	-0.001	0.9985
% error	0.009	0	16.66	0.18	0	9.09	0
Cuadrante: $(-x, -y)$							
MATLAB	-0.9805	-1.2966	0.0067	0.2881	-0.0061	0.0039	0.9576
Python	-0.9803	-1.2966	0.0067	0.2882	-0.0062	0.0038	0.9575
% error	0.02	0	0	0.03	1.63	2.56	0.01

Cuadro 1: Tabla de comparación de cuaterniones en MATLAB y Python.

parámetro del constructor fue necesario crearla como una función anónima en el script en el que se crea el objeto. Esta clase al recibir la función de la fuente de información, que devuelve la posición y orientación del marcador sobre la plataforma, permite capturar los datos al mismo tiempo que los despliega, al estar ejecutando el mismo método. Cabe mencionar que para hacer esta ejecución en simultáneo se investigó la librería *multiproccesing* y *threading* de Python, lo que permitiría ejecutar dos tareas en paralelo y capturar los datos del OptiTrack y animar la ventana. Sin embargo, estas alternativas no fueron aplicables ya que la ventana de Pygame no es *thread-safe* lo que llega a dar problemas en la ejecución de la ventana con otras tareas en paralelo.

Para visualizar la actualización de la pose de un marcador en la plataforma, se sigue la estructura descrita en el pseudocódigo mostrado en el Algoritmo 2. En la ruta `src/monitoring` del repositorio, se puede ejecutar un script con estas funciones de forma independiente a la interfaz para realizar pruebas (`monitoring_display_test.py`).

10.1.4. Generación de reportes de las poses de los Pololu 3Pi+ utilizando el sistema de captura OptiTrack

Por el momento, el servidor del Robotat proporciona información sobre la actualización de la pose de los marcadores montados en los Pololu 3Pi+. De esta pose lo que interesa para las aplicaciones de los robots móviles sobre ruedas es la posición (x, y) y la orientación

	x **(m)**	y **(m)**	θ $(\circ)$
Cuadrante:$(+x, +y)$			
MATLAB	1.0670	1.1934	-28.7133
Python	1.0671	1.1929	-28.7187
% error	0.009	0.04	0.01
Cuadrante:$(+x, -y)$			
MATLAB	1.1234	-0.8554	30.4124
Python	1.1233	-0.8553	30.4022
% error	0.008	0.01	0.03
Cuadrante: $(-x, +y)$			
MATLAB	-1.0810	0.8344	-173.8000
Python	-1.0806	0.83397	-173.7923
% error	0.009	0.05	0.0044
Cuadrante: $(-x, -y)$			
MATLAB	-0.9804	-1.2966	146.4894
Python	-0.9804	-1.2966	146.5000
% error	0	0	0.007

Cuadro 2: Tabla de comparación de Euler en MATLAB y Python.

θ. Otra modificación que se agregó a la clase hija de monitoreo fue agregar la generación de un archivo CSV utilizando el módulo del mismo nombre de Python [62]. La información a guardar en el CSV proviene de la misma función fuente que devuelve las posiciones y orientación para la ventana de animación. Para la animación en tiempo real los datos pasan primero por un proceso de mapeo para que puedan ajustarse a la escala de la ventana de animación como se ve en el Pseudocódigo 2, sin embargo, para los datos del CSV se pasan los datos en el formato que lo devuelve el servidor, ya convertido la orientación, a ángulo de Euler sobre el eje z.

Para validar este informe generado, se utilizó un Pololu 3Pi+ con su marcador montado. Se configuraron las velocidades de las ruedas izquierda y derecha para que el robot móvil diera una vuelta completa. Luego, se colocó el robot en cada uno de los cuadrantes para generar un archivo CSV para cada uno. Se verificó que la trayectoria circular coincidiera con lo observado en la plataforma física, la ventana de monitoreo de Pygame y la gráfica generada a partir de las dos primeras columnas del archivo CSV. Se hizo una plantilla para hacer la gráfica del CSV de forma que coincidiera con la visualización de la ventana de Pygame (recordando la perspectiva que tiene la ventana de simuluación con respecto a la plataforma física). Para utilizar esta plantilla debe copiarse de la carpeta de `docs` y colocarlo en la carpeta de plantillas de Office para poder utilizarlo y tener la visualización correcta. En la Figura 36 se puede ver un ejemplo de haber puesto a girar a un Pololu en el cuarto cuadrante $(+x, -y)$ con la plantilla creada, se puede ver que las coordenadas de la gráfica que coinciden con la visualización de la ventana de simulación y de la monitoreo con respecto

Algoritmo 2: Funciones para el monitoreo de la pose de un marcador en la plataforma del Robotat

```
Iniciar la conexión con el servidor del Robotat
Inicializar arreglos para almacenar datos
Inicializar arreglos para la visualización de datos
for Cada dato de posición recibido desde el robotat do
    Procesar los datos y agregarlos a los arreglos correspondientes
Mapear los datos para su visualización en la ventana de animación
Inicializar variable run_animation a Verdadero
Obtener y procesar los datos en tiempo real
Mapear los datos para su visualización
Retornar los datos procesados
Crear una instancia de la ventana de animación animation_window
Agregar el personaje del robot al animation_window
Inicializar animation_window
while Verdadero do
    Iniciar la animación en animation_window
```

a la orientación de los ejes.

10.1.5. Validación de resultados del monitoreo

En los vídeos Test 1 - Monitoreo de PID exponencial [1] y Test 2 - Monitoreo de PID exponencial [2] se puede observar a un Pololu ejecutando un control punto a punto con un PID con acercamiento exponencial para llegar a la meta $(1.0, -1.0)m$. En estos vídeos en vez de validar que haya llegado al punto deseado se hace la comparación del movimiento del robot con el obtenido en la ventana de Pygame. Para ambos casos se puede observar la misma trayectoria ejecutada para llegar desde un punto inicial aleatorio hasta la meta deseada (robot físico y personaje del robot en Pygame). En el vídeo Test 3 - Monitoreo en el primer cuadrante y CSV [3] se muestra la ventana de monitoreo mientras el robot da un círculo en el primer cuadrante de la plataforma del Robotat, y se muestran los datos capturados en el archivo CSV. Al graficar los resultados con la plantilla generada para que tenga la misma perspectiva de visualización que la ventana de Pygame, se puede obtener la trayectoria circular que ejecutó el robot. Se obtienen los resultados de la forma en que los proporciona el servidor, es decir los que aún no están mapeados para su representación escalada en la ventana de Pygame. Esto muestra una correcta captura de datos durante la ejecución del experimento, en los momentos en que ocurría timeout se pueden ver pequeños saltos en los datos graficados, pero con los reintentos se logró restablecer la comunicación y seguir recibiendo datos.

