Proyecto Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Programación, Control e Implementación de un Robot Móvil de Seguimiento Reactivo basado en Visión Artificial y ROS

Autor: Diana Sotelo Castillo

Tutores: Sergio Luis Toral Marín

Samuel Yanes Luis

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2023

ACE-TI | Aplicaciones Cibernéticas de la Electrónica a las Tecnologías de la Información

Proyecto Fin de Grado
Grado en Ingeniería de Tecnologías Industriales

Programación, Control e Implementación de un Robot Móvil de Seguimiento Reactivo basado en Visión Artificial y ROS

Autor:
Diana Sotelo Castillo

Tutores:
Sergio Luis Toral Marín
Catedrático
Samuel Yanes Luis
Investigador

Dpto. Ingeniería Electrónica
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2023

Proyecto Fin de Grado: Programación, Control e Implementación de un Robot Móvil de Seguimiento Reactivo basado en Visión Artificial y ROS

Autor: Diana Sotelo Castillo
Tutores: Sergio Luis Toral Marín
Samuel Yanes Luis

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

A mi abuela, quien me enseñó a esforzarme por mi mañana.
Quien me inculcó el amor por las matemáticas.
Y, sobretodo, quien siempre me quiso como una madre.

Agradecimientos

De todo corazón, debo agradecer primero a mi abuela, quien me ha otorgado su apoyo incondicional y me ha apoyado en este camino, sin ella no sólo no habría terminado este trabajo, sino que no sería la persona que soy hoy. A mis padres, que tanto han aguantado durante estos años, sin su cuidado y paciencia jamás habría llegado tan lejos. A mis amigos y a mi pareja, que me han entendido cuando no podía salir de casa, en los momentos en los que debía leer, escribir, corregir e investigar, y aún así me han obligado a salir cuando sabían que necesitaba aire fresco. Gracias por la paciencia, haber estado ahí para escucharme y por los abrazos. También quiero agradecerle a mi tutor de Trabajo Final de Grado y a los profesores con los que he compartido horas en esta Universidad. De todos he aprendido algo y los llevaré siempre conmigo.

Diana Sotelo Castillo

Escuela Técnica Superior de Ingeniería de Sevilla.

Sevilla, 2023

Resumen

Este Trabajo de Fin de Grado aborda la implementación de un sistema de seguimiento reactivo en un robot diferencial. Para ello se hace uso de un robot de configuración diferencial basado en la CPU Jetson Nano llamado Jetbot. Este robot se monta y se prepara con un sistema operativo Ubuntu y varias librerías para el control del robot. Tras el montaje, se hace una evaluación de las capacidades del robot y se calibran los motores. Además, se desarrolla un programa que posibilita la teleoperación del robot para realizar pruebas. Se programan nuevas librerías acorde a los resultados de estas pruebas iniciales. El robot pasa por varias iteraciones en sus componentes, que se cambian para ajustarse a las necesidades del proyecto.

Una vez preparada la plataforma, se programa una aplicación que ponga a prueba la capacidades del robot. Esta es programada en Python, un sistema de seguimiento reactivo en tiempo real basado enteramente en visión y desarrollado con las herramientas del software Robot Operating System (ROS) en mente. La aplicación primero se prueba en un entorno de simulación, utilizando el simulador CoppeliaSim, donde se realizan diversos experimentos exitosos que confirman el funcionamiento de los algoritmos desarrollados. Despues de que los experimentos en simulación son un éxito, se implementa la aplicación en el Jetbot y se desarrolla un programa de apoyo para calibrar los algoritmos de visión artificial.

Finalmente, se realizan experimentos de la aplicación desarrollada en el robot, con unos resultados que no llegan a ser satisfactorios. Sin embargo, la experiencia obtenida y el diseño del sistema en sí resultaron muy útiles y se espera que los resultados obtenidos en este Trabajo de Fin de Grado sean útiles para el desarrollo de sistemas similares.

Abstract

This Final Degree Project approaches the implementation of a reactive tracking system on a differential drive robot. In order to do this, a differential drive based robot equipped with a Jetson Nano CPU is used. This robot, named Jetbot, is assembled and prepared with a specialized Ubuntu operative system which includes necessary libraries for the control of the robot. When the robot is working, an evaluation of its capabilities is carried on, followed by a motors calibration and the developing of a teleoperation program that is used to perform tests. New libraries are developed based on the gathered experimental results of these tests. The robot also goes through various iterations on its components, which get changed to fit the needs of the project.

Once the platform is ready, an application is developed to put to test the capabilities of the robot. This application, programmed on Python, is a real time reactive tracking system based entirely on image and developed with the intent to use the Robot Operating System (ROS) software. The application is first tested on a simulated environment, using CoppeliaSim to carry out the simulation. In this environment, a series of successful experiments confirm that the developed algorithms work correctly. After the simulation experiment's success, the application is implemented on the Jetbot and an assistant program is developed in order to help with the computer vision algorithms'calibration.

Finally, more experiments of the application are carried out in the robot, resulting on non-satisfactory results. Nevertheless, the attained experience and the design of the system itself have been useful and it is expected that the results exposed on this Final Degree Project will be useful for the development of future similar systems.

Índice

Agradecimientos III
Resumen V
Abstract VII

1 Introducción y objetivos **3**
1.1 Motivación 3
1.2 Objetivos 5
1.3 Procedimientos 6

2 Estado del arte **7**
2.1 Robótica móvil. 7
2.2 Visión Artificial en robótica móvil. 8

3 Metodología **11**
3.1 Hardware 11
3.1.1 Jetbot AI KIT 12
Montaje del robot 12
Cambio de motores y ruedas 15
3.1.2 NVIDIA Jetson Nano Developer Kit 18
3.1.3 Cámara 18
Raspberry Pi Camera Module V2 19
Cámara USB: papalook PA452 19
Cámara USB: ELP-USBFHD01M-L36 20
3.2 Herramientas de Software 21
3.2.1 Ubuntu 21
3.2.2 Python 22
3.2.3 Jupyter Notebook 22
3.2.4 OpenCV 22
3.2.5 NumPy 25
3.2.6 Gamepad de Piborg 25
3.2.7 GStreamer 27
3.2.8 ROS 28

3.2.9 CoppeliaSim 29
3.3 Aplicaciones de Software desarrolladas por el alumno. 30
3.3.1 Aplicaciones para calibración y pruebas. 31
Calibración de motores. 31
Librerías de control. 33
Librerías para manejo de cámara. 34
Teleoperación. 35
3.3.2 Aplicaciones para el sistema final. 35
Nodo de Imagen. 36
Nodo de Control. 37
Nodo Central. 39
Sistema completo. 41
Calibración de sistema de visión. 42

4 Experimentos y resultados 43
4.1 Pruebas de hardware. 43
4.1.1 Calibración de motores. 43
Descripción del experimento 43
Resultados 44
Conclusiones 45
4.1.2 Calibración de umbrales para detección de color 45
Descripción del experimento 45
Resultados 46
Conclusiones 47
4.2 Aplicación en el entorno de simulación. 47
4.2.1 Prueba de nodo de imagen en simulación. 47
Descripción del experimento 47
Resultados 48
Conclusiones 50
4.2.2 Prueba de nodo de control en simulación. 50
Descripción del experimento 50
Resultados 51
Conclusiones 51
4.2.3 Prueba de sistema completo en simulación. 52
Descripción del experimento 52
Resultados 53
Conclusiones 54
4.3 Aplicación en el robot real. 54
4.3.1 Prueba de nodo de imagen en robot real. 54
Descripción del experimento 54
Resultados 55
Conclusiones 56
4.3.2 Prueba de nodo de control en robot real. 57
Descripción del experimento 57
Resultados 58

Conclusiones 59
4.3.3 Prueba de teleoperación. 59
Descripción del experimento 59
Resultados 59
Conclusiones 60
4.3.4 Prueba de sistema completo en robot real. 60
Descripción del experimento 60
Resultados 61
Conclusiones 62

5 Conclusiones y futuras líneas 65
5.1 Conclusiones generales 65
5.2 Limitaciones encontradas durante el proyecto 65
5.2.1 Hardware 66
5.2.2 Software 67
5.3 Futuras líneas de trabajo 67

Índice de Figuras 69
Índice de Tablas 71
Bibliografía 73

Capítulo 1: Introducción y objetivos

Uno de los campos de la ingeniería que ha crecido con más importancia durante los últimos años es el de la robótica, fuertemente motivada por la necesidad y deseo de automatizar toda tarea demasiado peligrosa, inconveniente o compleja para que un humano la realice, a fin de mejorar nuestra calidad de vida en su conjunto. Y es que la robótica tiene un gran abanico de posibilidades: automatizar todo tipo de procesos, facilitar tareas tediosas, alejar a operarios humanos de entornos peligrosos o incluso entretener al público general.

En este documento se busca explorar las posibilidades de la robótica móvil, comprendiendo su estado actual y su potencial, además de implementar una aplicación sencilla que muestre las capacidades de esta tecnología de forma superficial.

1.1 Motivación

El presente *Trabajo Fin de Grado (TFG)* nace para apoyar a otro proyecto que esta siendo llevado a cabo por el Departamento de Electrónica de la Escuela Técnica Superior de Ingeniería, que tiene como objetivo el despliegue de una flota de vehículos acuáticos autónomos (Autonomous Surfaces Vehicles o ASV) en el lago Ypacarai (Asunción, Paraguay)

Figura 1.1 *Vehículo autónomo* usado para la supervisión del lago Ypacarai.

para la monitorización de contaminantes por medio de algoritmos basados en Algoritmos Genéricos y Aprendizaje por Refuerzo que planeará la ruta que seguirá la flota a fin de analizar todo el área del lago de la forma más eficiente posible [10].

El lago Ypacarai es un importante foco de turismo en la zona así como la base de un enorme ecosistema que se extiende por toda la región. Sin embargo, durante los últimos 40 años el crecimiento de la actividad agrícola en la zona ha provocado una eutrofización artificial (exceso de nutrientes en el agua causado por la actividad humana) en el lago, que da lugar al crecimiento de cyanobacterias en forma de grandes parches de algas peligrosas tanto para la fauna y flora locales como para el ser humano. Este crecimiento resulta caótico y difícil de predecir y controlar, por lo que es necesario un continuo seguimiento del crecimiento de los parches de algas, requiriendo operarios especializados que se desplacen a las zonas contaminadas en lanchas y tomen muestras de agua.

Es aquí donde entra el interés del uso de ASVs, los cuales han sido propuestos en el pasado con buenas expectativas[7]. Estos ASVs estarían equipados para navegar el lago y tomar muestras de su agua, a fin de monitorizar de forma remota la evolución del estado del lago. Los ASVs permiten ser operados directamente por un operario o por medio de una serie de puntos predefinidos o waypoints que el vehículo sigue de forma autónoma, siendo esta última la solución más interesante desde el punto de vista de automatizar el proceso completo. Por tanto, el siguiente paso consiste en tener un sistema de planificación o *path planning* que, utilizando la información del mapa del lago, pueda generar waypoints según un criterio concreto. En la publicación de Yanes et al. este problema se divide en dos, según el criterio con el que se necesite recorrer el lago: si se quiere cubrir el lago de forma *homogénea*, visitando todas las zonas con la misma frecuencia sin priorizar ningún punto en concreto; o si queremos priorizar aquellas zonas que se conoce son más propensas a la contaminación, utilizando información recogida con anterioridad, para recorrer el lago de forma *no homogénea*.

Estos dos problemas, el homogéneo y el no homogéneo, se pretenden resolver con algoritmos de Deep Reinforcement Learning y soluciones de Inteligencia Artificial para generar los caminos a seguir por los ASVs. Sin embargo, este tipo de aplicación resulta difícil de probar en el laboratorio, dada la necesidad de un medio acuático de gran tamaño para replicar adecuadamente el problema. Por tanto, resulta muy útil el disponer de un *testbed* que permita realizar experimentos en el laboratorio, utilizando robots móviles sencillos capaces de seguir los waypoints generados por el algoritmo de path planning en una menor escala.

Un testbed se define como una plataforma o entorno destinado a la experimentación y desarrollo de proyectos de gran amplitud. Estas plataformas permiten realizar experimentos consistentes, repetibles y en condiciones controladas. En este caso, se tratará de un entorno de pruebas de pequeño tamaño que simulará el lago contaminado y permitirá desplegar una pequeña flota de robots diferenciales pequeños que conocerán en todo momento su posición absoluta, tendrán conexión entre ellos y con un ordenador central que ejecutará los algoritmos y enviará las instrucciones. Cada robot emulará el funcionamiento de un ASV, tomando datos y maniobrando entre waypoints.

Es en este punto donde entra este proyecto, buscando documentar el montaje y despliegue

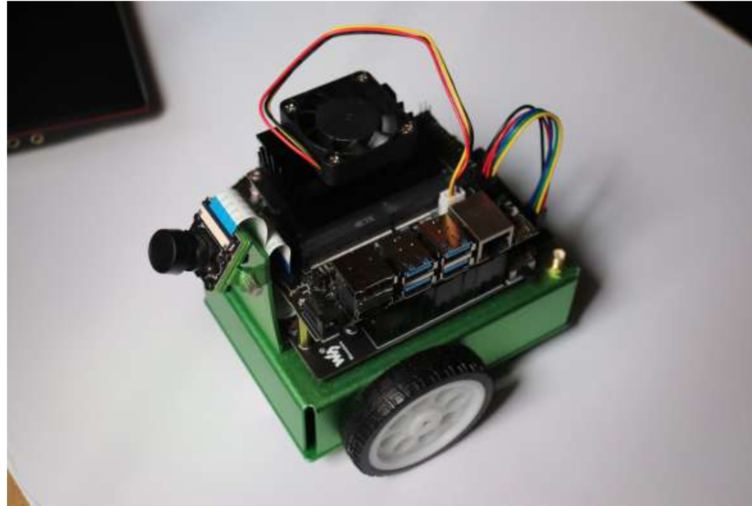


Figura 1.2 *Jetbot AI Kit de Waveshare*.

de **una unidad de *Jetbot AI Kit de Waveshare*** y probar sus capacidades tanto a nivel de hardware como de software, a fin de confirmar la elección de este robot para el testbed y asegurar que cumpla con los requisitos de esta aplicación, facilitando el despliegue del resto de unidades de la flota.

Además, a fin de extender el proyecto y aprovechar el equipo, se propone diseñar una aplicación usando ROS para explorar las capacidades del hardware una vez puesto en acción, mostrando el proceso desde el desarrollo del código en simulación hasta la implementación en el sistema real.

1.2 Objetivos

Se ha elegido marcar como objetivo final el desarrollo de una aplicación basada en **ROS y en técnicas de visión artificial**, a fin de crear un sistema completo de recolección y tratamiento de datos a partir de un sensor externo (en este caso una cámara), toma de decisiones basada en esos datos y actuación sobre el sistema. Resulta de especial interés lograr que la aplicación funcione utilizando únicamente los datos de la cámara sin ningún sensor auxiliar, para utilizar en la medida de lo posible sólo las herramientas del *Jetbot AI Kit*.

Los objetivos de este trabajo serán, por tanto, los siguientes:

- **Montar y calibrar el Jetbot AI Kit.**
- **Probar el software pre-instalado y evaluar las capacidades del sistema.** Preparar las librerías necesarias para la actuación del sistema.
- **Desarrollar una aplicación simple que permita la teleoperación**, testear la actuación sobre el sistema de forma no autónoma.

- **Desarrollar una aplicación de seguimiento reactivo en un entorno de simulación**, basándose en una cámara, técnicas de visión artificial y ROS.
- **Implementar y calibrar dicha aplicación en el Jetbot.**
- **Realizar experimentos** tanto en el entorno de simulación como en el sistema real.

Todos estos puntos se tratarán a continuación de forma breve y se profundizarán a lo largo del documento.

1.3 Procedimientos

El objetivo final de este y cualquier TFG es el de aprender, investigar temas de intereses de forma autónoma y llegar a conclusiones propias. Por tanto, se sigue esa misma estructura en la realización de este proyecto, comenzando por las tareas de aprendizaje e investigación para terminar usando esos conocimientos en los resultados finales.

Se comienza, por tanto, explorando brevemente el estado del arte de los temas de interés de este proyecto, robótica móvil y visión artificial. Esto se desarrolla en el capítulo 2.

Una vez ha sido cubierta la investigación general, en el capítulo 3 se describen las herramientas concretas empleadas y desarrolladas en el proyecto, tanto a nivel de hardware como a nivel de software. Cada herramienta se describe de forma general, explicando que es, para que sirve y sus características; además de dar una explicación simple de su utilidad y su interés para el resto del proyecto.

Los siguientes puntos se desarrollan en el capítulo 4, donde se muestran las pruebas y experimentos llevados a cabo. Los resultados se exponen de forma ordenada y fácil de leer, de modo que pueden ser referenciados en el capítulo 5 para justificar las conclusiones obtenidas en este proyecto.

Capítulo 2: Estado del arte

2.1 Robótica móvil.

Los robots móviles se definen ampliamente como máquinas autónomas capaces de desplazarse por su entorno. Actualmente se pueden encontrar aplicaciones para robots móviles en prácticamente cualquier industria, desde flotas de dispositivos destinados al almacenaje utilizados en logística hasta robots domésticos que se encuentran en todos los hogares, ayudando en diversas tareas de forma independiente.

Esta rápida extensión se debe, en parte, al desarrollo de sistemas embebidos cada vez más variados, económicos y potentes, muchos de ellos destinados a un público casual que busca experimentar con lo que la robótica puede ofrecer. Por esto resulta especialmente interesante probar como algunas de estas herramientas pueden ser utilizadas en el ámbito de la investigación y desarrollo de nuevas tecnologías, como es el caso del *Jetbot AI Kit* usado en este proyecto. Este tipo de kits, promocionados tanto para desarrolladores como para aficionados, han ganado gran popularidad durante los últimos años y se han convertido en una de las mejores opciones para el desarrollo de aplicaciones de inteligencia artificial y robótica.

Figura 2.1 Ejemplo de robot móvil..

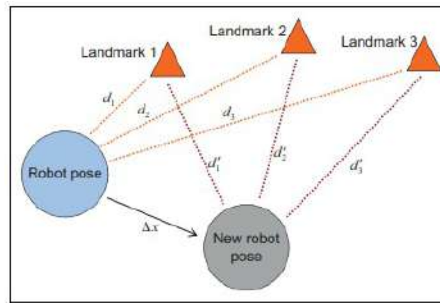


Figura 2.2 Representación gráfica del objeto del SLAM..

Y es que *Jetbot AI Kit* es un sistema destinado a la aplicación de Inteligencia Artificial (IA) en un robot móvil. Esto no es casual, pues la IA ha tenido un impacto significativo en la mejora de la autonomía y la eficiencia de los robots móviles en diversos entornos: la aplicación de la IA en la robótica móvil ha permitido la navegación autónoma, la percepción del entorno y la toma de decisiones en tiempo real. En el caso de la inspección del lago Ycaparai [10], por ejemplo, se hace uso de un sistema inteligente para planear el camino que ha de seguir el robot.

Tal y como se describe en el articulo "A survey of image semantics-based visual simultaneous localization and mapping: Application-oriented solutions to autonomous navigation of mobile robots"[14], la pieza clave para la navegación autónoma de robots móviles es la "localización y mapeo simultaneos" o "SLAM", que permite que un robot en una localización desconocida en un entorno desconocido utilice datos de su entorno para localizarse y crear de forma incremental un mapa. Esto se logra identificando puntos de interés en el entorno del robot haciendo uso de diversos sensores de diversa naturaleza. Sin embargo, este tipo de sistemas sufre mucho en entornos dinámicos, complejos y a gran escala.

