

Trabajo Fin de Grado

Ingeniería Electrónica, Robótica y Mecatrónica

Simulación y puesta en marcha de un sistema de Pick and Place basado en visión por computador del robot Braccio Tinkerkit de Arduino, controlado mediante el framework ROS 2 Humble

Autor: Iván Luque Valverde

Tutor: Federico Cuesta Rojo

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2025



Trabajo Fin de Grado
Ingeniería Electrónica, Robótica y Mecatrónica

**Simulación y puesta en marcha de un sistema
de Pick and Place basado en visión por compu-
tador del robot Braccio Tinkerkit de Arduino,
controlado mediante el framework ROS 2
Humble**

Autor:
Iván Luque Valverde

Tutor:
Federico Cuesta Rojo
Profesor Titular

Dpto. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2025

Trabajo Fin de Grado: Simulación y puesta en marcha de un sistema de Pick and Place basado en visión por computador del robot Braccio Tinkerkit de Arduino, controlado mediante el framework ROS 2 Humble

Autor: Iván Luque Valverde

Tutor: Federico Cuesta Rojo

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:

Agradecimientos

En primer lugar, quiero expresar mi más sincero agradecimiento a mi familia, mis padres y hermano que no han dudado nunca de mí y me han apoyado incondicionalmente en cada paso de mi vida académica y personal. Sin ese empujón para luchar por aquello que deseaba desde pequeño, no estaría aquí hoy.

A continuación, me gustaría agradecer a mi tutor, Federico Cuesta Rojo, por abrirme las puertas a embarcarme en este proyecto increíble que enlaza la robótica manipuladora con la percepción y control. Gracias por su orientación, experiencia y apoyo a lo largo de este proyecto.

Una ingeniería es un camino largo y muy complicado. Cómo no agradecer a todos aquellos compañeros, amigos, que me llevó de este viaje y que han hecho de la universidad un hogar, un lugar mucho más ameno donde las risas y los buenos momentos han sido la tónica dominante incluso en las largas sesiones de estudio. Sin duda, me llevo un pedacito de cada uno de vosotros.

Finalmente, agradecer a todas aquellas personas que, de una forma u otra, han aportado su granito de arena para que este proyecto haya sido posible. Desde aquellos maestros que me enseñaron las bases de la tecnología, matemáticas y física en el instituto, hasta la propia universidad de Sevilla por darme la oportunidad de formarme y crecer como persona y ahora, como ingeniero.

Iván Luque Valverde

Sevilla, 2025

Resumen

Este Trabajo de Fin de Grado diseña, simula y valida un sistema de pick-and-place de bajo coste con el manipulador educativo Braccio Tinkerkit, controlado por Arduino y coordinado en ROS 2 Humble. Se parte de un repositorio modular al que se incorporan paquetes propios para la teleoperación con gamepad, percepción y calibración y transferencia al entorno real. En simulación, el entorno integra Gazebo, MoveIt2 y RViz2, con una cámara cenital y objetos cúbicos. La percepción aplica segmentación por color, extracción de centroides y una calibración por homografía para convertir píxeles a coordenadas del mundo. La planificación se basa en una cinemática inversa específica del Braccio (con manejo de simetrías y alturas) y en la ejecución de trayectorias con ros2_control; el agarre se simula mediante el plugin gazebo_link_attacher.

La fase sim-to-real replica el escenario con una cámara de un teléfono móvil, enlazada al ordenador mediante DroidCam; y marcadores adicionales, reajustando umbrales de visión. Los resultados muestran una ejecución impecable en simulación y un desempeño aceptable en el robot real, limitado por la rigidez del hardware, la apertura de la pinza y la sensibilidad a la iluminación, entre otros factores. El proyecto entrega un repositorio abierto, documentado y extensible para docencia e investigación, con utilidades de prueba y configuración reproducible, dedicado a la comunidad de ROS 2 y robótica educativa.