10.1.6. GUI - Pestaña de monitoreo

Para monitorear algún marker de forma independiente se recomienda utilizar el mismo script utilizado para las pruebas (`monitoring_display_test.py`), ya que en este es más práctico cambiar el número de marker a utilizar en las pruebas de forma independiente. Para

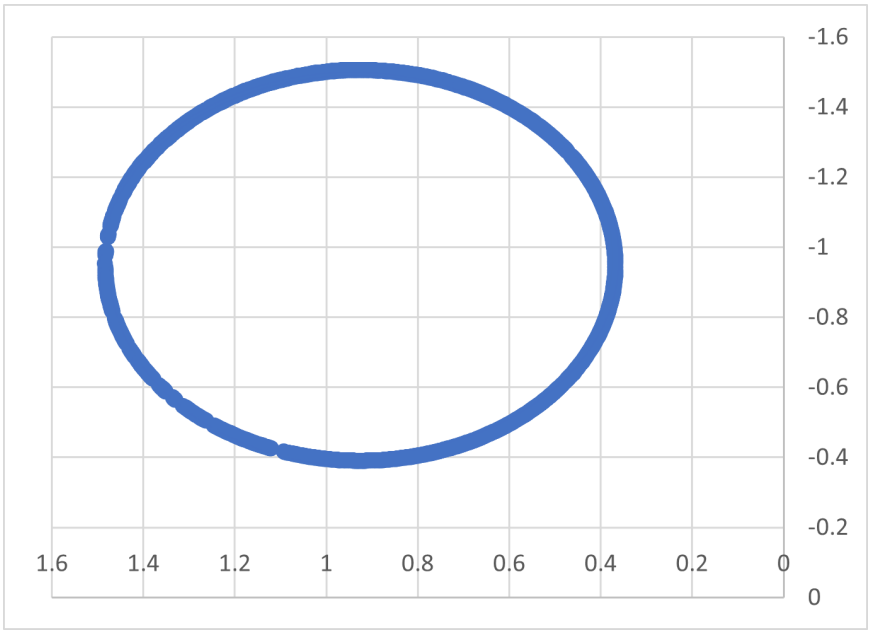

Figura 36: Gráfica generada a partir de los datos capturados del servidor del Robotat.

integrar las funciones de monitoreo en la interfaz y permitir que se relacione con las herramientas de simulación y programación, se trabajó un módulo aparte: `monitoring_main.py`. En este módulo se creó una clase con métodos para obtener y procesar los datos recibidos de las funciones de conexión con el servidor. También están incluidos los métodos para hacer el mapeo de los valores para su despliegue en la ventana de animación, y posteriormente inicializar la ventana de monitoreo. Las opciones de usuario para la pestaña de monitoreo se muestran en la Figura 37. Estas se describen a continuación:

1. Se debe escribir el nombre con el que se guardará el archivo CSV para ese experimento. Además en el botón al lado del nombre del archivo se puede seleccionar la ubicación para guardar el archivo.
2. Al presionar el botón de grabar se abrirá la ventana de animación que se actualiza con la información del marcador. Automáticamente se comienza a escribir la data de la pose del marker en el archivo CSV. Para terminar de grabar la información se debe cerrar la ventana de monitoreo.
3. En esta consola se desplegarán mensajes importantes con respecto a los pasos que se deben seguir o aspectos a considerar durante la captura de datos.
4. Adicional a esto se agregó una imagen de un marker del OptiTrack para indicar la forma en que se debe colocar el marker sobre el robot para que la ventana de monitoreo de Pygame tenga el display correcto de la imagen del robot. Esta se puede observar en la Figura 38.

Cabe resaltar que en el código del monitoreo estas funciones reciben el identificador del marker que se va a monitorear a partir de los parámetros guardados de la simulación, es decir, en el mismo archivo JSON que se crea para la simulación se deberá modificar el número del tag que se utilizará en el robot físico.

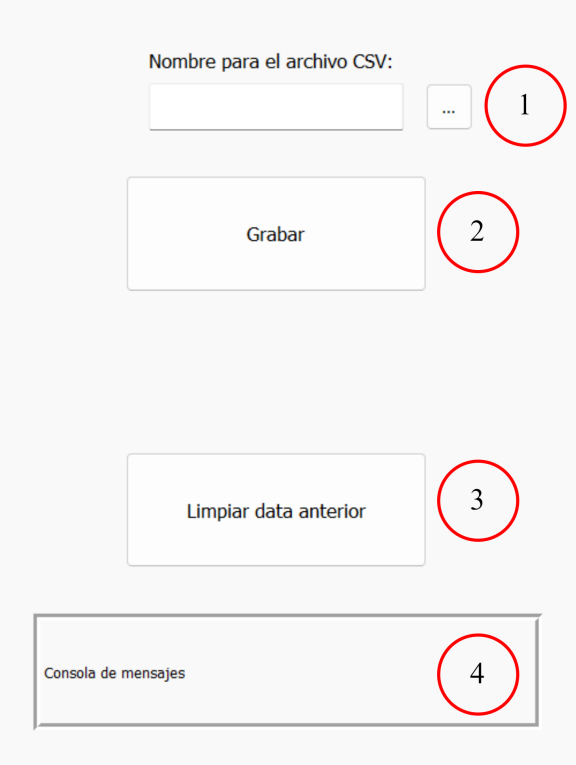

Figura 37: Opciones de usuario para la pestaña de monitoreo.

10.2. Validación de las herramientas de simulación, programación y monitoreo integradas

A continuación, se muestran los resultados de haber ejecutado un flujo de experimentación completo con las herramientas desarrolladas. Es decir, se simularon los mundos JSON creados para validación de las herramientas, se programaron los ESP32 con base en los parámetros necesarios de la simulación, y finalmente se monitorearon los datos de la pose del marker colocado sobre el robot con el servidor del Robotat. En los vídeos se muestra la coincidencia aproximada del comportamiento del controlador hacia la meta deseada, considerando que se están comenzando en puntos aproximados al inicial de la simulación y las constantes del controlador calibradas para la plataforma física, no se esperaba que tuvieran la misma trayectoria para llegar al punto deseado pero si el mismo comportamiento y tendencia a la meta establecida. En cada uno de los vídeos se puede ver primero los parámetros importantes de la simulación, así como su descripción en el archivo JSON. Luego se hace énfasis en la modificación de las velocidades lineales y angulares a partir del controlador especificado en el JSON, y otros parámetros que se modifican en el código del ESP32. Finalmente, se muestra la ventana de monitoreo y el robot físico funcionando en simultáneo.

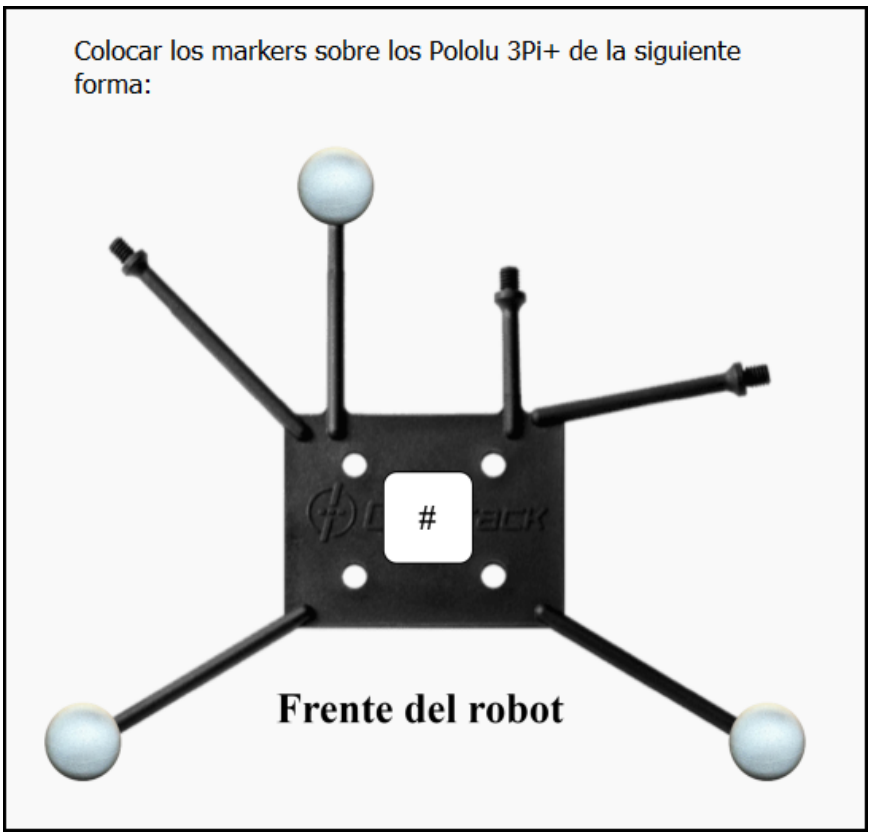

Figura 38: Cómo se debe colocar el marker sobre el robot.