Esta limitación hace destacar las variantes de SLAM que hacen uso de cámaras como sus únicos sensores. Estos sistemas, denominados visual SLAM, permiten una mayor adaptabilidad y simplicidad en los sistemas. Y es gracias a la visión artificial y a la inteligencia artificial que en la actualidad este tipo de SLAMs nos han permitido llegar a una era, definida por Cadena et al.[5], de "percepción robusta" en la que tenemos una gran cantidad de recursos y comprensión sobre la navegación autónoma y robusta para los robots móviles.

2.2 Visión Artificial en robótica móvil.

La visión artificial ha desempeñado un papel fundamental en el desarrollo de robots móviles, proporcionando capacidades perceptivas similares a las de los seres humanos. A través de la implementación de algoritmos y técnicas de procesamiento de imágenes, los robots móviles pueden adquirir información visual del entorno que les rodea y utilizarla para la navegación, la detección de objetos y la toma de decisiones.

Uno de los aspectos clave de la visión artificial en robots móviles es la capacidad de reconocer y localizar objetos. Mediante el análisis de características visuales, como formas, colores y texturas, los robots pueden identificar y distinguir objetos en su entorno. Esto, tal

Figura 2.3 Experimentos en un entorno real del seguimiento de personas..

y como se expone en el apartado anterior, puede ser utilizado para algoritmos de navegación autónoma para identificar puntos de interés en el entorno.

Además de la detección de objetos y la navegación, la visión artificial en robots móviles también se ha utilizado en aplicaciones como la monitorización y el seguimiento de personas. En el paper "A Novel Vision-Based Tracking Algorithm for a Human-Following Mobile Robot"[8] , se propone un sistema de visión artificial para el seguimiento de personas usando un robot móvil. El sistema utiliza algoritmos de visión por computadora para detectar y rastrear a las personas en tiempo real, lo que resulta especialmente útil en escenarios de seguridad o asistencia a personas mayores.

Capítulo 3: Metodología

Para la realización de este Trabajo se han utilizado diversas herramientas de terceras partes, tanto hardware como software. En este apartado, estas herramientas serán detalladas y se expondrá el uso que han tenido en el proyecto. Además, se han desarrollado múltiples aplicaciones de software. Estas aplicaciones también serán detalladas a continuación.

3.1 Hardware

Todo sistema robótico ha de disponer de un hardware capaz de llevar a cabo las tareas requeridas: proporcionando actuación, sensores y potencia para ejecutar el software. A continuación, se detalla el hardware usado en el proyecto:

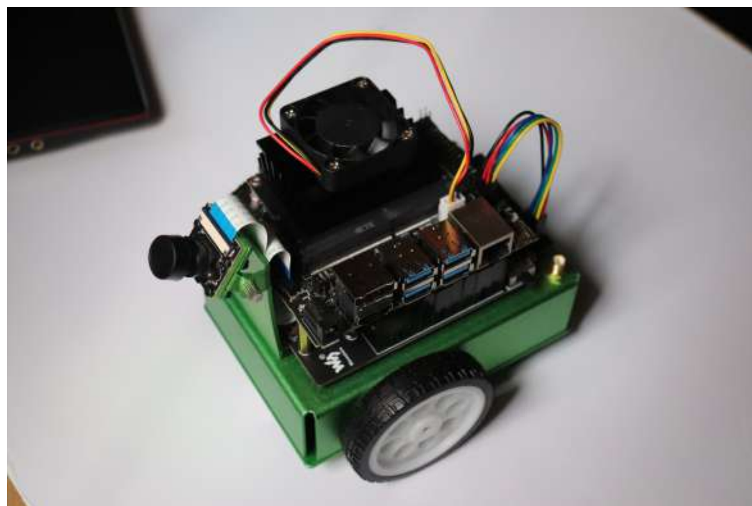


Figura 3.1 *Jetbot AI Kit de Waveshare..*

3.1.1 Jetbot AI KIT

La base de este TFG es el *Jetbot AI KIT de Waveshare*, un pequeño robot de configuración diferencial promocionado como una herramienta para el desarrollo de aplicaciones de Inteligencia Artificial, basado en el *NVIDIA Jetson Nano* que posteriormente se explicará con detalle. Su principal objetivo es el uso de Redes Neuronales y otras tecnologías similares basadas en IA para la obtención de datos a partir de las imágenes obtenidas por la cámara que trae equipada el vehículo, por lo que este dispone de una pequeña pero potente GPU *Jetson Nano* capaz de procesar imágenes y ejecutar aplicaciones de Inteligencia Artificial de forma nativa.

La información obtenida por la cámara o como resultado de los algoritmos y programas ejecutados en la *Jetson Nano* pueden usarse para la propia operación del robot o enviarse por comunicaciones a otros dispositivos, por medio de comunicación inalámbrica como WiFi o Bluetooth o por medio de los pines de comunicación del propio kit de desarrollo.

Mecánicamente, el robot consiste en un pequeño vehículo de configuración diferencial consistente en dos ruedas locas y dos ruedas conectadas a motores de corriente continua encargados de la actuación. Esto permite movilidad libre en superficies planas, dándole la capacidad al robot de girar sobre su propio eje y moverse hacia delante o hacia atrás de forma sencilla. Lógicamente, al tratarse de un robot móvil, la alimentación es por medio de baterías recargables, eliminando la necesidad de alimentar por medio de cable.

A continuación se listan las especificaciones de serie del robot:

- CPU: NVIDIA Jetson Nano
- Cámara: Sensor IMX219 (8MP 160º FOV wide angle)
- Display: Pantalla OLED de 0.91 pulgadas, 128x32 píxeles
- Comunicaciones: NIC AC8265 (2.4 GHz/5 GHz WIFI dual + Bluetooth 4.2)
- Batería: 3 unidades de baterias 18650 en serie, con circuito de protección contra sobrecarga/sobre-descarga de baterías, contra cortocircuitos y contra sobreintensidades.
- Motores: motor TT con ratio de reducción de 1:48 y velocidad nominal de 240 RPM
- Chasis: Aleación de aluminio

Montaje del robot

El *Jetbot AI Kit* proporciona todos los componentes del robot listos para ser ensamblados. Este proceso se detalla a continuación, paso por paso:

1. Se comienza por el inferior del robot, el chasis de aluminio (Figura 3.2). Al chasis

primero se le atornillan los dos motores, uno a cada lado, asegurándolos a los laterales de la estructura tal y como se muestra en la figura 3.3.

Figura 3.2 Chasis desmontado.

Figura 3.3 Montaje de los motores en el chasis.

2. A continuación se asegura al chasis los puertos para conectar las antenas al módulo wi-fi, que más adelante se conectará en la Jetson Nano; y se colocan cuatro tornillos para asegurar sobre ellos la electrónica del robot. En la figura 3.4 se aprecian los conectores de las antenas y los tornillos.

Figura 3.4 Parte superior del chasis.

3. Se monta el soporte para la cámara. Se atornilla la cámara al soporte y el soporte al chasis, tal y como se ve en la figura 3.5.

4. Finalmente, antes de cerrar la parte inferior del chasis, se coloca la primera pieza

Figura 3.5 Soporte para la cámara sujeto al chasis.

de electrónica asegurándola con tornillos al chasis (figura 3.6. Aprovechando que el chasis sigue abierto por debajo, se conectan los motores a la placa por medio de conectores situados justo debajo.

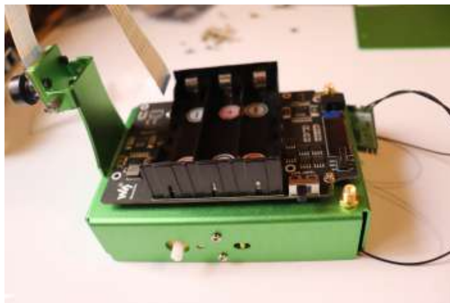


Figura 3.6 Placa sujeta al chasis.

5. Finalmente se cierra el chasis, sujetando a la chapa inferior dos ruedas locas en las partes posterior y anterior (figura 3.7). A su vez, se colocan las dos ruedas grandes en los ejes de los motores, asegurándolas con un tornillo tal y como se muestra en la figura 3.7. Se colocan las baterías en la orientación indicada en el porta-baterías.

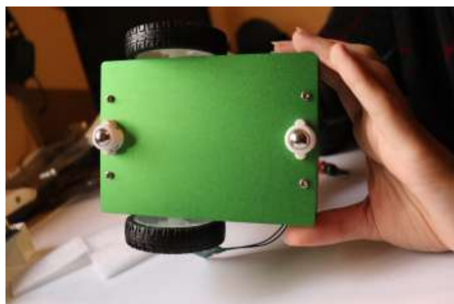


Figura 3.7 Cerrado del chasis y posicionamiento de las ruedas en el eje.

6. Para terminar, se asegura la Jetson Nano encima de la otra placa usando tornillos y se conecta el módulo wi-fi. Se atornilla encima del dispensador de calor un ventilador y se conecta a la Jetson. A su vez, se conecta la cámara al bus serie (puerto CSI) y se

conectan la Jetson y la placa con baterías (figura 3.8).

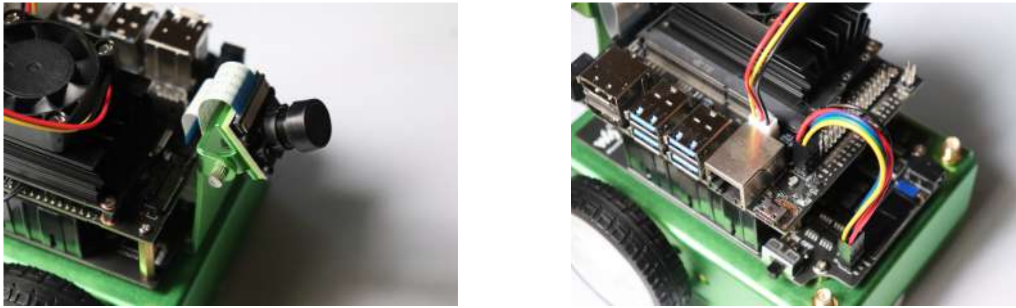


Figura 3.8 Conexionados de la cámara, el ventilador y entre placas.

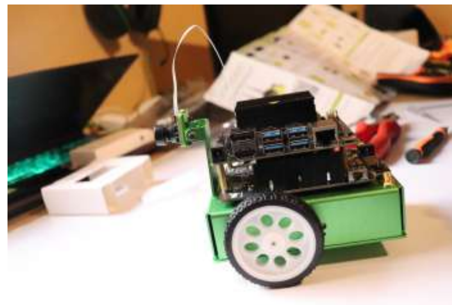


Figura 3.9 Jetson asegurada en la parte superior del robot.

Siguiendo estos pasos, se monta finalmente el Jetbot AI Kit dejándolo listo para su funcionamiento (figura 3.9). Sin embargo, antes de poder usar el robot, hay que asegurarse de cargar bien las baterías y preparar el sistema operativo que se cargará en la Jetson Nano.

Para cargar el sistema operativo en la Jetson Nano, hay de cargar una imagen de Ubuntu proporcionada por NVIDIA configurada especificamente para la Jetson en una tarjeta SD utilizando software especializado para ello. En este Trabajo se utilizó *Balena Etcher* para cargar la imagen en la tarjeta SD que más tarde se introduciría a la Jetson Nano.

Cambio de motores y ruedas

Debido a la mala calidad de las ruedas originales y al desgaste de los motores a lo largo de los experimentos, se decidió cambiar ambos elementos antes de realizar los experimentos finales y aprovechar para realizar un mantenimiento del dispositivo. A continuación se muestra el proceso:

- El primer paso es soltar los tornillos que sujetan la Jetson Nano al resto de la electrónica, con cuidado de soltar todos los cables y desacoplar el ventilador. Se muestra en la figura 3.10.

- A continuación se aprovecha para inspeccionar la salud de las baterías. Para ello, se utiliza un multímetro para tomar medidas del voltaje de cada pila. Si este voltaje esta

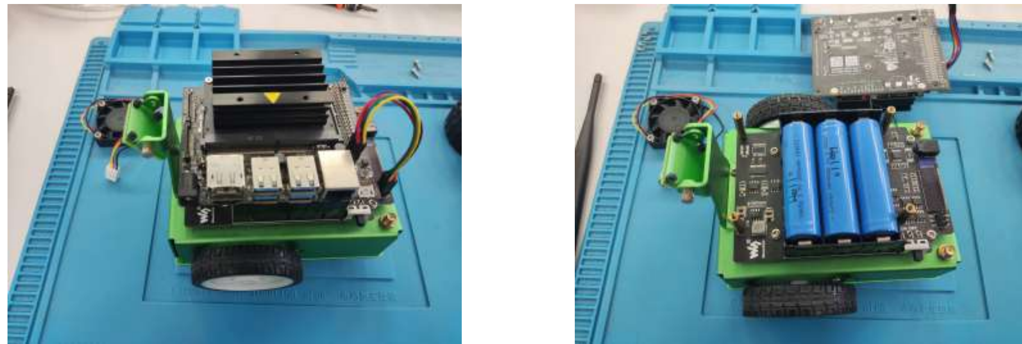


Figura 3.10 Desmontaje de la Jetson para acceder a la batería.

demasiado bajo, la batería esta dañada y debería sustituirse. Tal y como se muestra en la figura 3.11, las baterías estaban sanas.

Figura 3.11 Medida de tensión de baterías.

- Se procede a separar la placa con las baterías del chasis de aluminio del robot, con cuidado ya que quedan dos cables conectando el chasis a la placa Jetson Nano. Con el chasis ya separado, se quita también el soporte para la cámara. Por último, se abre el chasis por la parte inferior, desvelando los motores (figura 3.12).

- Se cambian los motores por otros nuevos e idénticos a los anteriores. Las ruedas también son cambiadas por otras del mismo diámetro, aunque más gruesas a fin de aumentar la fricción con el suelo. Los nuevos componentes se muestran ya instalados en la figura 3.13. Los nuevos motores no disponían de los mismos conectores, por lo que se realiza una soldadura de estaño para añadir el conector adecuado a los motores. Las soldaduras son debidamente aisladas con tubos termo-retráctiles para evitar cortocircuitos.

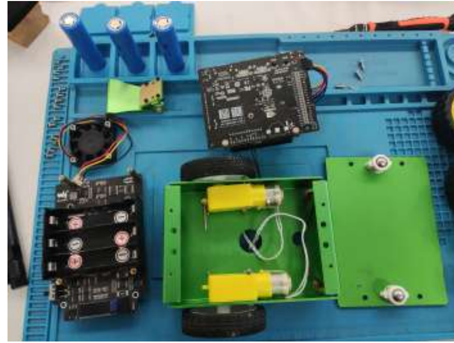
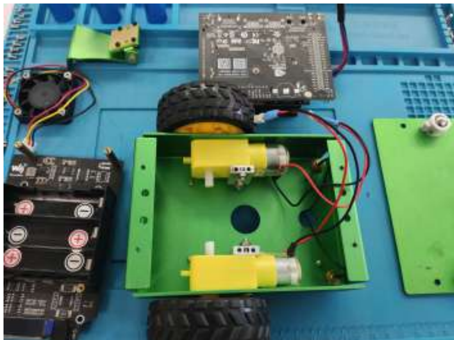


Figura 3.12 Vista inferior del chasis abierto.



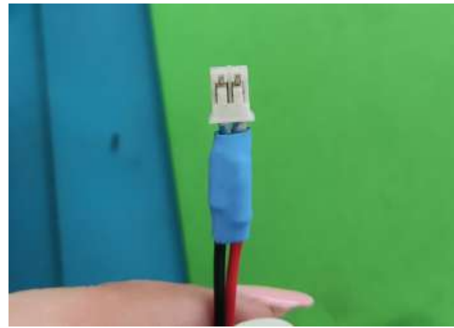


Figura 3.13 Nuevos motores y ruedas montados.

- Finalmente, se vuelve a montar todo el dispositivo, aprovechando para limpiar cada componente. El resultado final se muestra en la figura 3.14.

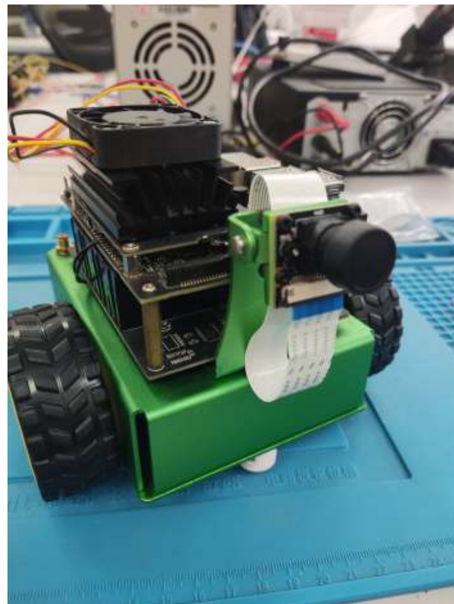


Figura 3.14 Jetbot con nuevas piezas instaladas.

Como nota final, las ruedas nuevas resultaron ser demasiado anchas en la parte en la

que se unen a los ejes, provocando fricciones indeseadas entre la rueda y el chasis. Para solucionar eso, se diseñan e imprimen en 3D pequeñas piezas cilíndricas que van en el eje entre la rueda y el motor y que alejan la rueda del chasis para evitar estas fricciones.

3.1.2 NVIDIA Jetson Nano Developer Kit

Figura 3.15 NVIDIA Jetson Nano Developer Kit.

El Jetbot AI KIT viene equipado con el *NVIDIA Jetson Nano Developer Kit* o *Kit de desarrollo*. En realidad, la Jetson Nano es una GPU desarrollada por NVIDIA para pequeños proyectos de robótica, IoT y desarrollo del producto, siendo una solución pequeña, potente y muy eficiente desde el punto de vista energético, pues solo consume 5W en su configuración de mínimo consumo.

Además de la GPU (la Jetson Nano propiamente dicha), el Kit de desarrollo añade un procesador, memoria RAM, múltiples puertos I/O y varios accesorios para que el dispositivo sea capaz de mantener el sistema operativo de forma independiente. Concretamente, el kit cuenta con una CPU cuadcore Cortex-A57, una GPU 128-core con arquitectura Maxwell y 4GB de memoria LPDDR4 como sus especificaciones más importantes.

Con estas especificaciones, el ordenador soporta sobradamente Ubuntu, lo que permite usar herramientas de este sistema operativo como son ROS (Robotic Operative System) o la programación en Python, pilares de este trabajo. Es por esto que se le instala una versión de Ubuntu 18.04 que puede obtenerse con el Jetbot AI Kit y que incluye todo lo necesario para que la Jetson Nano pueda manejar el robot.

3.1.3 Cámara

Tras realizar pruebas con el software, se ha demostrado que la cámara de serie del Jetbot presenta una importante distorsión radial en como representa los colores (figura 3.16. Esto no sería necesariamente un problema en aplicaciones que no dependan fuertemente en la detección de colores, no obstante, en el caso de este proyecto, la consistencia en el color requerimiento fundamental. Por ende, se toma la decisión de cambiar la cámara de serie por otra sin esta distorsión.

Figura 3.16 Ejemplo de la distorsión radial en la cámara original.

Raspberry Pi Camera Module V2

A fin de reemplazar la cámara de serie, se trata de implementar la *Raspberry Pi Camera Module V2* o *PiCam V2*, ya que se trata de una cámara similar y que usa el mismo puerto CSI.

Esta cámara no tiene los problemas de color que presenta la cámara de serie y presenta una buena calidad. Sin embargo tiene un gran defecto que afecta significativamente al funcionamiento del sistema. Con un campo de visión o *Field of View (FOV)* de apenas 62.2 grados, se pierde mucha información vital para guiar al robot. Por esto, se opta por probar otras opciones.

Cámara USB: papalook PA452

La primera prueba de una cámara con interfaz USB se realiza con una *webcam* comercial típica con objetivo de probar el rendimiento y comprobar si es eficiente evitar usar el bus serie (interfaz CSI), ya que las cámaras USB son mucho más simples de usar.