Abstract

This Final Degree Project designs, simulates, and validates a low-cost pick-and-place system using the educational Braccio Tinkerkit manipulator, controlled by Arduino and coordinated in ROS 2 Humble. It builds on a modular repository augmented with custom packages for gamepad teleoperation, perception and calibration, and transfer to the real setup. In simulation, the stack integrates Gazebo, MoveIt2, and RViz2, with an overhead camera and cubic objects. The perception pipeline applies color segmentation, centroid extraction, and a homography based calibration to convert pixel data into world coordinates. Motion planning is based on a custom inverse kinematics solution for the Braccio arm (handling symmetries and height adjustments) and trajectory execution via ros2_control; grasping is simulated with the gazebo_link_attacher plugin.

The sim-to-real phase replicates the setup using a mobile phone camera linked to the computer via Droid-Cam, requiring additional markers and recalibration of vision thresholds. Results show flawless execution in simulation and acceptable performance on the real robot, limited by hardware rigidity, gripper aperture, and lighting sensitivity, among other factors. The project provides an open, documented, and extensible repository for education and research, with test utilities and reproducible configuration, dedicated to the ROS 2 and educational robotics community.

Índice

<i>Agradecimientos</i>	I
<i>Resumen</i>	III
<i>Abstract</i>	V
<i>Índice</i>	VII
1. Introducción	1
1.1. Motivación del proyecto	1
1.2. Objetivos del Trabajo	2
2. Estado del arte	3
2.1. Introducción a la robótica	3
2.2. Tipos de robots	3
2.2.1. Robots industriales	4
2.2.2. Robots de servicios	5
2.2.3. Robots médicos	5
2.3. Estudio de mercado	5
2.4. Braccio Tinkerkit	7
3. Plataformas de desarrollo y simulación	11
3.1. Plataformas de desarrollo	11
3.1.1. Matlab	11
3.1.2. ROS	11
3.1.3. ROS 2	11
3.1.4. Comparativa de las plataformas	12
3.2. Simuladores, planificadores y visores en ROS 2	12
3.2.1. Simuladores físicos	13
3.2.2. Planificación	13
3.2.3. Visualización y depuración	13
3.2.4. Elección de herramientas	13
4. Diseño del sistema	15
4.1. Repositorio ROS2 Braccio	15
4.2. Extensiones y mejoras implementadas	16
4.3. Entorno de simulación	16
4.3.1. Robot manipulador	16
4.3.2. Spawner de Cámara y Cubos	17
5. Control mediante PS4 controller	19
5.1. Arquitectura y filosofía de control	19
5.2. Mapeo de botones y joysticks	20
5.2.1. Joysticks	20
5.2.2. Botones	20
5.3. Flujo de datos y control	21

6.	Percepción y localización de objetivos	23
6.1.	Sensor de la cámara	24
6.2.	Detección de objetos	24
6.3.	Matriz de Homografía	25
7.	Planificación de agarre y manipulación	27
7.1.	Cinemática inversa	27
7.1.1.	Fundamentos teóricos	27
7.1.2.	Configuración simétrica	28
7.1.3.	Cálculo de las posiciones de aproximación y agarre	29
7.1.4.	Test cinemática inversa	31
7.2.	Repositorio attach/detach	32
7.3.	Flujo de acción	33
8.	Sim to real y validación	37
8.1.	Montaje robot físico	37
8.2.	Espacio de trabajo	38
8.3.	Calibraciones y ajustes	39
8.4.	Validación del sistema	40
8.4.1.	Visión	40
8.4.2.	Localización	40
8.4.3.	Trayectoria	41
8.4.4.	Manipulación	41
9.	Conclusión	43
	<i>Índice de Figuras</i>	45
	<i>Índice de Tablas</i>	49
	<i>Bibliografía</i>	51

1 Introducción

El presente documento titulado «*Simulación de un sistema de Pick and Place con un robot Braccio Tinkerkit de Arduino bajo ROS 2*» es el trabajo presentado para superar el Trabajo de Fin de Grado del Grado de Ingeniería Electrónica, Robótica y Mecatrónica.