10.2.1. Escenarios de prueba para un agente

Los mundos simulados se encuentran en la carpeta `src/worlds`. Estos mundos se describen a continuación:

- world_1robot_pid:

  El controlador aplicado al robot es un PID hacia una meta $(-1.50, 2.00)m$. Los resultados se pueden ver en el vídeo Test 1 - Escenario *world_1robot_pid.json* [1].

- world_1robot_pidExp:

  El controlador aplicado al robot es un PID con acercamiento exponencial hacia una meta $(1.00, -2.00)m$. Los resultados se pueden ver en el vídeo Test 2 - Escenario *world_1robot_pidExp.json* [2].

10.2.2. Modificaciones a las funciones de monitoreo para el caso multi-agente y primeras pruebas

Para dotar a las herramientas de monitoreo de mayor versatilidad de aplicación, se comenzó con la modificación de las funciones de lectura de pose del servidor del Robotat para que devolviera de forma correcta dos conjuntos de datos. De igual forma se tuvo que

modificar las funciones de obtener y procesar datos, así como la de mapeo para la ventana de Pygame, para que manejara información de múltiples marcadores. Estas funciones se encuentran en la misma clase mencionada para su integración con la interfaz. La función que se utiliza para animar el monitoreo se selecciona con base en la cantidad de robots que se hayan simulado previamente. En general, las herramientas de simulación ya tienen implementadas las opciones para manejar múltiples agentes. En el caso de las herramientas de programación, también son capaces de modificar los scripts del ESP32 en el caso haya múltiples robots, ya que el proceso se debe hacer uno por uno. La modificación mayor fue en las funciones de monitoreo para que la información fuera procesada de forma simultánea para dos marcadores al recibir la información del servidor. Estas modificaciones se dejan como el punto de partida para monitorear más de un agente a la vez, y seguir experimentando con los tiempos de recepción, el manejo de errores y mayor funcionalidad que se le pueda dar al tener mayor información de la plataforma. Estas funciones se pudieron validar con los mundos simulados descritos a continuación:

- world_2robots_pid:

  En este escenario se quiso demostrar cargar el mismo controlador a dos robots diferentes con metas diferentes. El primer robot con meta $(-1.50, -2.0)m$ y el segundo con meta $(1.50, 2.0)m$. Los resultados se pueden ver en el vídeo Test 3 - Escenario *world_2robots_pidExp.json* [3].

- world_2robots_pid_n_exp:

  En este caso se simularon dos robots funcionando con diferentes controladores, uno con un PID normal y el otro con PID con acercamiento exponencial. El primero con meta $(0.0, 0.0)m$ y el segundo con meta $(1.50, 1.50)m$. Los resultados se pueden ver en el vídeo Test 4 - Escenario *world_2robots_pid_n_exp.json* [4].

CAPÍTULO 11

Punto de partida de desarrollo de herramientas de ROS2 para los agentes Pololu 3Pi+ en el ecosistema del Robotat

En este capítulo, se establece un punto de partida para el camino que se puede seguir en futuras implementaciones de ROS2 en el ecosistema del Robotat. Se describe el proceso realizado para ejecutar un paquete de comunicación demostrativo, entre microcontrolador y ordenador, con la arquitectura de ROS2. El objetivo de estas pruebas fue explorar las capacidades de las herramientas existentes y comprender los pasos esenciales para su correcta ejecución. Además, se mencionan algunos pasos importantes para llevar a cabo la comunicación entre ambos dispositivos, asumiendo siempre el entorno de una máquina virtual.

11.1. Ejecución de un paquete de comunicación entre ESP32 con micro-ROS y ordenador con ROS2

Para implementar ROS2 en los agentes robóticos Pololu 3Pi+ dentro del ecosistema del Robotat, es esencial tener en cuenta las herramientas disponibles, tanto a nivel de software como de hardware. Dado que los agentes robóticos utilizan actualmente un microcontrolador ESP32-WROOM-32, resulta apropiado utilizar el port de micro-ROS para este dispositivo.

Con el fin de demostrar la funcionalidad de estos paquetes de librerías y la arquitectura base de ROS con los ESP32, se procedió a la instalación de VirtualBox con Ubuntu 22.04 LTS, ROS2 Humble (actualmente, la distribución con la que micro-ROS es compatible), micro-ROS y PlatformIO en la máquina virtual. La prueba de comunicación consistió en incrementar un contador en el ESP32 y recibir este valor constantemente en otro nodo ejecutado en ROS2 de forma nativa, para implementar una aplicación de tipo *publisher-listener*. A continuación, se detallan los pasos para ejecutar un nodo *publisher* en un ESP32 y un nodo *listener* en un ordenador con sistema operativo Ubuntu.

11.1.1. Compilación del paquete de micro-ROS para ESP32

En el Código 11.1 de la documentación de micro-ROS [35] se detalla el proceso a seguir para la instalación de micro-ROS con la distribución de ROS2 Humble. Es importante destacar que se debe agregar la creación de un Agente de micro-ROS que es el que permite la configuración del ESP32 con esta versión. Además, se debe hacer *source* tanto a la instalación de ROS2 (en caso no esté en el *bashrc*) y la instalación de micro-ROS para que se puedan ejecutar las funciones de los nodos en la terminal.

```
# Source the ROS 2 installation
source /opt/ros/humble/setup.bash

# Create a workspace and download the micro-ROS tools
mkdir microros
cd microros
git clone -b humble https://github.com/micro-ROS/micro_ros_setup.git src/
    micro_ros_setup

# Update dependencies using rosdep
sudo apt update && rosdep update
rosdep install --from-paths src --ignore-src -y

# Build micro-ROS tools and source them
colcon build
source install/local_setup.bash

# Download micro-ROS-Agent packages
ros2 run micro_ros_setup create_agent_ws.sh

# Build step
ros2 run micro_ros_setup build_agent.sh
source install/local_setup.bash
```

Código 11.1: Proceso de instalación y creación de agente de micro-ROS

11.1.2. PlatformIO con micro-ROS

Para facilitar la configuración del ESP32 con los paquetes de micro-ROS, se recomienda utilizar PlatformIO, este entorno ya cuenta con una librería [63] para el desarrollo del firmware del microcontrolador. El archivo de configuración de PlatformIO tiene como configuración base lo mostrado en el Código 11.2. En el archivo principal del proyecto de PlatformIO se debe configurar el nodo del ESP32, en este caso, el editor. Cuando ya se tiene configurado, el firmware se carga al microcontrolador vía Serial.

```
[env:esp32doit-devkit-v1]
platform = espressif32
board = esp32dev
framework = arduino
monitor_speed = 115200
lib_deps =
    https://github.com/micro-ROS/micro_ros_platformio
```

```
board_microros_distro = humble
```

Código 11.2: Archivo de configuración del proyecto PlatformIO para aplicaciones de micro-ROS en ESP32

Resolución de problemas

Al trabajar en una máquina virtual, es crucial detectar el puerto serial al que está conectado el ESP32. Se determinó que los permisos de VirtualBox pueden estar mal configurados, lo que causó problemas al intentar detectar el ESP32. Para solucionarlo, se debe presionar, en la ventana de VirtualBox, *Devices*, *USB*, y deberá aparecer el dispositivo del ESP32 disponible. Si la casilla no está marcada, debe seleccionarse para que Ubuntu lo reconozca. Luego, en la terminal de Ubuntu, se deben ejecutar los comandos mostrados en el Código 11.3.

```
$ sudo adduser <username> dialout
$ sudo chmod a+rw /dev/ttyUSB0
```

Código 11.3: Permisos a Ubuntu para detectar el puerto serial del ESP32

Con estas configuraciones en el editor de PlatformIO en VSCode ya se puede seleccionar manualmente el puerto de carga, y realizar la operación de carga de firmaware - *Upload*.