Figura 3.17 *Cámara USB* papalook.

Las pruebas con la cámara USB *papalook PA452* (figura 3.17 resultan en un gran éxito, además de proporcionar una mayor calidad de imagen y color, es capaz de proveer una feed de vídeo fluido sin problemas a bajas resoluciones. Sin embargo, tiene serios problemas:

- El tamaño de la cámara hace que su uso resulte poco conveniente, siendo imposible sujetarla al cuerpo del robot de forma estable.
- El consumo de la cámara, tanto en recursos como en potencia, perjudican al sistema completo.

Cámara USB: ELP-USBFHD01M-L36

Finalmente se opta por comprar una cámara más adecuada al problema abordado en este proyecto, la *ELP-USBFHD01M-L36* (figura 3.18. Esta cámara, diseñada para aplicaciones de alta velocidad, dispone de las siguientes especificaciones que la hacen perfecta para esta aplicación:

- Ligera y compacta, tan solo un poco más grande que la PiCam V2 o la de serie. Con un simple adaptador impreso en 3D, la instalación en el robot es sencilla.
- Poco consumo y altas velocidades.
- FOV amplio, proporcionando mucha más información que el resto de cámaras.

Por estos motivos, se elige esta cámara para la versión final del robot, instalándose junto a un adaptador diseñado en CATIA por el alumno e impreso en 3D. En la figura 3.19 se puede ver la cámara ya montada en el robot.

Figura 3.18 *Cámara USB* ELP-USBFHD01M-L36.

Figura 3.19 Adaptador para sujetar la cámara al Jetbot.

3.2 Herramientas de Software

En este apartado se exponen las bases, el sistema operativo y lenguaje de programación usados, y las librerías específicas; detallando como funcionan, el rol de cada herramienta en el proyecto, su utilidad y por qué se han elegido.

3.2.1 Ubuntu

Ubuntu[1] es un sistema operativo de software libre y código abierto orientado al público general. Se trata de una distribución de Linux basada en Debian, patrocinada por la compañía británica *Canonical*.

Ubuntu se centra en ofrecer facilidad de uso, accesibilidad y simplicidad, como se puede ver, por ejemplo, en el diseño de su escritorio similar al que se puede encontrar en un sistema Windows, al contrario que los entornos de escritorio que pueden eencontrarse en otras distribuciones de Linux. Otro aspecto que diferencia al sistema de otras distribuciones de Linux, es su amplia comunidad de usuarios, que pueden participar de forma libre en el desarrollo del sistema operativo ayudando a dar soporte y publicando una gran cantidad de información en internet.

Existen múltiples versiones de Ubuntu, que son lanzadas cada aproximadamente seis meses. En este trabajo, se utiliza la versión de ubuntu 18.04 LTS ("Long Time Support", que significa que esta versión recibirá soporte durante muchos más años que una versión no LTS).

Se utiliza este sistema operativo para la Jetson Nano por dos motivos. El primero es su simplicidad y poco gasto de recursos, que lo hacen una opción perfecta para un sistema pequeño. El segundo es que es compatible de forma sencilla y natural con ROS y Python, herramientas fundamentales para el proyecto.

3.2.2 Python

Python[13] es un lenguaje de programación de alto nivel, interpretado, dinámico, multiplataforma y multiparadigma. Este lenguaje nace buscando la simplicidad, centrando su sintaxis en la legibilidad y haciendo la lectura y aprendizaje general del lenguaje considerablemente más sencilla que en otros lenguajes de programación. Es un lenguaje flexible con una gran comunidad de usuarios que contribuyen constantemente con nuevas librerías y aplicaciones.

Como se ha explicado anteriormente, es un lenguaje interpretado y dinámico. Esto, en contraste con lenguajes compilados (como, por ejemplo, C), significa que las aplicaciones funcionan interpretando directamente el código a medida que se ejecutan a través de una aplicación interprete que ha de estar instalada en el sistema que ejecute el código. De este modo, cualquier sistema puede ejecutar el código directamente siempre y cuando tenga acceso al interprete, lo que implica que las aplicaciones se pueden portear fácilmente. El lado negativo de esto es la velocidad de ejecución, que disminuye notoriamente al tener que ir interpretando el código línea a línea a medida que se ejecuta la aplicación.

Este lenguaje se utilizará en casi todas las aplicaciones desarrolladas en el proyecto.

3.2.3 Jupyter Notebook

Jupyter Notebook[2] es una herramienta desarrollada como parte del *Jupyter Proyect*, una iniciativa de código libre sin animo de lucro con el objetivo de dar soporte a aplicaciones de ciencia de datos y computación científica para todos los lenguajes de programación.

Jupyter Notebook como tal es una interfaz web que permite trabajar en un entorno de desarrollo local o remoto y crear documentos donde se pueden, de forma simultanea, incluir fragmentos de código ejecutable con fragmentos con texto, imágenes, vídeos y todo lo que se pueda incluir en una aplicación web. Esta aplicación permite usar cualquier lenguaje de programación, aunque dispone de un motor para la ejecución de Python que es el usado en este proyecto.

En este proyecto este entorno se utiliza a modo de servidor, para permitir que ordenadores conectados a la misma red local que el Jetbot puedan conectarse y utilizar esta interfaz para desarrollar código o manejar los archivos remotamente, de forma cómoda, sencilla y sin tener que conectar una pantalla a la Jetson Nano del robot.

3.2.4 OpenCV

OpenCV (Open Source Computer Vision Library[4] es una librería de software libre dedicada a la visión artificial y el "machine learning". Es multiplataforma, soporta todo tipo de entornos de programación (C++, Python, Java y MATLAB) y además esta ampliamente documentada y explicada, por lo que es la opción más popular para el desarrollo de soluciones de visión artificial. En este proyecto se utiliza la versión de Python.

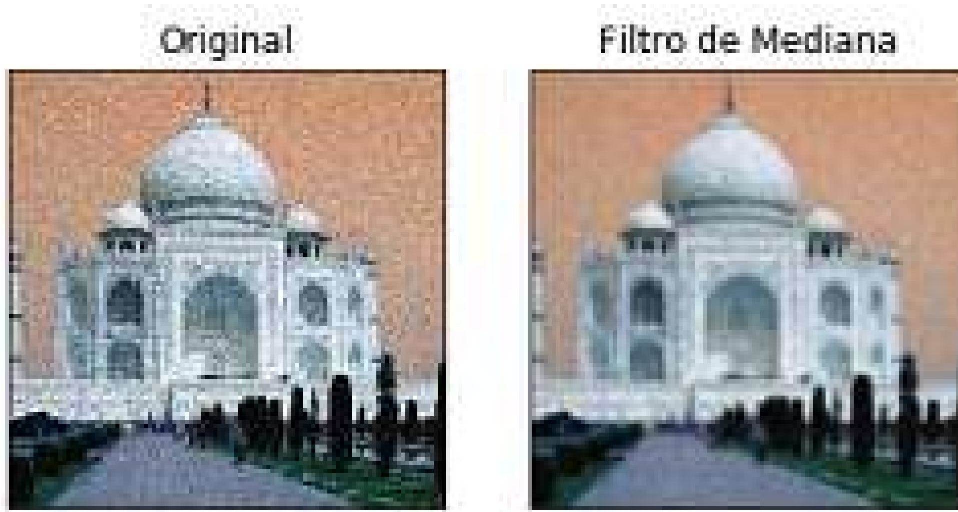


Figura 3.20 Ejemplo de filtro de medianas.

Hay más de 2500 algoritmos, aunque en este proyecto solo se usan unos pocos, que se explican brevemente a continuación:

- *cv2.cvtColor*: Se utiliza para cambiar los espacios de color, por ejemplo, para cambiar de RGB (Red Green Blue) a escala de grises.

  El espacio de color es la forma en se representan los colores en una imagen o vídeo digitales.

- *cv2.medianBlur*: Se utiliza para suavizar las imágenes y aliviar el ruido que puedan contener aplicando un filtro de medianas.

  El filtro de medianas es un tipo de filtro no lineal que recorre la imagen tomando para cada píxel una pequeña cuadricula al rededor del mismo (llamada vecindario) y reemplaza el valor de ese píxel por el valor de la mediana de los valores de todos los píxeles dentro de dicha cuadricula. Se usa mucho para aliviar los efectos del ruido de sal y pimienta. Un ejemplo se puede ver en la figura 3.20.

- *cv2.morphologyEx y cv2.getStructuringElement*: Estas funciones se utilizan para aplicar operaciones morfológicas, una serie de operaciones no lineales que se aplican a partir de un "elemento estructural". Funcionan de forma similar al filtro de medianas, utilizando un vecindario que creado con *"cv2.getStructuringElement"* al cual se le puede dar la forma que se desee. Este elemento estructural se aplica con *"cv2.morphologyEx"* en una imagen (que normalmente es una imagen binaria), aplicando una de las distintas operaciones disponibles.

  Las operaciones morfológicas más comunes son la *erosión* y la *dilatación*, que permiten, respectivamente, "adelgazar" o "erosionar" el área que compone la imagen y "agrandar" o "dilatar" esa área. Combinando estas dos operaciones básicas, se pueden obtener operaciones más avanzadas como: la operación gradiente, que permite obtener los bordes de una zona a partir de la diferencia entre las operaciones de

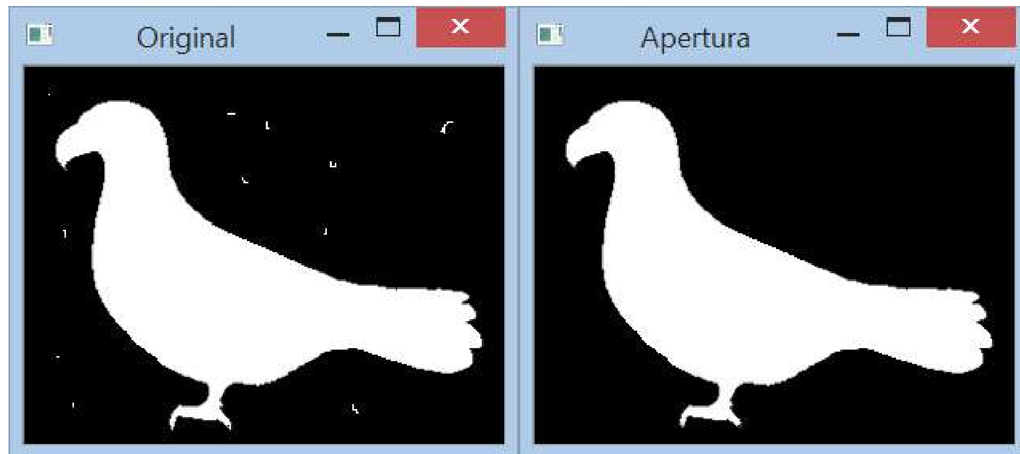


Figura 3.21 Ejemplo de operación de apertura.

dilatación y erosión; la operación apertura, que se obtiene de aplicar una erosión y después una dilatación y que se puede utilizar para eliminar ruido fuera de una máscara; o la operación de cierre, obtenida de una erosión tras una dilatación y que permite "cerrar" los huecos dentro de una figura o zona. Un ejemplo de la operación de apertura puede verse en la figura 3.21

- *cv2.inRange*: Se utiliza para segmentar imágenes, es decir, seleccionar ciertas secciones de una imagen que cumplen una condición concreta. En el caso de esta función, esta condición concreta es que los valores de color de los píxeles a segmentar estén dentro cierto rango dado. De este modo, se pueden diferenciar zonas de un cierto color del resto de la imagen. Este proceso da lugar a una *máscara*, una imagen binaria donde la zona que cumple la condición (tener cierto color) vale 1 y la que no lo cumple (el resto de la imagen) vale 0.

- *cv2.findContours*: Se utiliza para encontrar los contornos de una imagen binaria. Esto permite aplicar distintas técnicas de análisis de formas para conocer características de los objetos (por ejemplo, área u orientación) además de saber la cantidad de objetos que se han segmentado y permitir elegir aquellos que sean de interés.

- *cv2.contourArea y cv2.moments*: Se trata de funciones de análisis de formas, utilizadas para obtener propiedades de secciones segmentadas o contornos. Estas propiedades pueden ser el área (como es el caso de *"cv2.contourArea"*), la orientación o el centro geométrico de la forma.

  Estas propiedades obtienen a partir de los *momentos de la sección*, dados por la función *"cv2.moments"*. Sin embargo, no todas se pueden obtener directamente de los momentos y algunas requieren operar con los valores de los momentos. Por ejemplo, la posición del centroide en el plano X,Y se obtiene con la formula:

$$C_x = \frac{M_{10}}{M_{00}}, \quad C_y = \frac{M_{01}}{M_{00}}$$

  donde M_{ij} es el momento "ij" de la imagen.

- *cv2.circle*: Se utiliza para dibujar un circulo en una imagen. Permite especificar color, posición y tamaño.

3.2.5 NumPy

NumPy[9] es una librería de software libre creada para implementar computación numérica en Python, permitiendo realizar operaciones muy potentes con matrices y vectores de un modo similar a como se haría en MATLAB.

NumPy es muy usado en conjunto a openCV, pues este trata las imágenes como matrices o hiper-matrices con los valores de cada píxel. Así pues, las herramientas de computo numérico que pone a nuestra disposición *NumPy* son muy útiles a la hora de manejar imágenes. Ambas librerías son indispensables si se desea desarrollar aplicaciones de visión artificial.

3.2.6 Gamepad de Piborg

Gamepad[6] es una librería para Python creada con el objetivo de ofrecer una solución sencilla para el manejo de controladores de videojuegos en Python. Esta librería permite manejar controladores de tres modos:

- *Modo "Polling":* Se trata del modo más sencillo. Tal y como se ilustra en la figura 3.22, la librería interpreta los eventos del controladores uno a uno.

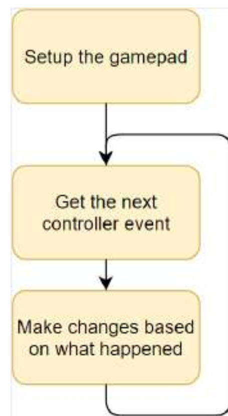


Figura 3.22 Funcionamiento del modo Polling.

Funciona llamando repetidamente a la función getNextEvent() para obtener la siguiente actualización del controlador en el orden en que ocurrieron. Cada llamada devuelve tres cosas: El tipo de evento , el botón o joystick que cambió y el nuevo valor al que dicho botón o joystick cambia. Este modo no se puede utilizar al mismo tiempo que los modos asíncrono o de eventos, ya que esos modos ya leen de forma autónoma las actualizaciones del controlador.

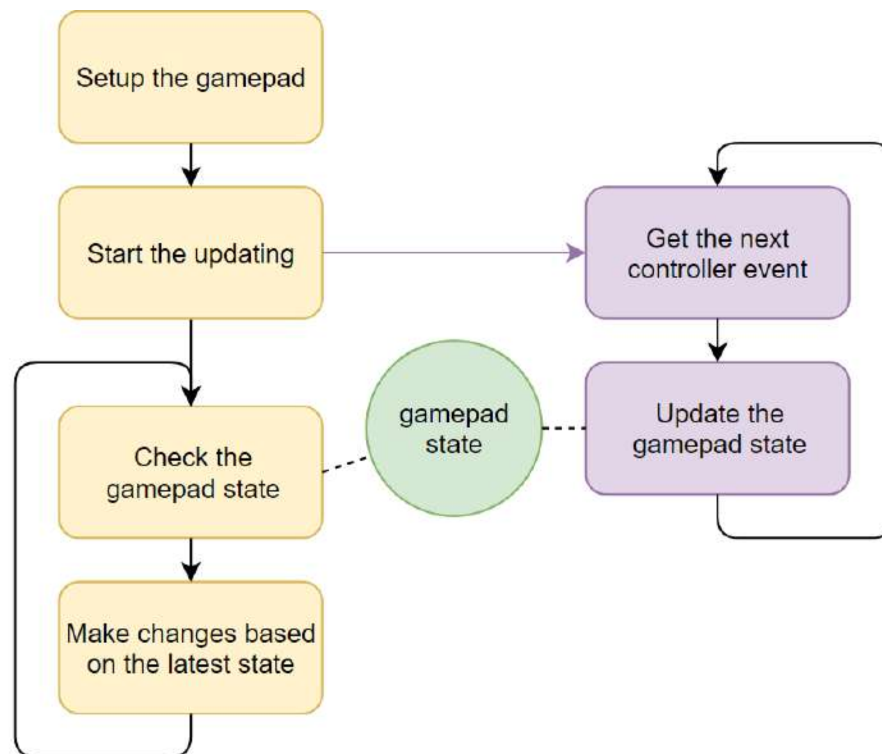


Figura 3.23 Funcionamiento del modo asíncrono.

- *Modo asíncrono:* El modo asíncrono funciona leyendo las actualizaciones del mando en un hilo en segundo plano y actualizando el estado del objeto en Python para que coincida con el estado del controlador. Su funcionamiento se ilustra en la figura 3.23

  Se inicia llamando a la función startBackgroundUpdates() y debe detenerse al final del script llamando a la función disconnect().

  El estado actual del controlador se puede consultar utilizando las siguientes funciones:

  - isConnected() - comprueba si el controlador está conectado.
  - isPressed(X) - comprueba si un botón está actualmente pulsado.
  - beenPressed(X) - verifica si un botón fue pulsado desde la última comprobación.
  - beenReleased(X) - verifica si un botón fue liberado desde la última comprobación.
  - axis(X) - lee la última posición del eje de un joystick.

  En todos los casos, X es el nombre del botón/eje a comprobar o su índice numérico.

  El modo asíncrono no se puede utilizar al mismo tiempo que el modo "Polling" pero se puede utilizar con el modo de eventos.

- *Modo de eventos:* El modo de eventos funciona leyendo las actualizaciones del

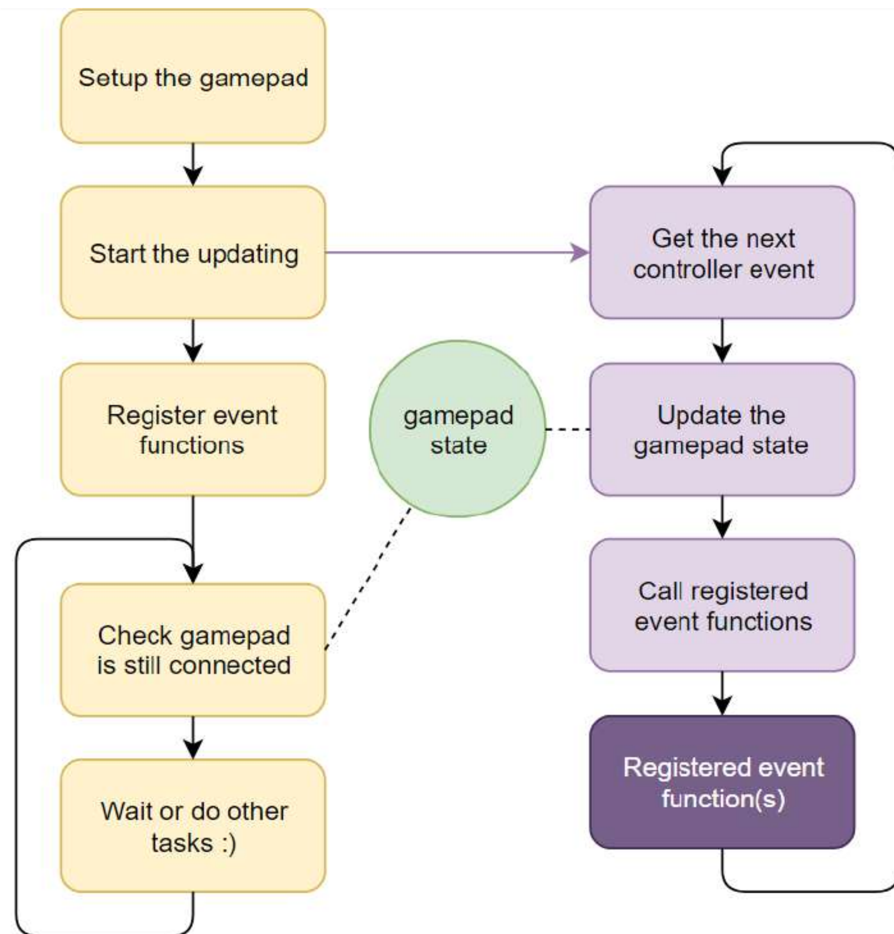


Figura 3.24 Funcionamiento del modo de eventos.

controlador en un hilo en segundo plano y llamando a funciones predefinidas cuando se producen cambios. Su funcionamiento se ilustra en la figura 3.24

Se inicia llamando a la función startBackgroundUpdates() y debe detenerse al final del script llamando a la función disconnect(). Una vez el modo ha sido iniciado, se pueden registrar funciones para distintos eventos como cuando se pulsa un botón, se suelta, cambia de estado o cuando se mueve el eje de un joystick. La misma función se puede registrar con múltiples eventos. También pueden registrar múltiples funciones con el mismo evento.

El modo de eventos no se puede utilizar al mismo tiempo que el modo "Polling" pero se puede utilizar con el modo asíncrono.

En este proyecto se utilizan en conjunto el modo asíncrono con el modo de eventos.

3.2.7 GStreamer

GStreamer es una librería multiplataforma destinada a la creación de sistemas para el manejo de multimedia, utilizando un sistema de "plugins" que pueden conectarse entre sí

para configurar aplicaciones capaces de tareas simples como la reproducción de vídeos hasta tareas complejas como la mezcla de audio o la edición de vídeo.

Estas aplicaciones se construyen enlazando distintos nodos llamados "plugins", cada uno con una funcionalidad distinta, que se van pasando la información multimedia de uno a otro de forma lineal o ramificada. Esta forma de "apilar plugins" en una aplicación se suele denominar "pipeline". Normalmente estas "pipelines" comienzan con una o varias fuentes o "sources" y terminando en uno o varios sumideros o "sinks", habiendo en medio distintos "plugin" encargados de procesar el vídeo.

A continuación, se detallan algunos de los plugin usados:

- **v4l2src:** Este plugin permite obtener vídeo de una cámara conectada por medio de la API v4l2 de Linux. Esto implica poder obtener la imagen de cualquier cámara conectada al Jetbot que sea reconocida por el sistema operativo.

- **videomedian:** Un plugin que, en tiempo real, aplica un filtro de medianas al vídeo. Hacer preprocesamiento con este plugin es mucho más rápido que hacerlo posteriormente en Python con OpenCV.

- **gaussianblur:** De forma similar al anterior, realiza un filtro gaussiano al vídeo.

- **appsink:** Este plugin, normalmente colocado al final de la pipeline de GStreamer (es un "sink"), se encarga de pasar la información de la media procesada por la pipeline a otra aplicación. Por ejemplo, permite recoger la imagen ya preprocesada directamente en Python y usarla en código.

- **xvideosink:** Otro plugin "sink", este permite visualizar en pantalla la media que procesa la pipeline. En este trabajo se utiliza puramente para realizar pruebas y comprobar que todo funciona bien.

En este proyecto, GStreamer se utiliza para obtener el vídeo de la cámara conectada al robot sin tener que depender de las librerías que vienen de serie con el Jetbot, lo que permite controlar como llega la señal, la resolución de captura o la tasa de fotogramas por segundo. Además, el uso de ciertos plugin como *"videomedian"* permite dar un preprocesado a la imagen mucho más eficiente que por medio de Python y OpenCV. Posteriormente en este documento se detalla la pipeline construida para lograr todo esto y se explica como esta se integra con el resto del código.

3.2.8 ROS

ROS (Robot Operating System)[12] es un framework diseñado para el control de robots. Es un conjunto de herramientas diseñadas para el desarrollo y programación de sistemas robóticos complejos y robustos, basado en una arquitectura de nodos que envían y reciben mensajes y señales por medio de servicios similares a los que daría un sistema operativo.

Una de las mayores atracciones de ROS es la gran variedad de paquetes que están

disponibles para sus usuarios. Estos paquetes permiten implementar de forma rápida, sencilla y modular todo tipo de funcionalidades: mapeo, navegación, localización, control, etc.

ROS funciona con un sistema de nodos, topics y servicios, formando una estructura de grafos interrelacionando los nodos entre ellos, con los sensores y actuadores y con el usuario. Los nodos son los bloques principales de construcción de un sistema en ROS, procesos programados en C++ o en Python que se encargan de llevar a cabo las distintas tareas de forma modular e independiente del resto de nodos. Estos nodos se comunican entre ellos mediante dos mecanismos: los topics, que son unas direcciones a las que uno o varios nodos pueden publicar datos de forma síncrona y uno o varios nodos pueden suscribirse para leer los datos publicados; y los servicios, un método de comunicación asíncrono que puede ser llamado por otros nodos o por el usuario y que permite enviar mensajes del mismo estilo que los publicados en los topics de forma puntual.

Las herramientas y paquetes de ROS utilizados en este proyecto son:

- *catkin:* Herramienta para compilar los distintos nodos, mensajes personalizados y demás componentes de cada paquete de ROS.

- *roslaunch:* Herramienta para lanzar sistemas de ROS a partir de las instrucciones dadas en un archivo ".launch" escrito en XML. Permite lanzar múltiples nodos de forma simultanea configurándolos usando parámetros personalizables llamados "rosparam".

- *rviz:* Un simulador 3d para robots. Es altamente compatible con ROS y permite visualizar de forma sencilla la información de la que dispone el robot leyendo los topics a los que este se suscribe y mostrándola de forma visual.

La aplicación final del proyecto es un sistema basado en ROS, donde se hace un uso profundo de todas las herramientas que puede ofrecer. Posteriormente, se detalla el sistema en profundidad.

3.2.9 CoppeliaSim

CoppeliaSim[11] es un simulador de robots muy versátil y ligero utilizado tanto en investigación como en industria y educación. Entre sus muchas funcionalidades, resulta muy interesante para este trabajo su fácil integración con ROS, que permite la comunicación con la simulación por medio de topics.

En este proyecto se utiliza como entorno de experimentación para tener una prueba de que el sistema de ROS propuesto es viable, permitiendo probar los algoritmos y la comunicación entre nodos de forma simple en un ordenador personal antes de implementar el sistema en el robot real.

Para esta prueba de concepto, se ha desarrollado una "réplica" del Jetbot en CoppeliaSim basada en el robot Pioneer, ya que ambos tienen configuración diferencial y el Pioneer

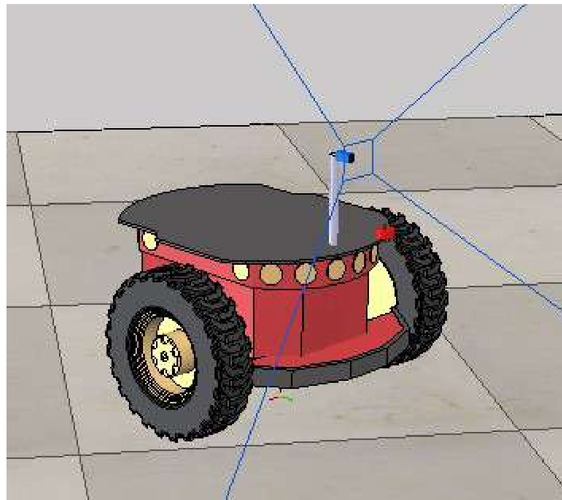


Figura 3.25 Robot en CoppeliaSim..

ya está en CoppeliaSim sin necesidad de pasos adicionales. A este robot Pioneer se le instala una cámara de características similares a la usada en el robot real (misma resolución, FOV amplio) para tratar de replicar las condiciones reales lo mejor posible. El robot y las características de la cámara se muestran en las figuras 3.25 y 3.25 respectivamente.

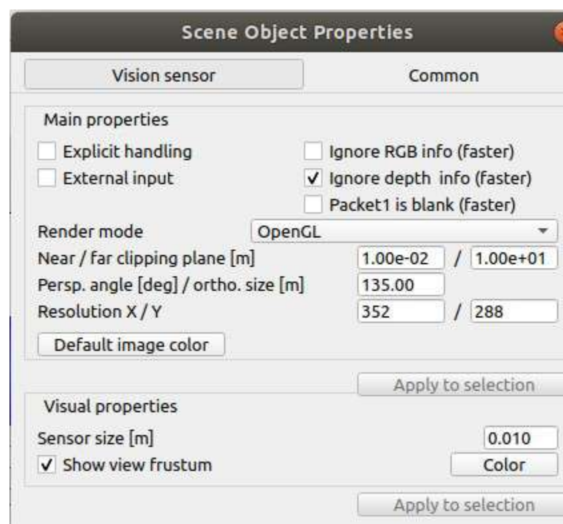


Figura 3.26 Ajustes de la cámara usadas para la simulación..

3.3 Aplicaciones de Software desarrolladas por el alumno.

Finalmente, se exponen aquellas aplicaciones de software desarrolladas para este trabajo, haciendo uso de las herramientas de software ya expuestas.

3.3.1 Aplicaciones para calibración y pruebas.

Una gran parte del código desarrollado se centra en permitir y facilitar los procesos de calibración y testeo del hardware. Sin estas aplicaciones como base, el sistema final sería imposible de implementar.

Calibración de motores.

El primer obstáculo a la hora de utilizar este robot es la calibración de sus motores. Cuando se controla un motor de corriente continua este se controla según la corriente que se le da, girando más rápido a más corriente reciba. Para esta aplicación se va a modelar esta corriente con un parámetro “y“ comprendido entre -1 y 1, donde:

- *“y = 1“* implica que el motor recibe la máxima corriente que permite.
- *“y = -1“* implica que el motor recibe la máxima corriente que permite en sentido contrario, invirtiendo el sentido de giro.
- *“y = 0“* implica que el motor no recibe corriente.

En una primera aproximación al problema de controlar un motor de continua, podría parecer una buena idea utilizar este parámetro “y“ como señal de control. Sin embargo, existen dos problemas con esto:

- El primero es que la corriente no se relaciona directamente con la velocidad real del robot. Estos motores, cuando tienen que mover una carga, tienen un umbral de corriente mínima para poder empezar a desplazarse. Por ejemplo, cuando “y = 0.1“, lo más probable es que el motor no se mueva en absoluto. Esto es un problema pues en numerosas ocasiones un controlador buscará hacer movimientos lentos y para ello generará una señal de actuación pequeña. Si la señal de actuación es directamente “y“, estos pequeños valores no moverán el robot en absoluto.
- El segundo viene derivado del hecho de que no existen dos motores exactamente iguales. Para distintos motores, un mismo valor de “y“ va a proporcionar distintas velocidades. De ahí surge la necesidad de tener alguna forma de "ajustar" ese valor de “y“ de forma individual a cada motor, ya que en la aplicación de interés (un robot diferencial) se dispone de dos motores distintos trabajando juntos.

Para corregir estos problemas con el parámetro “y“, se propone crear la señal de control “x“: un valor intermedio que representa la velocidad deseada en una escala del -1 al 1 para ambos motores, donde:

- *“x = 1“* implica que ambos motores giran a su velocidad máxima.
- *“x = -1“* implica que ambos motores giran a su velocidad máxima en sentido inverso de giro.

- *“x = 0“* implica que ambos motores están parados.

Este parámetro “x“ permite calcular la corriente “y“ que se le envía a cada motor a partir de la expresión “y = alpha * x + beta“ para “x > 0“ y “y = alpha * x - beta“ para “x < 0“, donde “alpha“ y “beta“ son parámetros propios de cada motor. El interés de obtener el parámetro “y“ de este modo radica en los parámetros “alpha“ y “beta“, que permiten personalizar la respuesta de cada motor. “beta“ define ese umbral mínimo de tensión necesaria para mover el robot mientras que “alpha“ permite modificar la pendiente con la que sube “y“ para un valor de “x“ concreto. Estos ajustes permiten que, en teoría, para una misma entrada “x = 0.1“ introducida en ambos motores, estos se muevan a la misma velocidad, que debería ser aproximadamente un décimo de su velocidad máxima.

En la figura 3.27 se muestra una gráfica con la relación entre “x“ e “y“.

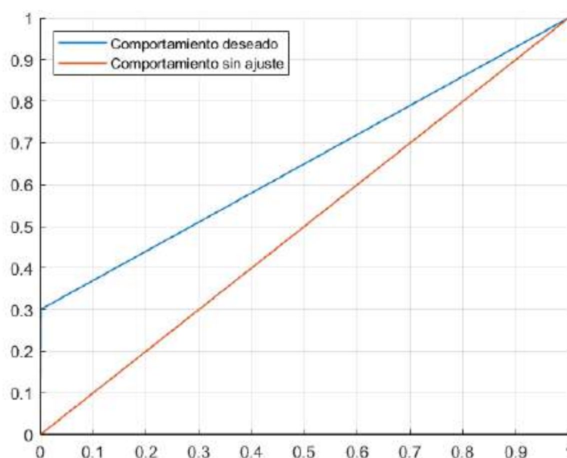


Figura 3.27 Relación entre ”x” e ”y” con y sin ajuste..

La calibración consiste en encontrar esos valores de “alpha“ y “beta“ que permiten a ambos motores moverse aproximadamente a la misma velocidad para un mismo valor de entrada “x“. Para encontrar “beta“, ha de buscarse el mínimo valor que hace que cada motor se mueva. Para “alpha“, se busca que ambos motores respondan igual a una misma entrada “x“. Para poder realizar este proceso de forma cómoda, se ha desarrollado un "notebook de Jupyter" que explica el proceso paso a paso e implementa una interfaz sencilla (figura 3.28) para que el usuario pueda probar distintos valores de “alpha“ y “beta“ de forma interactiva.

No obstante, hay un gran problema con este proceso de calibración: las librerías utilizadas para controlar el Jetbot. En ellas, “alpha“ y “beta“ solo sirven para ajustar una función lineal y positiva, lo que impide detener el motor dándole a x un valor de 0 o utilizar correctamente la marcha atrás. En la figura 3.29 se muestra una comparativa entre un comportamiento correcto para todo el intervalo [-1,1] y el de la librería:

Este error en las librerías posiblemente se deba a que el Jetbot nunca fue diseñado para un control fino. La solución a este problema se detalla en el siguiente apartado.

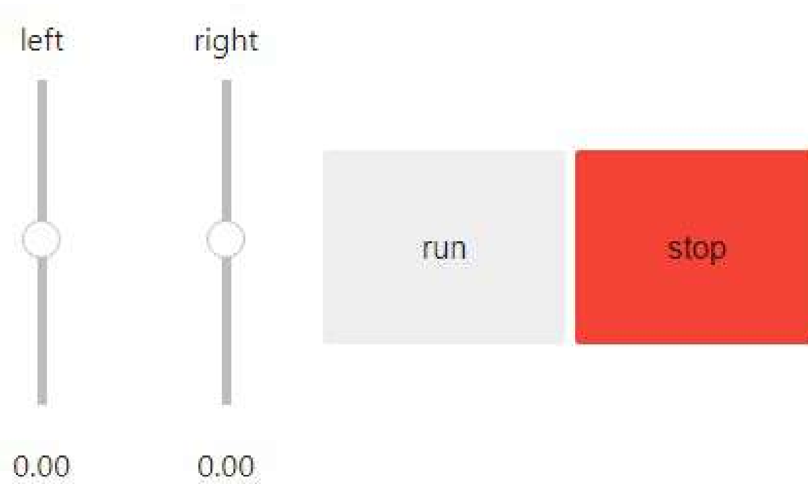


Figura 3.28 Interfaz usada para calibrar “alpha“ y “beta“.

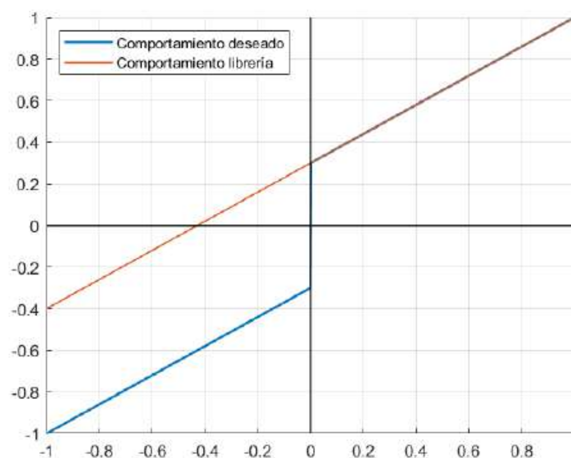


Figura 3.29 Calibración ofrecida por la librería "default" frente a la correcta.

Librerías de control.

Para solucionar el problema expuesto en el apartado anterior con las librerías de control predeterminadas del Jetbot, se han desarrollado unas nuevas librerías que introducen una calibración mas acertada. Estas librerías tendrán nuevos parámetros de calibración “alpha“, “beta“ y “gamma“ para cada motor.

El objetivo es crear un nuevo método que reciba valores de “x“ entre -1 y 1 los cuales de comporten siguiendo las siguientes especificaciones:

- Para valores en el intervalo “[-gamma, gamma]“, el motor no se moverá, es decir, recibirá un 0. Esto sirve de medida de seguridad por si los joystick estan algo descentrados, por lo que los valores de gamma seran muy pequeños, 0.01 o 0.02.
- Para el resto de valores positivos, el motor recibirá un valor que responde a la función “y = alpha * (x - gamma) + beta“.

- Para el resto de valores negativos, el motor recibirá un valor que responde a la función "y = alpha * (x + gamma) - beta".

Este comportamiento se muestra en la figura 3.30:

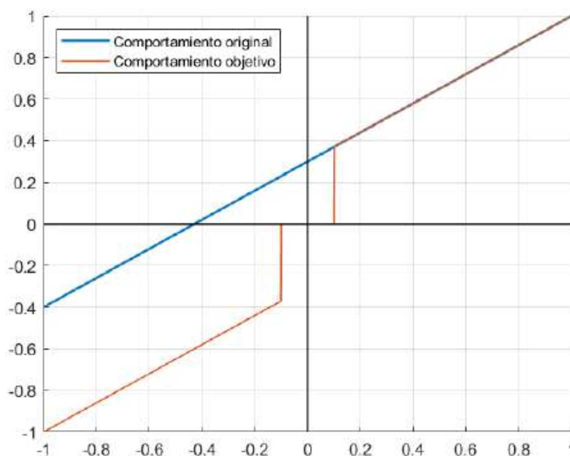


Figura 3.30 Calibración deseada para las nuevas librerías.

Siguiendo el esquema de las librerías originales, esta librería tiene dos componentes:

- Una clase Motor, que implementa este comportamiento de forma individual para cada motor. Aquí está la mayor diferencia entre las librerías original y modificadas, pues aquí se implementa el comportamiento definido en la figura 3.30.

- Una clase Robot que implementa el comportamiento para el robot completo, llamando a dos instancias de la clase motor.

Esta librería es la que se utiliza en el resto del proyecto para controlar el robot.

Librerías para manejo de cámara.

Para el manejo de la cámara se utilizará una librería especializada, que utiliza tanto *OpenCV* como *GStreamer* para facilitar la lectura de la cámara. La librería dispone de un método *"start_camera(capture_width, capture_height, framerate, pre_proc)"* que devuelve un "handler" a la cámara, es decir, un objeto que se puede utilizar para obtener la imagen de la cámara. Este "handler" habrá sido creado para recoger la imagen directamente de una "pipeline" de GStreamer personalizable que se encarga de:

- Leer la imagen de la cámara con una resolución y tasa de fotogramas dados como parámetro, aunque de serie están definidos como 320x240 píxeles a 30 fotogramas por segundo.