11.1.3. Resultados

En la Figura 40, se muestran los resultados del *publisher* que corresponde al ESP32. Para que el nodo del ESP32 comience a funcionar, es necesario ingresar la línea de comando que se muestra en el Código 11.4. En este caso, se muestra principalmente información sobre el proceso de inicialización del editor. Al presionar el botón de Reset del ESP32, se inicia el ciclo que se observa como "session established" en adelante. El nodo `/micro_ros_platformio_node` comienza a enviar datos a través del tópico `/micro_ros_platformio_node_publisher`, como se puede apreciar en el gráfico generado por el comando `rqt_graph`, que se muestra en la Figura 41.

En la Figura 39, se pueden observar la ejecución del *echo* del nodo editor, el cual despliega en la terminal lo que el ESP32 envía de forma constante.

```
ros2 run micro_ros_agent micro_ros_agent serial --dev /dev/ttyUSB0
```

Código 11.4: Comando para iniciar el nodo del ESP32 en la terminal del ordenador

Se incluyó un archivo de Python con las herramientas de ROS2 Humble para que actuara como el segundo nodo en la transacción *publisher-listener*. Este se ejecuta en la terminal al ingresar el comando que se muestra en el Código 11.5. Por lo tanto, la gráfica de nodos ahora muestra dos nodos y un tópico de por medio, como se puede observar en la Figura 42. Además, se puede ver la comparación entre lo que está siendo publicado por el nodo del

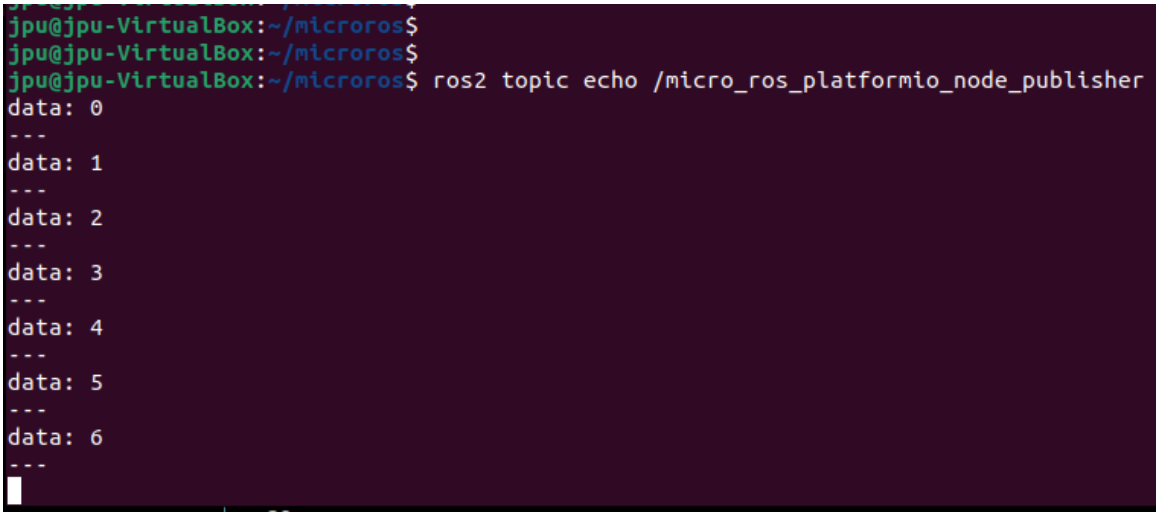

Figura 39: Echo del editor ejecutándose en el ESP32, incremento constante del contador.

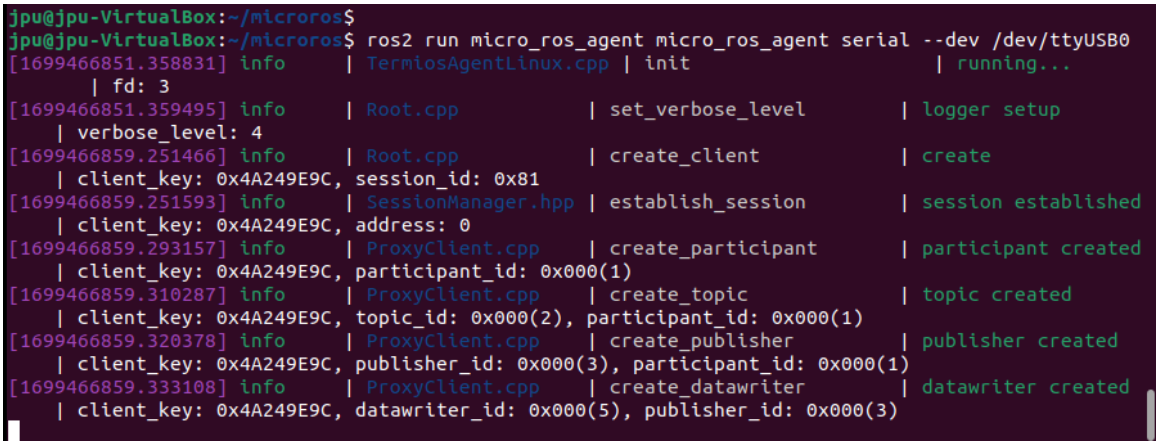

Figura 40: Proceso de ejecución del nodo del ESP32 mostrado en la terminal de Ubuntu.

ESP32 y lo que es recibido por el nodo de Python en la Figura 43. Esto reafirma el concepto de que ROS2 actúa como un intermediario, permitiendo la comunicación entre dispositivos que sigan la arquitectura base de los paquetes. En este caso, el ESP32 fue programado en un lenguaje distinto (C++) al suscriptor en el ordenador (Python).

```
1 python3 listener.py
```

Código 11.5: Comando para iniciar el nodo de Python con ROS2 en la terminal

11.2. Propuesta de implementación de ROS2 y micro-ROS en el Robotat

Después de la investigación realizada y las pruebas de comunicación descritas anteriormente, se puede concluir que las herramientas de ROS y micro-ROS ofrecen un potencial

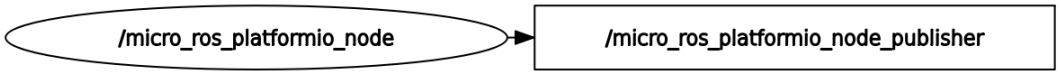

Figura 41: Rqt_graph del editor sin un nodo de recepción, únicamente nodo y tópico.