- Aplicar un preprocesamiento a la imagen en caso de que se especifique en la llamada al método. Este preprocesamiento le realizan un filtro de medianas y un filtro gaussiano

a la imagen que se captura, eliminando posible ruido introducido por la cámara a costa de perder nitidez en la imagen filtrada.

Este comportamiento puede modificarse al llamar al método de la librería, dando los parámetros deseados como argumentos de la función.

Teleoperación.

A fin de probar el movimiento del robot y la captura de imagen, se desarrolla una aplicación de teleoperación cuyo fin es controlar el robot y transmitir el vídeo capturado por la cámara, ambas cosas por medio de la red Wi-Fi local.

Esta aplicación permite el control del robot de forma remota y tiene dos versiones: una utilizando el teclado del ordenador y otra utilizando un controlador para videojuegos. La versión de operación con teclado utiliza las teclas "w", "a", "s" y "d" para operar el robot, leyéndose por medio de una función de *OpenCV*. Para la operación con controlador, se utiliza la librería *"Gamepad"* usando sus modos asíncrono y de eventos simultáneamente: el modo asíncrono se usa para leer el valor de los joystick de forma constante para manejar el robot mientras que el modo de eventos se usa para detectar pulsaciones de un botón concreto que provoca que el robot de media vuelta.

Para recibir la imagen se utiliza una "pipeline" simple de GStreamer, utilizando el plugin "udpsink" para emitir la imagen utilizando el protocolo UDP. Esto permite transmitir la imagen por medio de la red local sin necesidad de configuración adicional. La pipeline que se lanza desde el Jetbot es: "v4l2src device=/dev/video0 ! video/x-raw, width=320, height=240, framerate=30/1 ! videoconvert ! x264enc tune=zerolatency bitrate=500 speed-preset=superfast ! rtph264pay ! udpsink host=<IP-DEL-ORDENADOR> port=5000".

3.3.2 Aplicaciones para el sistema final.

Figura 3.31 Objetivos a detectar por el sistema.

El objetivo final de este proyecto es el de implementar un sistema de ROS que haga uso de visión artificial. La aplicación desarrollada consiste en un sistema capaz de identificar unos "objetos" como los de la figura 3.31 y perseguir uno de ellos, que se designa el "objetivo", mientras esquiva al resto. Esa designación de objetivo y obstáculo se realiza según el color del objeto y será el usuario quien decida que color es el objetivo.

En este apartado se detalla como funciona este sistema, explicando sus partes (nodos) una a una antes de exponer el sistema completo. Algunos nodos tienen versiones para simulación y para el caso real, en cuyo caso se detalla cada una.

Nodo de Imagen.

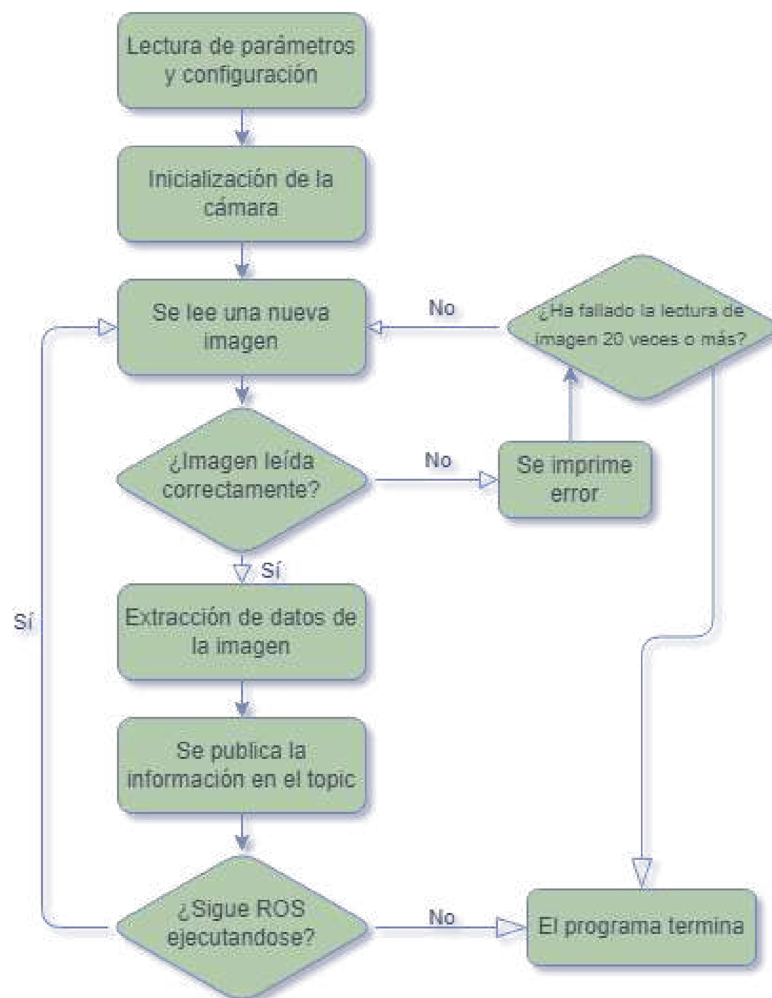


Figura 3.32 Comportamiento del nodo de imagen.

El nodo de imagen es el encargado de la obtención de datos a partir de la imagen obtenida por la cámara. Basándose en algoritmos de percepción es capaz de localizar los objetivos y diferenciarlos entre sí, obteniendo una estimación de su posición relativa al robot.

Antes de diseñar el nodo, son necesarias varias hipótesis:

- Los objetos a detectar son siempre cilindros de color uniforme rojo, verde o azul.
- En caso de que hayan varios objetos del mismo color, solo resultará de interés el más cercano, ya que los obstáculos más peligrosos siempre serán los más cercanos.
- La posición de cada objeto se definirá con dos parámetros: la posición horizontal del objeto en la imagen, expresada en píxeles; y la distancia del objeto al robot, que se estima con el área que ocupa el objeto en la imagen.
- Las condiciones de iluminación no cambian durante el experimento.

Partiendo de dichas hipótesis, se desarrolla el programa.

Como se expone en la figura 3.32, tras la lectura de parámetros e inicialización de la cámara, esta se lee de forma recurrente en intervalos regulares. Si no hay ningún error con la imagen, se extraen los datos de la imagen por medio una umbralización por colores utilizando umbrales específicos para cada color a detectar: verde, azul y rojo. La umbralización devuelve una máscara de píxeles donde 1 es el objeto u objetos detectados y 0 es todo lo demás. De esa mascara se extraen los datos del objeto detectado: posición horizontal a partir del centroide del objeto y distancia al robot a partir de su área. En el caso de que se detecten varios objetos del mismo color, solo se extraen los datos del más grande y se ignora el resto. Finalmente, los datos extraídos para cada color son enviados al nodo central por medio de ROS, publicados en un topic.

Existen dos versiones de este programa:

- *La versión para simulación:* Obtiene la imagen de un topic de ROS, donde la simulación publica la imagen de la cámara simulada. Esta imagen no tiene ningún tipo de ruido al ser obtenida de una cámara ideal.
- *La versión para el entorno real:* Obtiene la imagen de la cámara utilizando las librerías de manejo de imagen desarrolladas. Esta imagen se preprocesa con la librería para evitar el ruido a costa de nitidez, un intercambio beneficioso para esta aplicación.

Nodo de Control.

El nodo de control es el encargado de actuar sobre los motores según las ordenes de velocidad que le lleguen, convirtiendo un comando de velocidad lineal y angular deseadas en valores individuales de velocidad para cada motor.

Antes de diseñar el nodo, son necesarias varias hipótesis:

- La calibración de los motores ha sido llevada a cabo previamente.
- Para el caso real, la corriente que se les pasa a los motores se relaciona linealmente con la velocidad con la que estos mueven el robot.

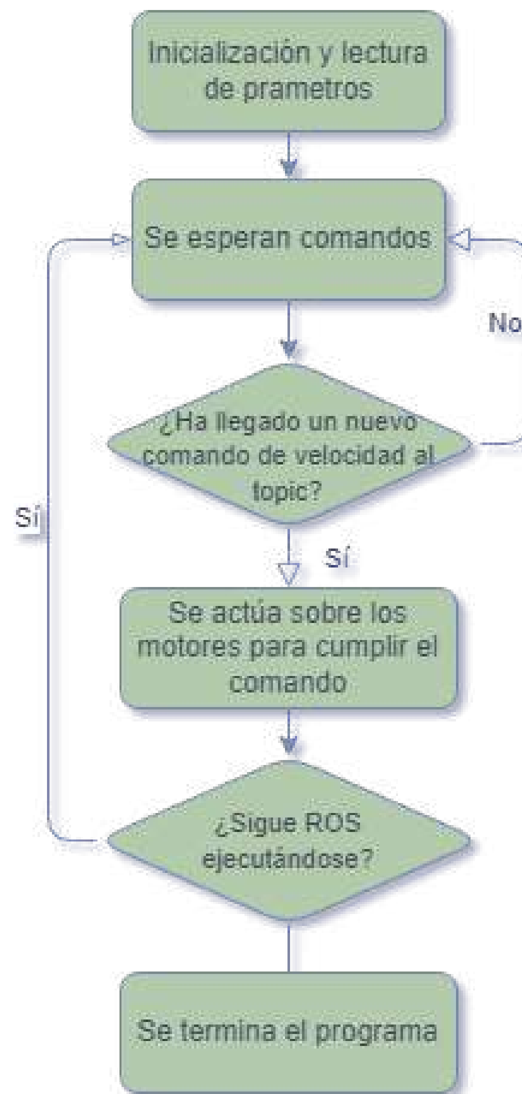


Figura 3.33 Comportamiento del nodo de control.

- Aunque se hable de velocidad, ninguno de los valores que se manejan representan una unidad real de medida de velocidad. Sin embargo, se asume que las acciones de control son directamente proporcionales a la velocidad real del robot.

Partiendo de dichas hipótesis, se desarrolla el programa.

Como se expone en la figura 3.33, tras la lectura de parámetros e inicialización de la actuación, el programa espera comandos de velocidad, que han de llegar en forma de publicaciones a un topic concreto. Cuando llega un nuevo comando de velocidad, este es procesado por una función que traduce ese comando a entradas de velocidad para cada motor debidamente saturadas.

Existen dos versiones de este programa:

- *La versión para simulación:* Para actuar sobre los motores del robot simulado cada motor tiene asignado un topic, donde se publica la velocidad deseada para cada uno.

- *La versión para el entorno real:* Para actuar sobre los motores se utilizan las librerías de control expuestas anteriormente. En este caso, en lugar de dar un valor de velocidad, se da un parámetro “x´´ que modela cuanta corriente llega a cada motor y que se ha supuesto que se relaciona con la velocidad de forma lineal.

Nodo Central.

El nodo central es el encargado de la toma de decisiones. Toma información de la posición de los objetos del nodo de imagen y la usa para decidir que orden de velocidad mandarle al nodo de control. Además, actúa como interfaz entre la aplicación y el usuario, poniendo a su disposición servicios de ROS.

Antes de diseñar el nodo, son necesarias varias hipótesis:

- Sólo existen los objetos definidos en la figura 3.31 en el escenario en el que el robot se mueve. Se ignoran paredes y no habrá ningún otro tipo de obstáculo.

- Se asume que nunca habrán dos objetos demasiado cerca los unos de los otros, siendo imposible que dos objetos tengan coordenadas demasiado similares.

Partiendo de dichas hipótesis, se desarrolla el programa.

Como se expone en la figura 3.34, tras la lectura de parámetros e inicialización de los topic y servicios de ROS, el programa hace dos cosas:

La primera es poner a disposición del usuario dos servicios de ROS:

- *Servicio de inicio:* Permite al usuario decidir cuando la aplicación comienza a actuar. Mientras no se llame a este servicio y se le mande un "start: True", el nodo central no publicará ordenes de movimiento. Además, este servicio permite al usuario elegir de que color será el objetivo que el robot ha de perseguir cuando inicie su movimiento. Este mismo servicio también permite detener la aplicación en cualquier momento.

- *Servicio de configuración:* Permite al usuario cambiar la configuración de la aplicación en cualquier momento, pudiendo cambiar el color del objetivo o las velocidades lineales y angulares máximas.

La segunda y más importante es esperar información del nodo de imagen y procesarla una vez llega. Esa información se utilizará posteriormente para construir un mapa local que representa el espacio que tiene el robot delante de sí mismo y a sus laterales, distinguiendo en el mapa entre "objetivo" y "obstáculos". El "objetivo" siempre se coloca en el mapa, incluso si no es visible. En caso de que el "objetivo" no sea visible, en el mapa se coloca cerca de la última posición conocida del mismo o se asume que debe estar muy lejos y a la derecha.

Una vez construido el mapa, este se utiliza para decidir la trayectoria: si no hay obstáculos bloqueando al objetivo, la trayectoria se define recta hacia el objetivo. Sin embargo, si hay algún obstáculo bloqueando al objetivo, el programa busca la trayectoria que le permita

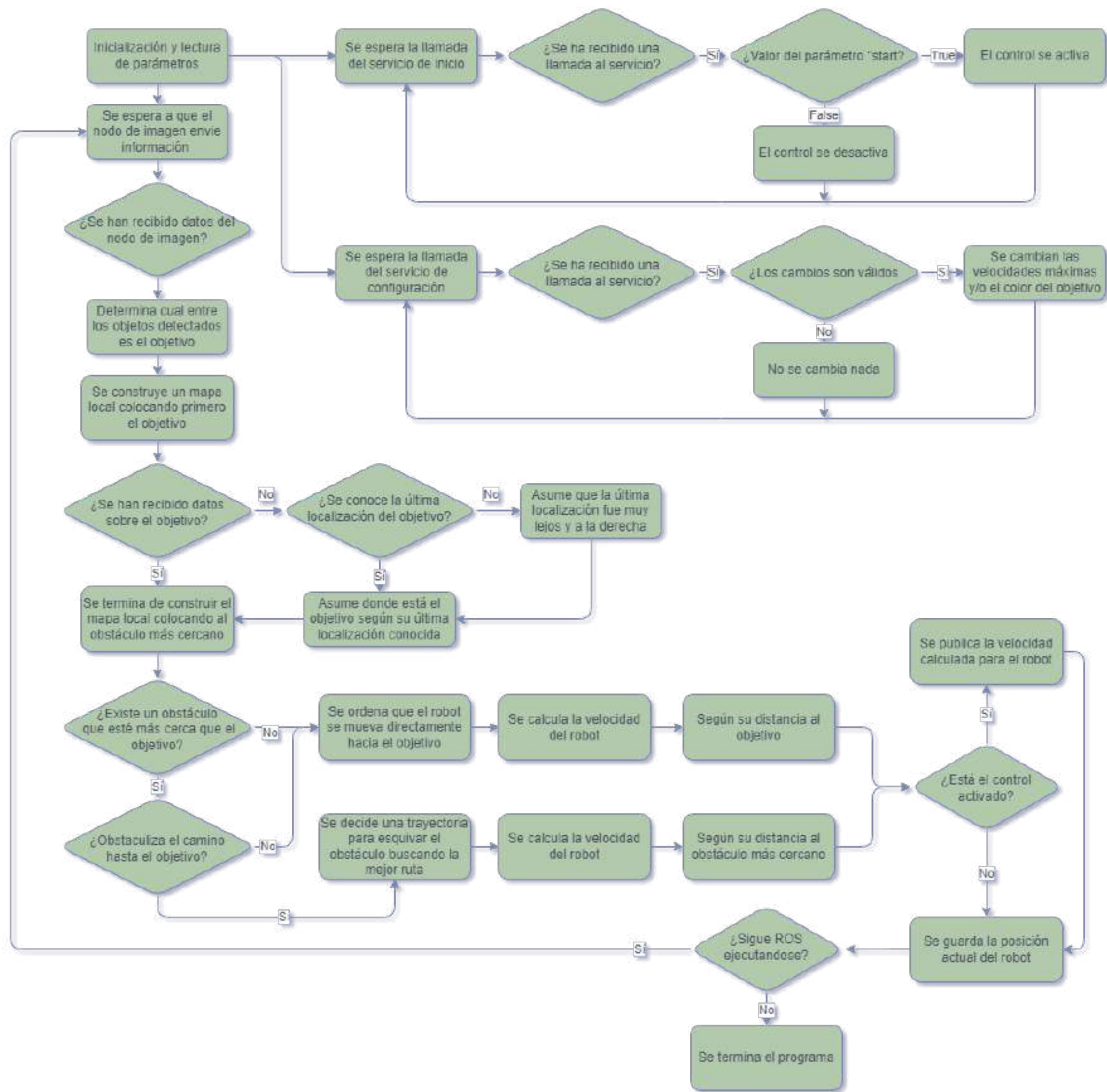


Figura 3.34 Comportamiento del nodo central.

esquivar el obstáculo sin perder de vista al objetivo. Esto define cuanto ha de girar el robot, o lo que es lo mismo, la velocidad angular que se comandará. La velocidad lineal se decide teniendo en cuenta la distancia hasta el objetivo o la distancia al obstáculo más cercano, dependiendo de que esté más cerca y de si hay algún obstáculo que se interponga directamente entre el robot y el objetivo. Se prioriza evitar choques.

Finalmente, si el control esta activado los comandos de velocidad calculados se envían al nodo de control, que actúa sobre el movimiento del robot.

El mismo nodo central se utiliza tanto para el caso real como para la simulación. Esto es producto del interés en "modularizar" al máximo la aplicación y permitir que sea fácilmente exportable a otros tipos de robots u entornos de simulación.

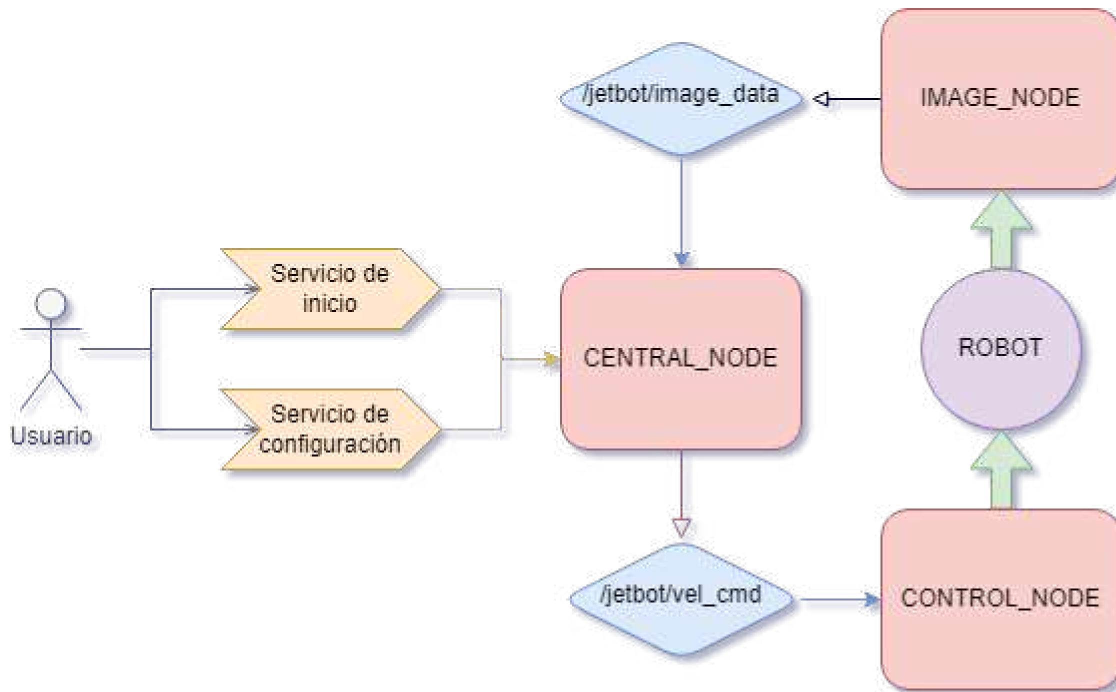


Figura 3.35 Sistema completo.

Sistema completo.

Todos los nodos expuestos tienen funcionalidades relativamente independientes, pero para poder llevar a cabo su objetivo necesitan un modo de poder comunicarse entre sí. Aquí es donde entra ROS, proporcionando dos sistemas para la comunicación entre procesos: los topics y los servicios. En esta aplicación se utilizan ambos. Como se puede ver en la figura 3.34, se utilizan los topics para comunicar los procesos entre sí, mientras que los servicios sirven como una interfaz para el usuario.

Este sistema se arranca de forma simultanea, estableciendo primero los topics y servicios antes de comenzar la operación de los nodos. El sistema comienza en un estado de bloqueo impuesto por el nodo central, que impide que el robot se mueva. Una vez el usuario llame al servicio de inicio, el nodo central comenzará a publicar comandos en el topic que le comunica al nodo de control y la aplicación comenzará a manejar el robot para llevarle hacia el objetivo.

Gracias a ROS, en el momento en el que se pide la terminación del "roscore", un programa maestro que gestiona el sistema completo de ROS, todos los nodos terminarán por sí mismos su ejecución de forma limpia y ordenada.

Los resultados de este sistema se probarán tanto en simulación como en un sistema real en diversos experimentos que se mostrarán en el siguiente apartado.

Calibración de sistema de visión.

La base del nodo de imagen de la aplicación final es una segmentación por color. Este método requiere que se definan unos umbrales de color entre los cuales se encuentran los objetos que se desean detectar, los cuales han de definirse previamente. Este proceso de "calibración" de los umbrales puede llegar a ser muy tedioso y repetitivo, ya que unas condiciones de iluminación distintas o un entorno de pruebas con un fondo diferente pueden alterar de forma significativa la validez de estos umbrales. Además, es necesario conocer el área que ocupan los objetivos a distintas distancias, lo que puede variar si varían los objetivos o la cámara.

Es por esto que se ha desarrollado una aplicación para calibrar estos umbrales, simplificando el proceso con una interfaz gráfica y un algoritmo de "calibración asistida". La interfaz consiste en tres ventanas: una con la imagen capturada por la cámara, otra con la máscara obtenida por la segmentación por colores y otra con botones deslizantes que permiten ajustar los valores del umbral directamente. Clicando en la primera imagen, se podrán seleccionar "puntos" de la imagen que se desea que se queden dentro de la segmentación. Al clicar en al menos 3 puntos, el algoritmo de calibración asistida calcula los umbrales que mejor mantendrán los puntos seleccionados dentro de la máscara. Posteriormente, se pueden utilizar los botones deslizantes para ajustar de forma fina la máscara.

Además, utilizando el teclado, se puede ordenar al programa que guarde los datos para ser usados más adelante o interactuar con la calibración asistida:

- Pulsando "b", "g" o "r": Esto guardará los umbrales actuales según el botón pulsado, siendo cada uno correspondiente a un color ("b" para azul, "g" para verde y "r" para rojo). Esto borra todos los puntos guardados por la calibración asistida y reinicia los umbrales.

- Pulsando "n", "m" o "f": Esto guardará el área actual del objeto según el botón pulsado, siendo cada uno correspondiente a una distancia ("n" para un objeto cercano, "m" para un una distancia media y "f" para distancias lejanas).

- Pulsando "s": Esto guardará toda la información recogida en un archivo de configuración que podrá leer el "nodo de imagen" de la aplicación final.

- Pulsando "retroceso": Esto borrará el último punto guardado por la calibración asistida, en caso de equivocación al clicar.

El fichero de configuración generado se guarda en formato JSON para ser leído posteriormente por los nodos que lo requieran y contiene los umbrales para cada color y el área que se supone que tiene un objeto cuando se encuentra a distintas distancias.

Capítulo 4: Experimentos y resultados

En este capítulo se detallan los experimentos realizados con el dispositivo, así como los resultados obtenidos, dividiendo cada experimento en tres apartados:

- Descripción del experimento: Descripción del entorno y condiciones en el que se desarrolla el experimento, en que consiste paso a paso y cuales son sus objetivos.
- Resultados: Resultados obtenidos del experimento, expuestos de la forma más objetiva posible.
- Conclusiones: Conclusiones extraídas del experimento en función de los resultados expuestos. De ser de relevancia, también se expone para que se utilizan dichas conclusiones y que relevancia tienen en el resultado final.

De este modo se espera proporcionar al lector una visión clara de los resultados de este trabajo.

4.1 Pruebas de hardware.

En este apartado se exponen los experimentos relacionados con el hardware, que no están directamente relacionados con el sistema final.

4.1.1 Calibración de motores.

Descripción del experimento

En este experimento se realiza una calibración de los motores. La calibración consiste en los valores de "alpha´´ y "beta´´ que permiten a ambos motores moverse aproximadamente

a la misma velocidad para un mismo valor de entrada “x´´.

Para realizar este experimento se utiliza el archivo “Calibration.ipynb´´, un *jupyter notebook* interactivo. El robot se colocará en el mismo entorno y suelo en el que se realizarán el resto de experimentos.

El experimento consiste en dos fases:

1. Encontrar el valor “beta´´: Para ello, partiendo de los motores sin calibrar, se prueban valores hasta hallar los valores mínimo que permite que ambos motores se muevan de forma simultanea hacia delante, sin desviarse. Los valores de “beta´´ elegidos se introducen en la calibración.

2. Encontrar el valor de “alpha´´: Partiendo de los motores con las “betas´´ ya calibradas, se calculan unos valores iniciales para “alpha´´ entorno a los que se realiza la calibración: aquellos que para una entrada “x = 1´´, “y = 1´´. A partir de los valores iniciales, se prueba la siguiente serie de valores de entrada: [-0.8, -0.3, -0.1, 0.1, 0.3, 0.8]. Para todos esos valores, el robot deberá moverse recto hacia delante o atrás, sin desviarse. En caso de que se desvié, se disminuirá el “alpha´´ del motor que se haya adelantado y se volverá a probar la serie de valores. Este proceso se repetirá de forma iterativa hasta encontrar los valores de “alpha´´ ideales.

Resultados

En la tabla 4.1 se muestran los resultados del experimento:

Tabla 4.1 Resultados del experimento de calibración de motores.

Prueba	Descripción	Resultado
Encontrar valor de beta	Se prueban valores de beta hasta que ambos motores comienzan a moverse de forma simultanea hacia delante.	No satisfactorio.
Encontrar valor de alpha	Se prueban distintas acciones de control para distintas alphas hasta que ambos motores se mueven de forma simultanea hacia delante para todas las acciones de control.	Parcialmente satisfactorio.

Aunque los motores pueden ser calibrados con el método descrito y se logran los objetivos definidos, la beta mínima es un valor demasiado alto y el motor se mueve a mucha velocidad incluso a su velocidad mínima.

Además, independientemente de las variaciones en alpha, el robot se desvía inevitablemente ya que las ruedas deslizan y los ejes de los motores no son tan rígidos como deberían. Sin embargo, se desestima la importancia de esas desviaciones.

Conclusiones

La calibración se ha llevado a cabo correctamente, pero los motores carecen de la fuerza para mover el robot de forma precisa y controlada. Esto provoca que para la mínima señal de control los motores se muevan a una velocidad elevada que se teme influya negativamente en la aplicación final.

4.1.2 Calibración de umbrales para detección de color

Descripción del experimento

En este experimento se probará la aplicación para la calibración de la umbralización por color, así como el desempeño de la propia técnica de umbralización.

Para realizar este experimento el robot se colocará en el mismo entorno y suelo en el que se realizan el resto de experimentos, frente a los mismos objetivos (mostrados en la figura) que se usarán para los experimentos de la aplicación final. Para tener acceso a la interfaz gráfica, el robot se conectará por HDMI a un monitor. Se utilizará el script “color_calibration.py´´.

Los objetivos del experimento son:

- Probar la interfaz gráfica.
- Probar la calibración asistida.
- Probar el ajuste fino de la calibración usando los “sliders´´.
- Probar que la umbralización solo detecta los objetivos para los 3 colores.
- Probar que el archivo generado es correcto y se corresponde con los valores encontrados en la aplicación.

El experimento consiste en:

1. Iniciar el programa.
2. Usar la calibración asistida para delimitar un objetivo rojo.
3. Usar la calibración fina hasta que la umbralización solo detecta los objetivos rojos.
4. Guardar los valores.
5. Repetir el proceso para todos los colores.
6. Colocar un objeto a las distancias de interés (cerca, media distancia y lejos) y guardar las áreas de cada objeto.

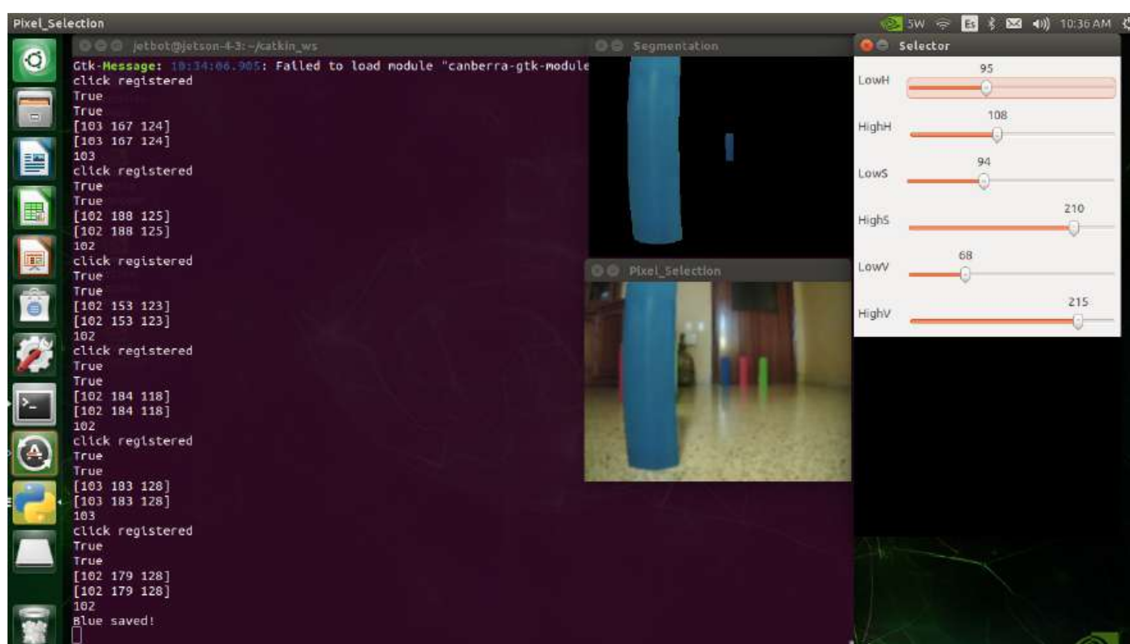


Figura 4.1 Aplicación de calibración de imagen en acción.

7. Guardar el archivo de configuración y comprobar que es correcto.

Resultados

En la tabla 4.2 se muestran los resultados del experimento:

Tabla 4.2 Resultados del experimento de calibración de cámara.

Prueba	Descripción	Resultado
Calibración de objeto rojo	Se calibra el algoritmo para segmentar un objeto rojo.	Satisfactorio.
Calibración del resto de colores	Se calibra el algoritmo para segmentar un objeto rojo, uno azul y uno verde.	Satisfactorio.
Distancias de interés	Se calibran las distancias de interés para un objetivo.	Satisfactorio.
Archivo de configuración	Se guarda el archivo de configuración y se comprueba que es correcto.	Satisfactorio.

La calibración se lleva a cabo correctamente y se comprueba que la segmentación funciona perfectamente. La interfaz responde correctamente y los datos se guardan con el formato adecuado y en el lugar adecuado. Un ejemplo de la interfaz se observa en la figura 4.1

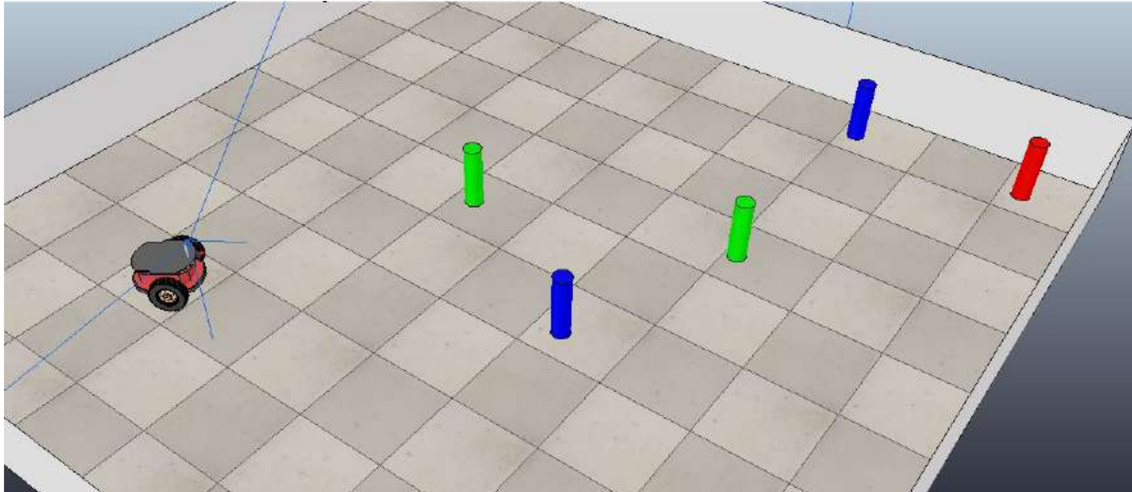


Figura 4.2 Entorno de simulación usado en los experimentos..

Conclusiones

La cámara puede calibrarse de forma rápida y sencilla con este programa, ahorrando tiempo y permitiendo probar la capacidad del algoritmo de segmentación para operar en las condiciones de iluminación del experimento. Además, el archivo de configuración generado prueba ser una buena forma de facilitar el uso para un usuario, ya que acaba por completo con la necesidad de modificar código para ejecutar la aplicación.

4.2 Aplicación en el entorno de simulación.

A continuación, se describen las diversas pruebas llevadas a cabo en el entorno de simulación CoppeliaSim. Para estos experimentos, se hará uso de un entorno simulado que consiste en un recinto cerrado con varios objetos de los tres colores primarios que van a ser detectados desperdigados por el recinto, que pueden moverse durante la ejecución de la simulación. El entorno se muestra en la figura 4.2.

4.2.1 Prueba de nodo de imagen en simulación.

Descripción del experimento

Experimento en el que se comprueba el correcto funcionamiento del nodo de imagen, en su variante para simulación. Para ello, se utiliza el modo de "verbose" que imprime en pantalla el resultado de los algoritmos de visión, así como el software RViz para tener una representación visual de dichos resultados.

El experimento se realizará en el entorno de simulación mostrado en la figura 4.2 en CoppeliaSim. El código se ejecutará usando un archivo *exp_imagen_sim.launch* que ejecuta el simulador, RViz y el nodo de imagen.

El objetivo del experimento es comprobar que el nodo de imagen puede:

- Diferenciar objetos rojos, verdes o azules del resto del entorno.
- Diferenciar objetos rojos, verdes y azules entre sí.
- Obtener la posición horizontal (eje x de la imagen) de los objetos detectados, expresado en píxeles.
- Obtener la distancia del objeto detectado hasta la cámara, en una escala del 1 (muy cerca) al 4 (muy lejos).
- Diferenciar cual es el objeto más cercano entre varios objetos del mismo color.

El experimento consiste en:

1. Iniciar el sistema de ROS, comprobando que toda la información se muestra de forma correcta.
2. Colocar un objeto de cada color, uno a uno, delante de la cámara. Comprobar que lo detecta correctamente.
3. Colocar dos objetos del mismo color delante de la cámara. Repetir para todos los colores. Comprobar que detecta correctamente al más cercano.
4. Colocar dos objetos de distinto color delante de la cámara. Repetir para todas las combinaciones. Comprobar que los detecta correctamente a ambos.
5. Colocar tres objetos de distinto color delante de la cámara. Repetir a distintas distancias. Comprobar que los detecta correctamente todos.
6. Colocar dos objetos de cada color delante de la cámara a la vez. Repetir a distintas distancias. Comprobar que los detecta correctamente los más cercanos de cada color.

Resultados

En la tabla 4.3 se muestran los resultados del experimento:

Tabla 4.3 Resultados del experimento del nodo de imagen en simulación.

Prueba	Descripción	Resultado
Inicio e interfaz	Se inicia el programa, comprobando que todos los nodos funcionan de forma correcta y la información es visible.	Satisfactorio.
Detección de objetivo rojo	Se comprueba la detección de un objetivo rojo a distintas distancias.	Satisfactorio.
Detección de objetivo verde	Se comprueba la detección de un objetivo verde a distintas distancias.	Satisfactorio.
Detección de objetivo azul	Se comprueba la detección de un objetivo azul a distintas distancias.	Satisfactorio.
Detección de múltiple objetivo rojo	Se comprueba la detección de dos objetivos rojos a distintas distancias.	Mayoritariamente satisfactorio.
Detección de múltiple objetivo verde	Se comprueba la detección de dos objetivos verde a distintas distancias.	Mayoritariamente satisfactorio.
Detección de múltiple objetivo azul	Se comprueba la detección de dos objetivos azul a distintas distancias.	Mayoritariamente satisfactorio.
Detección de objetivos rojo y azul	Se comprueba la detección de dos objetivos rojo y azul a distintas distancias.	Mayoritariamente satisfactorio.
Detección de objetivos rojo y verde	Se comprueba la detección de dos objetivos rojo y verde a distintas distancias.	Mayoritariamente satisfactorio.
Detección de objetivos azul y verde	Se comprueba la detección de dos objetivos azul y verde a distintas distancias.	Mayoritariamente satisfactorio.
Detección de tres objetivos	Se comprueba la detección de tres objetivos de distinto color a distintas distancias.	Mayoritariamente satisfactorio.
Detección de seis objetivos	Se comprueba la detección de seis objetivos de distinto color (dos azules, dos rojos, dos verdes) a distintas distancias.	Mayoritariamente satisfactorio.

El programa funciona bien de forma general, aunque el efecto ojo de pez que introducen las lentes con mayor FOV distorsiona las formas de los objetivos que se encuentran al borde de la cámara. Esto impide que se detecte correctamente la distancia hacia el objetivo.

La detección de objetivos como tal funciona perfectamente. El programa detecta perfectamente donde está cada objetivo en el eje X de la imagen, distinguiendo los tres colores principales entre sí. La detección de distancia hasta el objetivo funciona razonablemente bien, a excepción del problema ya expuesto.

La imagen capturada por la cámara en simulación se muestra en la figura 4.3. En la imagen se observan puntos de colores que muestran el resultado de la detección de color. Se ve como el programa detecta correctamente el objetivo más cercano de cada color. Los valores concretos, posición en el eje X y área del objetivo, se imprimen por consola.

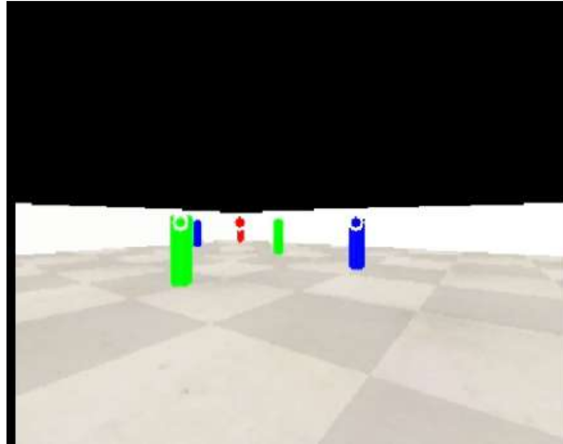


Figura 4.3 Imagen capturada por la cámara simulada.