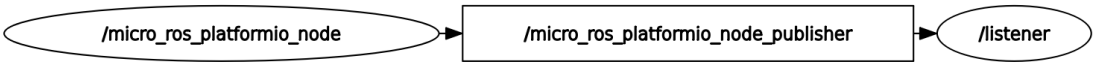

Figura 42: Rqt_graph del nodo editor publicando en el tópico, y un segundo nodo suscriptor.

prometedor para su implementación en el ecosistema del Robotat y en los agentes Pololu 3Pi+. Esto se debe a las capacidades del ESP32 y el soporte que micro-ROS brinda para este microcontrolador. Se sugiere explorar opciones específicas para el ESP32, como los medios de transmisión, ya que dispone de un módulo WiFi y Bluetooth. Se proponen pruebas para enviar datos desde el nodo del ESP32 aprovechando la infraestructura de software del Robotat.

En la Figura 44, se presenta el diagrama base propuesto para su implementación en los Pololu 3Pi+. En las pruebas realizadas, el ESP32 actuó como editor. Sin embargo, para la implementación en el Robotat, los Pololu 3Pi+ deberían tener una estructura más compleja, ya que tendrán que desempeñar roles de editores y suscriptores en diferentes momentos de su ejecución. Se propone un servidor en un ordenador, con un enfoque de acción, encargado de gestionar tareas como trayectorias, haciendo uso de tópicos y servicios para actualizar el estado del robot. El agente robótico actuaría como cliente en este escenario.

Además, se contemplan tópicos normales para el envío constante de datos, como velocidades de las ruedas o información de sensores, entre otros. En este contexto, los Pololu 3Pi+ actuarían como editores y el ordenador como suscriptor. Los círculos azules representan la implementación en micro-ROS, considerando su ejecución en ESP32, mientras que los verdes indican la implementación en ROS2 Humble, ejecutándose en un sistema operativo Linux Ubuntu.

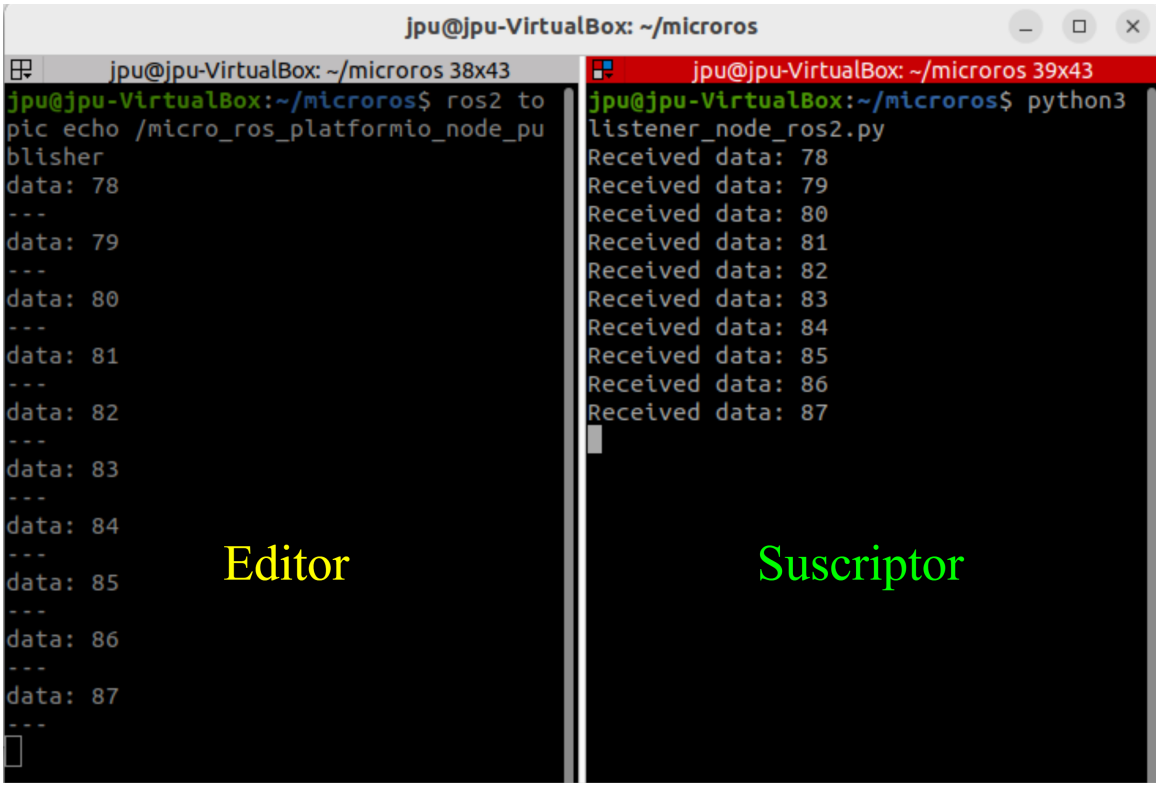

Figura 43: ESP32 publicando y suscriptor recibiendo la data en ROS2.

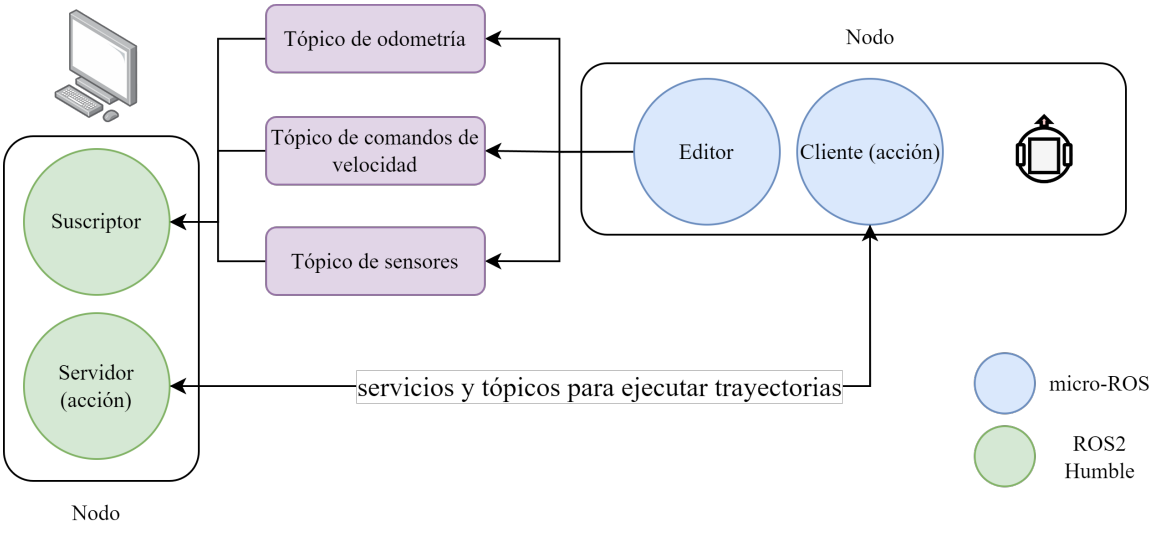

Figura 44: Diagrama de propuesta inicial de arquitectura de ROS en el Robotat.