Conclusiones

El experimento prueba la viabilidad de los algoritmos, que funcionan con suficiente frecuencia y sin error. Queda hacer pruebas con la cámara real, donde el ruido puede ser un problema serio. También se comprueba el riesgo de aumentar demasiado el FOV, aunque se estima que no es un problema grave para el funcionamiento de la aplicación ya que los obstáculos situados a los laterales no deberían ser un riesgo importante a considerar.

4.2.2 Prueba de nodo de control en simulación.

Descripción del experimento

Experimento en el que se comprueba el correcto funcionamiento del nodo de control, en su variante para simulación.

El experimento se realizará en el entorno de simulación mostrado en la figura 4.2 en CoppeliaSim, moviendo al robot por el entorno por medio del nodo de control. El código se ejecutará usando un archivo *exp_control_sim.launch* que ejecuta el simulador, RViz y el nodo de control.

El objetivo del experimento es comprobar que el nodo de control puede:

- Recibir correctamente ordenes publicadas en el topic *"/jetbot/vel_cmd"*.
- Traducir dichas ordenes a velocidades para la rueda derecha y la rueda izquierda.
- Aplicar las velocidades generadas en el robot simulado, haciendo que se mueva.

El experimento consiste en:

1. Iniciar el sistema de ROS, comprobando que toda la información se muestra de forma correcta.

2. Utilizar el comando de ROS "rostopic pub´´ para publicar ordenes en el topic *"/jetbot/vel_cmd"*. Por medio de esas ordenes, el robot deberá realizar los siguientes movimientos:

    - Movimiento recto hacia delante.
    - Movimiento recto hacia atrás.
    - Giro hacia la derecha sin moverse del sitio.
    - Giro hacia la izquierda sin moverse del sitio.
    - Movimiento hacia la derecha en círculos.
    - Movimiento hacia la izquierda en círculos.

Resultados

En la tabla 4.4 se muestran los resultados del experimento:

Tabla 4.4 Resultados del experimento del nodo de control en simulación.

Prueba	Descripción	Resultado
Inicio e interfaz	Se inicia el programa, comprobando que todos los nodos funcionan de forma correcta y la información es visible.	Satisfactorio.
Movimiento	Se comprueba el control en velocidad realizando diversos movimientos.	Satisfactorio.

El programa funciona a la perfección en simulación. Al ejecutarse y recibir ordenes, el programa imprime en la terminal información de interés, como la velocidad de cada rueda. Además, permite controlar el robot publicando comandos de velocidad lineal y angular en el topic correspondiente, que se traducen al movimiento deseado.

Conclusiones

El éxito en este experimento prueba la viabilidad de usar topics de ROS para controlar un robot diferencial y confirma que los cálculos que traducen el comando de velocidad a la velocidad de cada rueda son correctos y generan el movimiento deseado.

4.2.3 Prueba de sistema completo en simulación.

Descripción del experimento

Experimento en el que se comprueba el correcto funcionamiento del sistema completo, usando los nodos de imagen y de control en su variante para simulación.

El experimento se realizará en el entorno de simulación mostrado en la figura 4.2 en CoppeliaSim, haciendo que el robot persiga los objetos del color que se le indique. Para ejecutar el experimento, se usará un archivo *simulation.launch* que ejecuta el simulador, RViz y todos los nodos del programa.

El objetivo del experimento es comprobar que el nodo central puede:

- Iniciar el movimiento por medio de la llamada a un servicio de ROS.
- Llegar hasta el objeto rojo sin chocar contra otro objeto.
- Perseguir al objeto rojo cuando es movido en el simulador.
- Cambiar el color del objeto a perseguir durante la ejecución usando un servicio de ROS.
- Cambiar la velocidad máxima durante la ejecución usando un servicio de ROS.

El experimento consiste en:

1. Iniciar el sistema de ROS, comprobando que toda la información se muestra de forma correcta.
2. Llamar al servicio de inicio para que el movimiento del robot comience.
3. Permitir que el robot persiga un objeto de color rojo, esquivando al resto.
4. Una vez el robot halla llegado al objeto de color rojo, coger el objeto y desplazarlo manualmente, de modo que el robot lo persiga activamente durante unos segundos.
5. Acercar el objeto rojo hacia el robot. Observar si el robot intenta alejarse para evitar una colisión.
6. Llamar al servicio de configuración para que el robot pase a perseguir a un objeto de color azul. Esperar hasta que el robot haya encontrado el objeto y este observándolo.
7. Llamar al servicio de configuración para reducir la velocidad máxima del robot.
8. Coger el objeto azul y desplazarlo manualmente, de modo que el robot lo persiga activamente durante unos segundos.

Este experimento se repetirá tres veces: primero empezando con rojo y pasando a azul, luego empezando en azul y pasando a verde y por último empezando en verde y pasando a rojo. En cada una de estas iteraciones se probarán distintas distribuciones de objetos.

Resultados

En la tabla 4.5 se muestran los resultados del experimento:

Tabla 4.5 Resultados del experimento del sistema completo en simulación.

Prueba	Descripción	Resultado
Inicio e interfaz	Se inicia el programa, comprobando que todos los nodos funcionan de forma correcta y la información es visible.	Satisfactorio.
Servicio de inicio	Se comprueba que el movimiento del robot empieza al llamar al servicio de inicio.	Satisfactorio.
Localización de objetivo rojo	Se comprueba que el robot localiza el objeto rojo y lo alcanza.	Satisfactorio.
Persecución de objetivo rojo	Se comprueba que el robot persigue al objeto rojo en caso de que este sea movido.	Parcialmente satisfactorio.
Evasión de colisiones con objetivo rojo	Se comprueba que el robot se aleja del objetivo en caso de estar demasiado cerca.	Satisfactorio.
Cambio de color de objetivo	Se cambia el color objetivo al azul usando el servicio de configuración y se espera a que el robot lo encuentre.	Parcialmente satisfactorio.
Cambio de velocidad máxima	Se decrementa la velocidad máxima usando el servicio de configuración	Satisfactorio.
Persecución de objetivo azul	Se comprueba que el robot persigue al objeto azul en caso de que este sea movido.	Parcialmente satisfactorio.

El experimento ocurre de forma fluida y sin problemas graves. Son destacables ciertos comportamientos no deseados:

- El robot no es tan consciente de su entorno como debería y a menudo pasa demasiado cerca de los obstáculos, en ocasiones rozándolos.
- Cuando el objetivo se acerca demasiado (tras ser movido por alguien), el robot se aleja moviéndose en marcha atrás. Al no tener ningún tipo de sensórica en la parte trasera, el robot puede chocar contra obstáculos durante este movimiento.
- Si el objetivo queda bloqueado por un obstáculo de modo que el robot no pueda verlo, el algoritmo puede llegar a confundirse.

Todo lo relacionado con ROS funciona correctamente, los nodos se comunican entre sí

sin problemas usando topics y los servicios reaccionan y permiten cambiar los parámetros de la aplicación durante su ejecución.

Conclusiones

El resultado se considera correcto y, aunque se han encontrado ciertos problemas en la ejecución, ninguno se considera demasiado grave, pues se considera que dadas las limitaciones del sistema es un resultado razonable.

El programa se considera listo para ser implementado en el robot real: el algoritmo funciona correctamente y el nodo central se comunica correctamente con el resto de nodos, descartando problemas con ROS.

4.3 Aplicación en el robot real.

A continuación, se describen las diversas pruebas llevadas a cabo en un entorno real utilizando el Jetbot. Para estos experimentos se hará uso de una habitación cerrada con varios objetos de los tres colores primarios que van a ser detectados desperdigados por el recinto. En el entorno de pruebas hay varios objetos en el fondo que no son de interés y una iluminación desigual, además de paredes. El entorno se muestra en la figura .

4.3.1 Prueba de nodo de imagen en robot real.

Descripción del experimento

Experimento en el que se comprueba el correcto funcionamiento del nodo de imagen en un entorno real. Para ello, se utiliza el modo de "verbose" que imprime en pantalla el resultado de los algoritmos de visión, así como el software RViz para tener una representación visual de dichos resultados.

El código se ejecutará usando un archivo *exp_imagen_real.launch* que ejecuta RViz y el nodo de imagen. Para poder visualizar los resultados en RViz, se conecta al robot por HDMI un monitor.

El objetivo del experimento es comprobar que el nodo de imagen puede:

- Diferenciar objetos rojos, verdes o azules del resto del entorno.
- Diferenciar objetos rojos, verdes y azules entre sí.
- Obtener la posición horizontal (eje x de la imagen) de los objetos detectados, expresado en píxeles.

- Obtener la distancia del objeto detectado hasta la cámara, en una escala del 1 (muy cerca) al 4 (muy lejos).
- Diferenciar cual es el objeto más cercano entre varios objetos del mismo color.

El experimento consiste en:

1. Iniciar el sistema de ROS, comprobando que toda la información se muestra de forma correcta.
2. Colocar un objeto de cada color, uno a uno, delante de la cámara. Comprobar que lo detecta correctamente.
3. Colocar dos objetos del mismo color delante de la cámara. Repetir para todos los colores. Comprobar que detecta correctamente al más cercano.
4. Colocar dos objetos de distinto color delante de la cámara. Repetir para todas las combinaciones. Comprobar que los detecta correctamente a ambos.
5. Colocar tres objetos de distinto color delante de la cámara. Repetir a distintas distancias. Comprobar que los detecta correctamente todos.
6. Colocar dos objetos de cada color delante de la cámara a la vez. Repetir a distintas distancias. Comprobar que los detecta correctamente los más cercanos de cada color.

Resultados

En la tabla 4.6 se muestran los resultados del experimento:

Tabla 4.6 Resultados del experimento del nodo de imagen.

Prueba	Descripción	Resultado
Inicio e interfaz	Se inicia el programa, comprobando que todos los nodos funcionan de forma correcta y la información es visible.	Satisfactorio.
Detección de objetivo rojo	Se comprueba la detección de un objetivo rojo a distintas distancias.	Satisfactorio.
Detección de objetivo verde	Se comprueba la detección de un objetivo verde a distintas distancias.	Satisfactorio.
Detección de objetivo azul	Se comprueba la detección de un objetivo azul a distintas distancias.	Satisfactorio.
Detección de múltiple objetivo rojo	Se comprueba la detección de dos objetivos rojos a distintas distancias.	Satisfactorio.
Detección de múltiple objetivo verde	Se comprueba la detección de dos objetivos verde a distintas distancias.	Satisfactorio.
Detección de múltiple objetivo azul	Se comprueba la detección de dos objetivos azul a distintas distancias.	Satisfactorio.
Detección de objetivos rojo y azul	Se comprueba la detección de dos objetivos rojo y azul a distintas distancias.	Satisfactorio.
Detección de objetivos rojo y verde	Se comprueba la detección de dos objetivos rojo y verde a distintas distancias.	Satisfactorio.
Detección de objetivos azul y verde	Se comprueba la detección de dos objetivos azul y verde a distintas distancias.	Satisfactorio.
Detección de tres objetivos	Se comprueba la detección de tres objetivos de distinto color a distintas distancias.	Satisfactorio.
Detección de seis objetivos	Se comprueba la detección de seis objetivos de distinto color (dos azules, dos rojos, dos verdes) a distintas distancias.	Satisfactorio.

Los resultados son muy similares a los obtenidos en simulación, aunque el efecto ojo de pez es menos acusado en la cámara real y no da problemas con la detección de distancia. Como puede observarse en la figura 4.4, el objetivo rojo de la izquierda apenas es deformado por la cámara y es detectado en la misma distancia que los que están a su lado. Además, esta figura muestra la correcta detección con 6 objetivos simultáneos. Se aprecian ciertas fluctuaciones en la detección, provocadas por el ruido, pero no son importantes. La detección de distancia funciona correctamente, igual que la detección de objetivos. Esto se muestra en la figura 4.5.

Conclusiones

El nodo funciona perfectamente y los problemas con el FOV que se preveían durante las pruebas en simulación han resultado ser inofensivos. Otro potencial problema era el ruido, pero este ha sido contenido con éxito usando técnicas de visión artificial para filtrar la

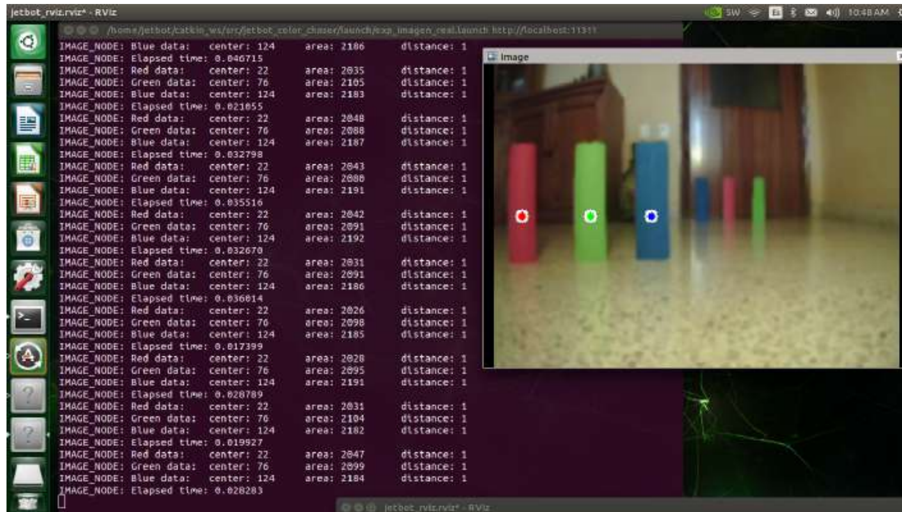


Figura 4.4 Detección de múltiples objetivos simultáneos.

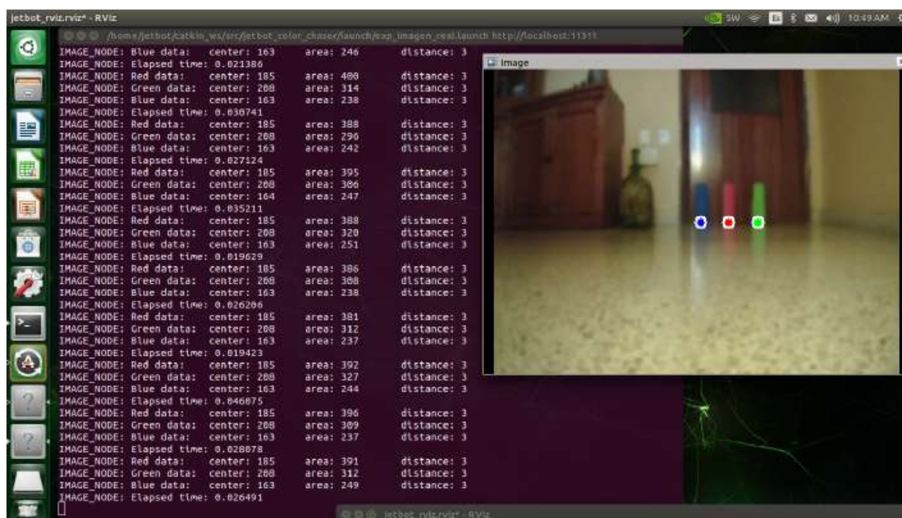


Figura 4.5 El nodo detecta sólo los objetivos más cercanos.

imagen.

4.3.2 Prueba de nodo de control en robot real.

Descripción del experimento

Experimento en el que se comprueba el correcto funcionamiento del nodo de control en un entorno real.

El experimento se realizará en una habitación vacía, moviendo al robot por el entorno por medio del nodo de control. El código se ejecutará usando el comando “rosrun´´ para ejecutar únicamente el nodo de control.

El objetivo del experimento es comprobar que el nodo de control puede:

- Recibir correctamente ordenes publicadas en el topic *"/jetbot/vel_cmd"*.
- Traducir dichas ordenes a velocidades para la rueda derecha y la rueda izquierda.
- Aplicar las velocidades generadas en el robot, haciendo que se mueva.

El experimento consiste en:

1. Iniciar el sistema de ROS, comprobando que toda la información se muestra de forma correcta.

2. Utilizar el comando de ROS “rostopic pub´´ para publicar ordenes en el topic *"/jetbot/vel_cmd"*. Por medio de esas ordenes, el robot deberá realizar los siguientes movimientos:

  - Movimiento recto hacia delante.
  - Movimiento recto hacia atrás.
  - Giro hacia la derecha sin moverse del sitio.
  - Giro hacia la izquierda sin moverse del sitio.
  - Movimiento hacia la derecha en círculos.
  - Movimiento hacia la izquierda en círculos.

Resultados

En la tabla 4.7 se muestran los resultados del experimento:

Tabla 4.7 Resultados del experimento del nodo de control.

Prueba	Descripción	Resultado
Inicio e interfaz	Se inicia el programa, comprobando que todos los nodos funcionan de forma correcta y la información es visible.	Satisfactorio.
Movimiento	Se comprueba el control en velocidad realizando diversos movimientos.	Parcialmente satisfactorio.

El nodo arranca y parece interpretar de forma correcta las ordenes recibidas, tal y como se puede leer en la terminal donde se imprimen todas las ordenes que se envían al robot. Las trayectorias listadas se han realizado, pero el control resulta demasiado abrupto y provoca movimientos bruscos que hacen a las ruedas deslizarse, provocan desviaciones y dificultan en general obtener el movimiento deseado.

Conclusiones

El control usando topics no es preciso, por lo que se realizarán pruebas con el programa de teleoperación a fin de concluir si es posible o no el control es posible. Sin embargo, el nodo cumple su objetivo y el problema parece hallarse en las características del hardware y no en el software.

4.3.3 Prueba de teleoperación.

Descripción del experimento

En este experimento se probará la capacidad del robot para ser operado de forma remota, utilizando para ello el software de teleoperación desarrollado.

Para realizar este experimento el robot se colocará en el mismo entorno y suelo en el que se realizan el resto de experimentos. El robot será teleoperado desde un ordenador portátil con Ubuntu 20.04.

Los objetivos del experimento son:

- Comprobar la conexión remota al robot.
- Comprobar la calidad del streaming de imagen.
- Probar la respuesta del robot al ser controlado con un controlador de videojuegos.

El experimento consiste en:

1. Conectarse de forma remota al robot.
2. Iniciar el software de teleoperación. Para ello se utiliza ROS y el comando “roslaunch´´, utilizando el fichero “teleoperation.launch´´ para lanzar el sistema de ROS que maneja la teleoperación.
3. Utilizar “GStreamer´´ para recibir el vídeo transmitido y observar la calidad y tiempo de respuesta del mismo. Esto se hará ejecutando en la máquina local el comando “gst-launch-1.0 udpsrc port=5000 ! queue ! application/x-rtp ! rtph264depay ! avdec_h264 ! videoconvert ! xvideosink´´, que mostrará el vídeo retransmitido en una ventana.
4. Realizar un circuito cerrado utilizando la teleoperación. El circuito cerrado debe tener giros a izquierda y a derecha y terminar en el mismo punto en el que empieza.

Resultados

En la tabla 4.8 se muestran los resultados del experimento:

Tabla 4.8 Resultados del experimento de teleoperación.

Prueba	Descripción	Resultado
Conexión remota	Conexión remota al robot usando SSH.	Satisfactorio.
Streaming de imagen	Se ejecuta “GStreamer´´ desde el ordenador local y se comprueba la calidad de la imagen	Satisfactorio.
Control con mando	Se realiza una vuelta en un circuito cerrado utilizando el mando.	Parcialmente satisfactorio.