CAPÍTULO 12

Conclusiones

- Durante la implementación del simulador, se empleó un enfoque basado en archivos JSON para la definición detallada de los escenarios de experimentación. Esto proporcionó una gran flexibilidad para adaptar la simulación a distintos entornos y configuraciones específicas. Se tomó la decisión de utilizar la librería de Pygame para la animación de la simulación, lo cual dio flexibilidad para manejar la actualización de posición y orientación de la representación del Pololu 3Pi+.
- La implementación de actualizaciones *over-the-air* para los ESP32 que utilizan los Pololu 3Pi+ representa un avance significativo en el proceso de actualización de firmware. Esta mejora facilita la experimentación y modificación continua de los escenarios de prueba de manera más eficiente. Se han establecido los pasos adecuados para llevar a cabo actualizaciones vía WiFi en los ESP32, lo que permite que este proceso sea replicable en la red WiFi del Robotat. Tras evaluar diversas alternativas para integrar las actualizaciones OTA en los ESP32, se concluyó que PlatformIO ofrece el entorno más versátil y adecuado para llevar a cabo este proceso desde distintas herramientas, incluyendo Python.
- La modificación de los parámetros y una nueva propuesta de código para los ESP32 a partir de los parámetros del mundo de simulación permite el manejo de actualizaciones OTA y la ejecución de un control punto a punto desde el mismo ESP32. Se pudo aprovechar las capacidades del servidor y de los ESP32 al recibir la pose del marcador desde el sistema de captura de OptiTrack para tener un mejor control sin tener que depender de la actualización de la pose actual desde un ordenador ejecutando funciones en MATLAB o Python.
- La auto-generación de código en C a partir de un script base de Python para los controladores es un paso que permite una mayor modularidad en el manejo de un flujo de experimentación completo. Al tener que obtener siempre el mismo tipo de parámetro (velocidad lineal y angular) para cualquier controlador aplicado se puede implementar cualquier estructura de control que retorne este tipo de valores. Esto se validó con el control PID y PID con acercamiento exponencial, al manejar diferentes parámetros

dentro del controlador, ambos son capaces de retornar (v, w) y que el ESP32 maneje la misma estructura para el caso con PID y el caso con PID Exponencial.

- Se logró una exitosa migración de las funciones originales de MATLAB a Python para la conexión al servidor del Robotat y la obtención de las poses de los marcadores en la plataforma de pruebas. Este proceso de portabilidad es fundamental para la integración completa de la infraestructura. Se verificó la precisión y consistencia de los datos obtenidos en Python en comparación con los resultados previos obtenidos en MATLAB. Esta comparación mostró la validación de la migración a Python al obtener un bajo porcentaje de error.

- La visualización de la pose de los marcadores en una ventana de animación es el primer paso para un monitoreo remoto de lo que está ocurriendo en la plataforma física del Robotat. La generación del reporte de la posición y orientación del robot durante el experimento, en un archivo CSV permite analizar posterior al experimento la trayectoria ejecutada por el robot, lo que permite una experimentación más íntegra del comportamiento del robot.

- Se pudo implementar un código base en un ESP32 con el paquete de librerías de micro-ROS para que se comunicara con un ordenador ejecutando el paquete de librerías de ROS2 Humble. Con la ejecución de un nodo *listener* y otro *publisher* se puede visualizar la capacidad de una transacción exitosa entre distintos dispositivos. El paquete de micro-ROS demuestra ser una valiosa herramienta a explorar debido a su práctica implementación en microcontroladores. Esto muestra las capacidades de estos conjuntos de herramientas para una futura integración al ecosistema del Robotat utilizando el hardware actual de los Pololu para aplicar varios de los conceptos de comunicación de ROS2 con los agentes robóticos.

CAPÍTULO 13

Recomendaciones

- Se recomienda llevar a cabo una serie de pruebas en el simulador de los Pololu 3Pi+, abordando una variedad de condiciones y algoritmos. Esto permitirá validar su integración efectiva en la estructura de software y detectar posibles puntos vulnerables. Posteriormente, se recomienda aumentar la programación defensiva y manejo de errores para mejorar la robustez del sistema.
- Es crucial establecer límites claros en cuanto a las opciones que el usuario puede modificar a nivel de programación. Esto no solo contribuirá a evitar configuraciones erróneas, sino que también proporcionará una experiencia más intuitiva y controlada.
- Se sugiere considerar la implementación de métodos que permitan la adaptabilidad de la ventana de animación según las preferencias de visualización del usuario. Esto aseguraría una experiencia personalizada y cómoda al interactuar con la aplicación.
- Explorar las opciones de OTA directo en el framework de ESP-IDF. En esta tesis se limitó a trabajar con las herramientas de Arduino siguiendo la infraestructura que ofrecía el Robotat previamente en cuanto a su implementación de conexión con el servidor y las librerías preexistenes para la implementación de OTA. Sin embargo, podría ser valioso explorar ESP-IDF al ser el framework nativo para ESP32. Además, al explorar otras herramientas se sugiere explorar la implementación de actualizaciones múltiples de forma simultánea.
- Para darle una aplicación mayor a las funciones de monitoreo se debe seguir desarrollando el monitoreo de múltiples marcadores en la plataforma. Esto requiere especial atención al estar manejando múltiple información en tiempo real, se deben determinar los límites de información que puede manejar al mismo tiempo y que la ventana de animación logra soportar. Además, la cantidad de datos capturados y el tráfico de datos para manejar otras rutinas en el servidor pueden ser de vital importancia para un correcto funcionamiento.
- Implementar una estructura más robusta de la auto-generación de código con base en las capacidades que se desean tener. Esto contribuiría a la modularidad del programa;

se debe buscar dar más opciones a nivel usuario y explorar las herramientas de autogeneración de código para el microcontrolador.

- Al momento de implementar la estructura propuesta de ROS para el Robotat, se recomienda comenzar ejecutando el mismo script realizado en este trabajo. Con esto se podrán validar que el sistema operativo Ubuntu y otras características de la comunicación básica estén funcionando correctamente. Posterior a esto, se debe ir construyendo la estructura desde lo más básico que serían los nodos y tópicos, hasta llegar a la implementación de servicios y acciones. Se deben delimitar bien las funciones que se van a implementar considerando las limitantes de hardware y software para el ESP32 y los Pololu.

CAPÍTULO 14

Bibliografía

[1] S. Wilson, P. Glotfelter, L. Wang et al., "The Robotarium: Globally Impactful Opportunities, Challenges, and Lessons Learned in Remote-Access, Distributed Control of Multirobot Systems," *IEEE Control Systems Magazine*, vol. 40, n.º 1, págs. 26-44, 2020. DOI: 10.1109/MCS.2019.2949973.

[2] S. Jones, E. Milner, M. Sooriyabandara y S. Hauert, "DOTS: An Open Testbed for Industrial Swarm Robotic Solutions," n.º arXiv:2203.13809, mar. de 2022, arXiv:2203.13809 [cs]. dirección: http://arxiv.org/abs/2203.13809.

[3] N. Dhanaraj, J. Maffeo, G. Pereira et al., "Adaptable Platform for Interactive Swarm Robotics (APIS): A Human-Swarm Interaction Research Testbed," *2019 19th International Conference on Advanced Robotics (ICAR)*, DOI: 10.1109/ICAR46387.2019.8981628. dirección: https://par.nsf.gov/biblio/10136382.

[4] C. Perafán, "Robotat: un ecosistema robótico de captura de movimiento y comunicación inalámbrica," 2021.

[5] M. Casini, A. Garulli, A. Giannitrapani y A. Vicino, "A Remote Lab for Experiments with a Team of Mobile Robots," *Sensors*, vol. 14, n.º 9, págs. 16 486-16 507, 2014, ISSN: 1424-8220. DOI: 10.3390/s140916486. dirección: https://www.mdpi.com/1424-8220/14/9/16486.

[6] D. Chaos, J. Chacón, J. A. Lopez-Orozco y S. Dormido, "Virtual and Remote Robotic Laboratory Using EJS, MATLAB and LabVIEW," *Sensors*, vol. 13, n.º 2, págs. 2595-2612, 2013, ISSN: 1424-8220. DOI: 10.3390/s130202595. dirección: https://www.mdpi.com/1424-8220/13/2/2595.

[7] M. Kulich, J. Chudoba, K. Kosnar, T. Krajnik, J. Faigl y L. Preucil, "SyRoTek—Distance Teaching of Mobile Robotics," *IEEE Transactions on Education*, vol. 56, n.º 1, págs. 18-23, 2013. DOI: 10.1109/TE.2012.2224867.

[8] A. K. Azad, *Internet accessible remote laboratories: Scalable e-Learning tools for engineering and science disciplines: Scalable e-Learning tools for Engineering and Science disciplines*. IGI Global, 2011.

[9] OEIT, "iLabs," en-US, *iCampus*, dic. de 2011. dirección: https://icampus.mit.edu/projects/ilabs/.

[10] S. Tzafestas y M. Others, *Web Based Control and Robotics Education*. sep. de 2009, ISBN: 978-90-481-2505-0. DOI: 10.1007/978-90-481-2505-0.

[11] I. Gustavsson, J. Zackrisson y T. Olsson, "Traditional Lab Sessions in a Remote Laboratory for Circuit Analysis," abr. de 2023.

[12] "WebLab-Deusto," dirección: https://weblab.deusto.es/website/.

[13] en-US, ago. de 2016. dirección: https://wyss.harvard.edu/technology/programmable-robot-swarms/.

[14] S. Luo, J. Kim y B.-C. Min, "Asymptotic Boundary Shrink Control With Multirobot Systems," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 52, n.º 1, págs. 591-605, 2022. DOI: 10.1109/TSMC.2020.3003824.

[15] *Pololu 3pi+ 32U4 User's Guide*, Pololu Corporation, https://www.pololu.com/docs/pdf/0J83/3pi_plus_32u4.pdf.

[16] *ESP32*, 2.1, Espressif Systems, https://www.espressif.com/sites/default/files/documentation/esp-wroom-02u_esp-wroom-02d_datasheet_en.pdf, mayo de 2023.

[17] *Protocolos*, https://www.ibm.com/docs/es/aix/7.1?topic=systems-protocols: IBM, mar. de 2021.

[18] *Protocolos TCP/IP*, https://www.ibm.com/docs/es/aix/7.2?topic=protocol-tcpip-protocols: IBM, abr. de 2021.

[19] *Protocolo de control de transmisiones (Transmission Control Protocol)*, https://www.ibm.com/docs/es/aix/7.1?topic=protocols-transmission-control-protocol: IBM, mar. de 2021.

[20] *User Datagram Protocol*, https://www.ibm.com/docs/es/aix/7.1?topic=protocols-user-datagram-protocol: IBM, mar. de 2021.

[21] *Protocolo de transmisión de control de corriente (Stream Control Transmission Protocol)*, https://www.ibm.com/docs/es/aix/7.1?topic=protocol-stream-control-transmission: IBM, mar. de 2021.

[22] H. Hall, *What does OTA mean? [Part I]*, https://ubuntu.com/blog/what-does-ota-mean: Ubuntu, ago. de 2022.

[23] *Over The Air Update (OTA)*, Espressif Systems, https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/ota.html, 2023.

[24] F. C. P. Kevin M. Lynch, *Modern Robotics. Mechanics, planning, and control*. dic. de 2019, págs. 522-530, ISBN: 9781107156302.

[25] M. Zea, *Control de robots móviles con ruedas*, Cátedra en Universidad del Valle de Guatemala, Guatemala, mayo de 2023.

[26] Open Robotics, *ROS2 Documentation: Foxy*, https://docs.ros.org/en/foxy/index.html: Open Robotics, 2023.

[27] Open Robotics, *Distributions - ROS2 Documentation: Rolling Ridley*, https://docs.ros.org/en/rolling/Releases.html: Open Robotics, 2023.

[28] Open Robotics, *Concepts - ROS2 Documentation: Foxy*, https://docs.ros.org/en/foxy/Concepts.html: Open Robotics, 2023.

[29] O. Robotics, *Understanding nodes*, 2023. dirección: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html.

[30] O. Robotics, *Interfaces*, 2023. dirección: https://docs.ros.org/en/humble/Concepts/Basic/About-Interfaces.html.

[31] T. Target, *IDL (interface definition language)*, 2023. dirección: https://www.techtarget.com/whatis/definition/IDL-interface-definition-language.

[32] O. Robotics, *Topics*, 2023. dirección: https://docs.ros.org/en/humble/Concepts/Basic/About-Topics.html.

[33] O. Robotics, *Services*, 2023. dirección: https://docs.ros.org/en/humble/Concepts/Basic/About-Services.html.

[34] O. Robotics, *Actions*, 2023. dirección: https://docs.ros.org/en/humble/Concepts/Basic/About-Actions.html.

[35] micro-ROS, *micro-ROS*, 2023. dirección: https://micro.ros.org/.

[36] E. Systems, *ESP32-DevKitC*, 2023. dirección: https://www.espressif.com/en/products/devkits/esp32-devkitc.

[37] E. Systems, *ESP32-WROOM-32E/32UE*, 2023. dirección: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32e_esp32-wroom-32ue_datasheet_en.pdf.

[38] I. Amazon Web Services, *FreeRTOS*, 2023. dirección: https://www.freertos.org/index.html.

[39] F. Finocchiaro, *micro-ROS porting to ESP32*, 2020. dirección: https://micro.ros.org/blog/2020/08/27/esp32/.

[40] micro-ROS, *First micro-ROS Application on Linux*, 2023. dirección: https://micro.ros.org//docs/tutorials/core/first_application_linux/.

[41] P. S. Foundation, *General Python FAQ*, 2023. dirección: https://docs.python.org/3/faq/general.html#what-is-python.

[42] D. Malan, *OOP and Inheritance*, 2011. dirección: https://ocw.mit.edu/courses/6-00sc-introduction-to-computer-science-and-programming-spring-2011/resources/lecture-11-oop-and-inheritance/.

[43] C. Hock-Chuan, *Object-oriented Programming (OOP) Basics*, 2020. dirección: https://www3.ntu.edu.sg/home/ehchua/programming/java/J3a_OOPBasics.html.

[44] P. S. Foundation, *Glossary - virtual environment*, 2023. dirección: https://docs.python.org/3/glossary.html#term-virtual-environment.

[45] P. S. Foundation, *Virtual Environments and Packages*, 2023. dirección: https://docs.python.org/3/tutorial/venv.html?highlight=virtual%20environments.

[46] P. organisation, *Pygame documentation*, 2023. dirección: https://www.pygame.org/wiki/about.

[47] P. S. Foundation, *Socket*, 2023. dirección: https://docs.python.org/3/library/socket.html?highlight=socket#module-socket.

[48] *FreeBSD*, 2023. dirección: https://www.freebsd.org/es/.

[49] OnionBulb, *TCP socket flow diagram*, 2010. dirección: https://realpython.com/python-sockets/.

[50] R. Computing, *PyQt5*, 2023. dirección: https://pypi.org/project/PyQt5/.

[51] R. Computing, *Qt Designer*, 2023. dirección: https://www.riverbankcomputing.com/static/Docs/PyQt5/designer.html.

[52] P. S. Foundation, *NumPy*, 2023. dirección: https://docs.python.org/3/library/subprocess.html.

[53] *Arduino CLI (Command Line Interface) Application*, https://www.arduino.cc/pro/software-pro-cli/: Arduino.

[54] *What is PlatformIO?* https://docs.platformio.org/en/latest/what-is-platformio.html: PlatformIO Labs.

[55] K. Andrýsek, *Basic OTA library for ESP32 and VSCode - PlatformIO*, https://github.com/JakubAndrysek/BasicOTA-ESP32-library, nov. de 2020.