La conexión por SSH funciona perfectamente y permite lanzar los programas sin problema. GStreamer permite enviar la imagen de la cámara sin retrasos una vez se ejecuta la pipeline en ambas máquinas. El circuito se ha podido recorrer de forma controlada, aunque el control sigue resultando abrupto y las ruedas deslizan demasiado.

Conclusiones

Se ha probado que la teleoperación es posible, demostrando que el nodo de control funciona correctamente. Sin embargo, el control sigue siendo difícil.

4.3.4 Prueba de sistema completo en robot real.

Descripción del experimento

Experimento en el que se comprueba el correcto funcionamiento del sistema completo en un entorno real.

El experimento se realizará en una habitación vacía, haciendo que el robot persiga los objetos del color que se le indiquen. Para ejecutar el experimento, se usará un archivo *launch.launch* que ejecuta todos los nodos del programa.

El objetivo del experimento es comprobar que el nodo central puede:

- Iniciar el movimiento por medio de la llamada a un servicio de ROS.
- Llegar hasta el objeto rojo sin chocar contra otro objeto.
- Perseguir al objeto rojo cuando es movido en el simulador.
- Cambiar el color del objeto a perseguir durante la ejecución usando un servicio de ROS.
- Cambiar la velocidad máxima durante la ejecución usando un servicio de ROS.

El experimento consiste en:

1. Iniciar el sistema de ROS, comprobando que toda la información se muestra de forma correcta.

2. Llamar al servicio de inicio para que el movimiento del robot comience.

3. Permitir que el robot persiga un objeto de color rojo, esquivando al resto.

4. Una vez el robot halla llegado al objeto de color rojo, coger el objeto y desplazarlo manualmente, de modo que el robot lo persiga activamente durante unos segundos.

5. Acercar el objeto rojo hacia el robot. Observar si el robot intenta alejarse para evitar una colisión.

6. Llamar al servicio de configuración para que el robot pase a perseguir a un objeto de color azul. Esperar hasta que el robot haya encontrado el objeto y este observándolo.

7. Llamar al servicio de configuración para reducir la velocidad máxima del robot.

8. Coger el objeto azul y desplazarlo manualmente, de modo que el robot lo persiga activamente durante unos segundos.

Este experimento se repetirá tres veces: primero empezando con rojo y pasando a azul, luego empezando en azul y pasando a verde y por último empezando en verde y pasando a rojo. En cada una de estas iteraciones se probarán distintas distribuciones de objetos.

Resultados

En la tabla 4.9 se muestran los resultados del experimento:

Tabla 4.9 Resultados del experimento del sistema completo en simulación.

Prueba	Descripción	Resultado
Inicio e interfaz	Se inicia el programa, comprobando que todos los nodos funcionan de forma correcta y la información es visible.	Satisfactorio.
Servicio de inicio	Se comprueba que el movimiento del robot empieza al llamar al servicio de inicio.	Satisfactorio.
Localización de objetivo rojo	Se comprueba que el robot localiza el objeto rojo y lo alcanza.	No satisfactorio.
Persecución de objetivo rojo	Se comprueba que el robot persigue al objeto rojo en caso de que este sea movido.	No satisfactorio.
Evasión de colisiones con objetivo rojo	Se comprueba que el robot se aleja del objetivo en caso de estar demasiado cerca.	Satisfactorio.
Cambio de color de objetivo	Se cambia el color objetivo al azul usando el servicio de configuración.	Satisfactorio.
Cambio de velocidad máxima	Se decrementa la velocidad máxima usando el servicio de configuración	Satisfactorio.
Persecución de objetivo azul	Se comprueba que el robot persigue al objeto azul en caso de que este sea movido.	No satisfactorio.

Cada una de las partes de la aplicación parece funcionar correctamente. Todos los nodos se inician como deben, la información se muestra correctamente en la pantalla y podemos observar en el terminal que el mapa se crea correctamente. El nodo de imagen emite la información de la detección de los objetos y el nodo de control permite controlar el robot. Los servicios funcionan y permiten controlar la ejecución de la aplicación y alterar sus parámetros.

Sin embargo, el robot es incapaz de perseguir al objetivo. Aunque el algoritmo de planificación funciona bien en situaciones estáticas, el algoritmo parece dejar de funcionar cuando hay movimiento: la detección de objetivos deja de funcionar correctamente y el control se vuelve errático con un comportamiento que da la impresión de sobreoscilar. El robot siempre gira un poco más de lo que debería, teniendo que corregir su rumbo girando en sentido contrario y así una y otra vez. Esto ocurre incluso utilizando la mínima velocidad posible. El control lineal, sin embargo, parece funcionar bien. El robot se acerca al objetivo si está lejos y se aleja si está cerca.

Conclusiones

El resultado, aunque no es satisfactorio, si que resulta esperado. Desde las primeras pruebas se ha observado un movimiento demasiado rápido y brusco en el robot real y eso ha pasado factura durante la ejecución del sistema completo. Las sobreoscilaciones en el control afectan negativamente a todo el sistema, haciendo oscilar la cámara y haciendo imposible seguir las ordenes de la planificación de trayectoria, pero resultan inevitables cuando la

velocidad de giro mínima es tan alta. La solución a este problema pasaría por cambiar los motores, pero por falta de tiempo y recursos este proyecto no ahondará en esta alternativa.

Lo que sí resulta satisfactorio es el comportamiento del sistema de ROS, como todos los nodos trabajan juntos y se comunican a tiempo real. El uso de topics y servicios queda bien ejemplificado con esta aplicación y los resultados en simulación lo muestran con claridad.

Capítulo 5: Conclusiones y futuras líneas

5.1 Conclusiones generales

En conclusión, el presente proyecto ha sido un ejemplo revelador de cómo incluso los esfuerzos más rigurosos y planificados pueden encontrar obstáculos inesperados y resultados no deseados. A pesar de los mejores intentos, este proyecto ha experimentado dificultades significativas y no ha alcanzado todos los objetivos planteados. El salto de simulación a hardware real ha resultado ser más problemática de lo esperado y las limitaciones encontradas durante el proyecto, que se discuten en el próximo apartado, han resultado ser insalvables en el estado actual del proyecto.

Sin embargo, no todo ha sido un fracaso. Si bien la aplicación desarrollada no se pudo implementar del todo en el robot real, los subsistemas y aplicaciones intermedias han funcionado correctamente y los experimentos en simulación han sido un éxito. En este proyecto se ha abordado múltiples retos desde el montaje del robot hasta la aplicación final, pasando por pruebas a varios niveles y se han superado en su mayoría, dejando un registro del proceso completo que sin duda alberga un gran valor y puede servir de ayuda a quienes pretendan desarrollar un sistema similar.

En última instancia, aunque este proyecto no haya cumplido plenamente con sus objetivos, en este último punto se plantean líneas futuras de trabajo en las que, con más tiempo y recursos, se podrían superar las limitaciones encontradas y lograr un resultado plenamente satisfactorio.

5.2 Limitaciones encontradas durante el proyecto

Durante el desarrollo de un proyecto de estas características, resulta imposible no encontrar varios baches en el camino. Como el viaje suele enseñar mucho más que el destino, en

este apartado se describen algunos de los problemas más limitantes encontrados y como se solucionaron, si es que se pudo.

5.2.1 Hardware

Trabajar con hardware es complicado, incluso en el caso de un kit preparado como es el caso del Jetbot. Las piezas fallan o se degradan, es fácil realizar conexiones erróneas y no es poco común que el software no sea del todo compatible con los distintos componentes. Concretamente, han habido dos subsistemas que han dado muchos problemas en el apartado de hardware:

- El primero es la actuación: el subsistema de motores y ruedas ha resultado ser especialmente problemático. Aparte de los ejes sueltos, problema que se expone en el punto 3.1.1, ha resultado especialmente notoria la falta de par de los motores de corriente continua usados en el robot. Se trata de motores pequeños de muy poco consumo pero que carecen de la fuerza para mover al robot en movimientos lentos y controlados. Esto ha limitado enormemente el control del robot real, que depende de movimientos delicados para sortear obstáculos y encontrar el objetivo. La solución adoptada en este proyecto ha sido la de reemplazar los motores por otros nuevos y las ruedas por otras más gruesas y menos susceptibles a deslizar en el suelo, lo que mejoró levemente el comportamiento general. Una mejor solución sería la de cambiar los motores por otros más adecuados, tal y como se menciona en las líneas futuras de trabajo en el siguiente apartado.

- Otra limitación importante ha ocurrido en la cámara. La visión es la clave de este proyecto y es por eso por lo que depende de una cámara capaz de captar correctamente los objetivos, con una calidad aceptable, un color nítido y un campo de visión amplio. Encontrar la cámara que pudiera proveer de estas cualidades ha requerido investigación y muchas pruebas, principalmente debido a dos restricciones: aumentar demasiado el peso iba a empeorar aún más el desempeño de los motores y aumentar demasiado el campo de visión iba a arruinar el detalle de la detección de objetivos en los bordes. Para encontrar el valor perfecto de campo de visión, se realizaron pruebas en simulación variando el valor de FOV de la cámara simulada y comprobando que la distorsión radial no afectara demasiado a los algoritmos de visión.

  Otro gran problema, relacionado con la cámara, ha sido la iluminación del campo de pruebas donde se realizaron los experimentos: la segmentación por color esta fuertemente influenciada por las condiciones de iluminación, que pueden cambiar a lo largo del día, con la posición de las luces de la sala y con la posición de la cámara respecto a estas. Para paliar la variación de iluminación se intento utilizar varias lamparas en la sala donde se realizaron los experimentos, aunque lo ideal sería una fuente de luz natural o una fuente de luz blanca y controlada. A falta de un entorno de pruebas adecuado, la solución adoptada frente a este problema ha sido la de realizar una calibración de los algoritmos de visión justo antes de cada experimento. En el siguiente apartado, se presenta como linea futura de trabajo el cambio de zona de pruebas.

- Otros problemas, típicos de trabajar con hardware, han retrasado notoriamente el desarrollo del proyecto, aunque no han afectado notoriamente al resultado final: conexiones erróneas o dañadas, piezas desgastadas que han necesitado recambios o componentes que no terminaban de encajar en el robot. La mayoría de estos problemas no se tuvieron en cuenta al principio y ha sido una gran lección de aprendizaje el como tratar con estos. Las conexiones dañadas fueron soldadas de nuevo utilizando estaño y aseguradas con tubos termo-retráctiles, utilizando las medidas de seguridad adecuadas. Las piezas desgastadas han requerido investigación para encontrar las alternativas adecuadas a precios razonables, buscando cumplir los requerimientos del proyecto. Y, finalmente, aquellos reemplazos que pese a la investigación previa a la compra no terminaban de encajar correctamente fueron colocados utilizando pequeñas piezas diseñadas en ordenador e impresas en 3D, como la sujeción de la cámara mostrada en el capitulo 3.

5.2.2 Software

Respecto al software, los principales limitantes encontrados han sido el rendimiento: procesar vídeo en tiempo real no es una tarea simple, menos aún sin conocimientos sobre streaming de vídeo. Sin la capacidad de modificar como se obtiene la imagen, resulta inevitable encontrar retrasos en la entrada de vídeo o frames que se pierden, lo que arruina por completo los algoritmos de detección y control. Esto ha sido finalmente corregido utilizando librerías de más bajo nivel (GStreamer) para controlar como la imagen era obtenida, bajando la calidad de la imagen aprovechando que nuestra aplicación no requería gran nivel de detalle y finalmente ejerciendo el control con una frecuencia de 10Hz.

Además, como se comentó anteriormente, las aplicaciones de visión artificial sufren en gran medida por una inevitable necesidad de controlar las condiciones lumínicas. Una iluminación ligeramente distinta puede provocar que tenga que rehacerse todo el proceso de calibración de los algoritmos de detección de objetivos. Desde el punto de vista del software, esto podría solucionarse de dos formas distintas: la primera, más simple, es la de tratar de compensar las diferencias de iluminación preprocesando la imagen capturada y manipulando el histograma; la otra solución, más drástica, pasa por volver a idear por completo el sistema de percepción utilizando una solución menos susceptible a la iluminación, como puede ser el uso de inteligencia artificial.

Otro limitante ha sido el uso de Python para la programación de los distintos nodos. Python, aunque sencillo y flexible, presenta importantes deficiencias en velocidad de ejecución que afectan muy negativamente al objetivo del control en tiempo real del robot. Utilizar lenguajes compilados como C++ es una buena solución a esto.

5.3 Futuras líneas de trabajo

Con las limitaciones anteriores en mente, en este apartado se plantean líneas de trabajo para mejorar el proyecto:

- *Cambio de motores a otros más potentes o a unos motores paso a paso.* Esto mejoraría enormemente la actuación y permitiría un control mucho más fino. Los motores que el Jetbot tiene instalados son demasiado bruscos y carecen del par necesario para mover el robot con la suavidad necesaria para posibilitar el control. Otra solución alternativa sería instalar una reductora aún mayor a los motores.

- *Reescritura de los nodos de ROS en C++.* C++ es un lenguaje compilado que ofrece velocidades de ejecución mucho más rápidas que Python. Además, también es compatible con ROS.

- *Preparación de un mejor entorno de pruebas.* Parte de la dificultad a la hora de realizar experimentos ha sido el entorno de pruebas o, más bien, la falta de uno. Un lugar fijo con iluminación consistente, suelo liso y homogéneo y relativamente vacío resulta un requisito indispensable para obtener resultados eficientes y repetibles.

- *Cambio de CPU.* El hardware utilizado se eligió al ser la unidad que se utilizará en el testbed que este TFG va a ayudar a preparar, que experimentará con algoritmos basados en inteligencia artificial y por tanto requiere la potencia que aporta la *Jetson Nano*. Sin embargo, en el caso de querer reproducir los experimentos llevados a cabo en este proyecto y ejecutar el código desarrollado, podría ser razonable utilizar otro sistema basado en CPUs más económicas y ligeras, tales como Rapsberry PI o Arduino Mega.

- *Detección de objetivos utilizando ARUCO.* La detección podría ser mucho mas eficiente y extraer más información de los objetivos si en lugar de utilizarse una segmentación por color se leyeran códigos ARUCO.

- *Desarrollo de algoritmos de percepción basados en inteligencia artificial.* El hardware usado para computo (Jetson Nano) es más que capaz de ejecutar redes neuronales convolucionales que permitirían obtener más información de la imagen. Esto permitiría reutilizar este sistema para el seguimiento de otros objetos, robots o incluso personas y aliviaría la necesidad de una cámara especialmente buena.

Índice de Figuras

1.1	Vehículo autónomo usado para la supervisión del lago Ypacarai	3
1.2	Jetbot AI Kit de Waveshare	5
2.1	Ejemplo de robot móvil.	7
2.2	Representación gráfica del objeto del SLAM.	8
2.3	Experimentos en un entorno real del seguimiento de personas.	9
3.1	Jetbot AI Kit de Waveshare.	11
3.2	Chasis desmontado	13
3.3	Montaje de los motores en el chasis	13
3.4	Parte superior del chasis	13
3.5	Soporte para la cámara sujeto al chasis	14
3.6	Placa sujeta al chasis	14
3.7	Cerrado del chasis y posicionamiento de las ruedas en el eje	14
3.8	Conexionados de la cámara, el ventilador y entre placas	15
3.9	Jetson asegurada en la parte superior del robot	15
3.10	Desmontaje de la Jetson para acceder a la batería	16
3.11	Medida de tensión de baterías	16
3.12	Vista inferior del chasis abierto	17
3.13	Nuevos motores y ruedas montados	17
3.14	Jetbot con nuevas piezas instaladas	17
3.15	NVIDIA Jetson Nano Developer Kit	18
3.16	Ejemplo de la distorsión radial en la cámara original	19
3.17	Cámara USB papalook	19
3.18	Cámara USB ELP-USBFHD01M-L36	20
3.19	Adaptador para sujetar la cámara al Jetbot	21
3.20	Ejemplo de filtro de medianas	23
3.21	Ejemplo de operación de apertura	24
3.22	Funcionamiento del modo Polling	25
3.23	Funcionamiento del modo asíncrono	26
3.24	Funcionamiento del modo de eventos	27
3.25	Robot en CoppeliaSim.	30
3.26	Ajustes de la cámara usadas para la simulación.	30

3.27 Relación entre ”x” e ”y” con y sin ajuste. 32
3.28 Interfaz usada para calibrar “alpha“ y “beta“ 33
3.29 Calibración ofrecida por la librería "default" frente a la correcta 33
3.30 Calibración deseada para las nuevas librerías 34
3.31 Objetivos a detectar por el sistema 35
3.32 Comportamiento del nodo de imagen 36
3.33 Comportamiento del nodo de control 38
3.34 Comportamiento del nodo central 40
3.35 Sistema completo 41

4.1 Aplicación de calibración de imagen en acción 46
4.2 Entorno de simulación usado en los experimentos. 47
4.3 Imagen capturada por la cámara simulada 50
4.4 Detección de múltiples objetivos simultáneos 57
4.5 El nodo detecta sólo los objetivos más cercanos 57

Índice de Tablas

4.1 Resultados del experimento de calibración de motores 44
4.2 Resultados del experimento de calibración de cámara 46
4.3 Resultados del experimento del nodo de imagen en simulación 49
4.4 Resultados del experimento del nodo de control en simulación 51
4.5 Resultados del experimento del sistema completo en simulación 53
4.6 Resultados del experimento del nodo de imagen 56
4.7 Resultados del experimento del nodo de control 58
4.8 Resultados del experimento de teleoperación 60
4.9 Resultados del experimento del sistema completo en simulación 62

Bibliografía

[1]

[2]

[3]

[4] Gary Bradski and Adrian Kaehler, *Learning opencv: Computer vision with the opencv library*, " O'Reilly Media, Inc.", 2008.

[5] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, José Neira, Ian Reid, and John J. Leonard, *Past, present, and future of simultaneous localization and mapping: Toward the robust-perception age*, IEEE Transactions on Robotics **32** (2016), no. 6, 1309–1332.

[6] Arron Churchill, *Gamepad library*.

[7] Jorge Fraga, João Sousa, Gonçalo Cabrita, Paulo Coimbra, and Lino Marques, *Squirtle: An asv for inland water environmental monitoring*, pp. 33–39, Springer International Publishing, Cham, 2014.

[8] Meenakshi Gupta, Swagat Kumar, Laxmidhar Behera, and Venkatesh K. Subramanian, *A novel vision-based tracking algorithm for a human-following mobile robot*, IEEE Transactions on Systems, Man, and Cybernetics: Systems **47** (2017), no. 7, 1415–1427.

[9] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant, *Array programming with NumPy*, Nature **585** (2020), no. 7825, 357–362.

[10] Samuel Yanes Luis, Daniel Gutiérrez Reina, and Sergio L. Toral Marín, *A multiagent*

deep reinforcement learning approach for path planning in autonomous surface vehicles: The ypacaraí lake patrolling case, IEEE Access **9** (2021), 17084–17099.

[11] E. Rohmer, S. P. N. Singh, and M. Freese, *V-rep: A versatile and scalable robot simulation framework*, 2013 IEEE/RSJ International Conference on Intelligent Robots and Systems, 2013, pp. 1321–1326.

[12] Stanford Artificial Intelligence Laboratory et al., *Robotic operating system*.

[13] Guido Van Rossum and Fred L. Drake, *Python 3 reference manual*, CreateSpace, Scotts Valley, CA, 2009.

[14] Linlin Xia, Jiashuo Cui, Ran Shen, Xun Xu, Yiping Gao, and Xinying Li, *A survey of image semantics-based visual simultaneous localization and mapping: Application-oriented solutions to autonomous navigation of mobile robots*, International Journal of Advanced Robotic Systems **17** (2020), no. 3, 1729881420919185.