[56] *PlatformIO Core - Terminal*, https://platformio.org/install/cli: PlatformIO Labs.

[57] *"platformio.ini" (Project Configuration File)*, https://docs.platformio.org/en/latest/projectconf/index.html: PlatformIO Labs.

[58] P. S. Foundation, *Subprocess — Process management*, 2023. dirección: https://docs.python.org/3/library/subprocess.html.

[59] S. D. Team, *Codegen*, 2023. dirección: https://docs.sympy.org/latest/modules/utilities/codegen.html.

[60] T. Q. C. Ltd., *QThread*, 2023. dirección: https://doc.qt.io/qtforpython-6/PySide6/QtCore/QThread.html.

[61] K. Walchko, *Squaternion module*, 2014. dirección: https://pypi.org/project/squaternion/.

[62] P. S. Foundation, *Python CSV*, 2023. dirección: https://docs.python.org/3/library/csv.html.

[63] micro-ROS.org, *micro-ROS library for Platform.IO*, 2023. dirección: https://github.com/micro-ROS/micro_ros_platformio.

CAPÍTULO 15

Anexos

15.1. Repositorio de desarrollo en Github

El repositorio se encuentra dividido en tres carpetas principales: `microros`, `src` y `docs`. La primera es la que contiene el código fuente para la ejecución de micro-ROS y ROS2, está en un folder separado ya que no tiene relación directa con las otras herramientas desarrolladas. El segundo folder contiene el código fuente de todas las herramientas y pruebas desarrolladas (simulación, programación y monitoreo). Finalmente, el tercer folder contiene documentación generada, como guías, archivos utilizados para generar imágenes y la plantilla para el reporte de datos CSV de la ventana de monitoreo. En la ruta principal del repositorio también se encuentra el archivo `requirements.txt` desde donde se pueden instalar todas las dependencias de Python. Estos detalles de instalación se dan en el archivo README.md del mismo repositorio.

La carpeta `src` tiene el código principal de la interfaz gráfica de usuario, y las herramientas divididas en diferentes directorios. Hay nueve carpetas:

- `controllers`: Contiene los archivos `.py` en donde se define cada controlador para la simulación.
- `monitoring`: Contiene el módulo del port de funciones de MATLAB a Python para la conexión al servidor del Robotat. También están las funciones principales para su integración con la interfaz gráfica y las pruebas realizadas.
- `ota`: Tiene las pruebas a ejecutar para probar el funcionamiento OTA de los ESP32 y el módulo de funciones de Python para su integración con Python. Aquí se incluyen los proyectos base de PlatformIO para realizar pruebas, preparación del ESP, modificaciones con respecto a la simulación, y también los controles base para la generación de código en C.
- `pictures`: Aquí se guardan todas las imágenes que se usan tanto en la interfaz, como la representación de los robots, la cuadrícula de la mesa de pruebas, entre otros.

- `robots`: Define las funciones relacionadas a la clase Robot.
- `tests_py`: Para mantener separados los códigos de pruebas individuales de ciertas funciones del simulador.
- `windows`: Contiene los módulos relacionados a la ventana de animación de Pygame, como el mapeo y las ventanas de Pygame de monitoreo.
- `worlds`: Como recomendación, aquí se deben guardar los archivos JSON que se quieran simular.
- `ui_files`: Para mantener la modularidad del código se tiene la definición generada por el PyQt5-designer en un directorio separado y únicamente se importan al código principal.
- Externo a estos directorios está el código principal de la interfaz gráfica integrando las funciones principales de los demás directorios.

`https://github.com/pu19249/ROBOTAT-Tools`

15.2. Enlaces a vídeos demostrativos y explicativos

Se realizaron vídeos demostrando las herramientas funcionando de forma individual, y en conjunto. Los enlaces están en una playlist de YouTube `https://www.youtube.com/playlist?list=PLmwuGdxB4AbiVyYBOB0Snp6qic_pKbkJL`, de la cual los vídeos corresponden a la siguiente organización:

15.2.1. Demostración de herramientas de simulación desde la interfaz

1. Test 1 - Demostración herramientas de simulación con un agente:
   `https://youtu.be/N-AmiSBQZ-M`
2. Test 2 - Demostración herramientas de simulación con dos agentes:
   `https://youtu.be/LfES_uyosCU`

15.2.2. Validación controladores ejecutándose en ESP32

1. Test 1 - Validación PID:
   `https://youtu.be/VuZ3mWiC4YM`
2. Test 2 - Validación PID:
   `https://youtu.be/ghoj36XUSY8`
3. Test 3 - Validación PID con acercamiento exponencial:
   `https://youtu.be/V9G8n7q3jQM`
4. Test 4 - Validación PID con acercamiento exponencial:
   `https://youtu.be/-Hr8AhBWwPY`

15.2.3. Demostración de OTA Updates en ESP32

1. Test 1 - Demostración actualizaciones OTA:
   `https://youtu.be/rkUwNwZPbS8`
2. Test 2 - Demostración actualizaciones OTA desde la interfaz:
   `https://youtu.be/JxKPBOprASM`

15.2.4. Demostración de la herramienta de monitoreo con Pygame

1. Test 1 - Monitoreo de PID exponencial:
   `https://youtu.be/sXAcQe8rg3Y`
2. Test 2 - Monitoreo de PID exponencial:
   `https://youtu.be/64evhpefP6A`
3. Test 3 - Monitoreo en el primer cuadrante y CSV:
   `https://youtu.be/Oj3Ea-bPb64`

15.2.5. Flujo de experimentación completo

1. Test 1 - Escenario *world_1robot_pid.json*:
   `https://youtu.be/Qfz9tkbjw4Q`
2. Test 2 - Escenario *world_1robot_pidExp.json*:
   `https://youtu.be/CaTRE_BAJIO`
3. Test 3 - Escenario *world_2robots_pidExp.json*:
   `https://youtu.be/QRjYcmd-sbM`
4. Test 4 - Escenario *world_2robots_pid_n_exp.json*:
   `https://youtu.be/DXY9iQqnsLA`

CAPÍTULO 16

Glosario

API: Application Programming Interface, son los procesos y funciones que brinda una determinada librería de programación. 22

DPI: Puntos por pulgada (dots per inch). En este contexto, se refiere a la resolución de la pantalla. 29

drag-and-drop: En general hace referencia a un sistema de arrastrar y soltar. En el contexto de GUIs arrastrar y soltar componentes para organizar la interfaz. 24

lambda: En el contexto de la programación con Python, estas funciones tipo lambda se refieren a funciones anónimas. Estas se pueden pasar como argumentos de otras funciones. 37

landmarks: Distintas características que un robot puede reconocer con diferentes métodos de entrada. 36

mantenibilidad: La facilidad con la que un sistema puede ser mantenido o modificado después de su implementación. 28

Modularidad: El enfoque de diseñar y organizar un sistema en partes independientes y reutilizables. 25

Runge Kutta 4: Un método numérico utilizado para resolver ecuaciones diferenciales. 34

ruta relativa de archivos: Especificación de la ubicación de un archivo en relación con el directorio actual. 29

SSID: Service Set Identifier. Es el nombre con el que se puede identificar una red local. 48

timeout: Se refiere a alcanzar un límite establecido de tiempo antes que se interrumpa alguna conexión de comunicación, servicio u otro. Es decir, si se establece un período de tiempo de internet, la conexión existirá hasta que se alcance este tiempo ocurriendo el "timeout". 56

Widgets: Son las partes de una GUI que permite al usuario accesar a funciones de la aplicación y/o sistema operativo de forma abstraida. Permiten la interacción del usuario con la aplicación. 22