El consiguiente aborda la simulación y validación de un sistema de recolección, clasificación y colocación de elementos, basado en el kit educativo Braccio Tinkerkit, controlado por una placa Arduino UNO y coordinado desde ROS 2 Humble.

La relevancia recae en el uso de los robots manipuladores en tareas de automatización y docencia, justificando el uso de plataformas de bajo coste para experimentar técnicas de percepción, planificación y control previas a la transferencia al robot físico. Este incremento de la robótica en la educación ha demostrado un aumento en el interés de los estudiantes por la ingeniería y la tecnología, así como una mejora en su comprensión de conceptos complejos, la resolución de problemas, el trabajo en equipo y la creatividad [1].

1.1 Motivación del proyecto

Este proyecto nace como una combinación de motivos personales, formativos y profesionales.

Desde un punto de vista personal, siempre he tenido un gran interés por la robótica y la automatización, fascinado por cómo las máquinas pueden interactuar con el mundo físico y realizar tareas complejas. Desde bien pequeño recuerdo el entusiasmo al desenvolver un regalo y descubrir un kit de construcción como el mostrado en la Figura 1.1, otorgándome horas innumerables de diversión y aprendizaje mientras ensamblaba y la enorme satisfacción al comprobar que, tras todo ese esfuerzo, había construido un robot que funcionaba. Finalmente, esos pequeños kits de construcciones, laboratorios o electrónica, fueron construyendo mi pasión por la robótica y la tecnología.

Desde un punto de vista formativo, este proyecto representa una oportunidad para aplicar y consolidar los conocimientos adquiridos a lo largo de la carrera, especialmente en áreas como la programación, la robótica y la percepción. Trabajar durante la asignatura *Laboratorio de Robótica* con un robot ABB IRB 120 despertó mi entusiasmo por este tipo de robots manipuladores, enlazado con los conocimientos adquiridos durante las asignaturas *Sistemas de Percepción* y *Ampliación de Robótica* constituyeron la oportunidad ideal para unificar estos conocimientos bajo el mismo proyecto.



Figura 1.1 Juguete para niños, kit de construcción de un vehículo todo-terreno Meccano

Finalmente, desde un punto de vista profesional, la experiencia adquirida en este proyecto será un valioso activo en mi futura carrera. La entrada en un ecosistema como ROS 2, apenas explorado durante la carrera, representa una oportunidad para adquirir habilidades demandadas en el mercado laboral tales como la búsqueda e implementación de repositorios o el manejo de un sistema de nodos y publicaciones. La robótica es un campo en constante evolución y crecimiento, y contar con experiencia práctica en el desarrollo de sistemas robóticos me posicionará favorablemente en el mercado laboral.

1.2 Objetivos del Trabajo

Este trabajo tiene como objetivo principal el diseño y desarrollo de un sistema de simulación para un robot manipulador, utilizando el kit Braccio Tinkerkit y ROS 2. Se busca crear un entorno virtual que permita la experimentación y validación de algoritmos de control y percepción, facilitando la transferencia de estos al robot físico.

De forma complementaria, se pretende establecer un flujo de trabajo que integre la simulación con el robot físico, permitiendo la validación de los algoritmos en un entorno real. Esto incluye la creación de un repositorio completamente modular donde la adición de nuevos elementos se realice de forma sencilla e intuitiva, asegurando la escalabilidad y flexibilidad de ambos sistemas. Para ello, en primer lugar, se ha seleccionado un repositorio existente en GitHub como base inicial y funcional para el desarrollo del sistema [2]. A continuación, se ha modelado hacia el objetivo deseado, adaptando y ampliando las funcionalidades del repositorio original con nuevas características referenciadas en otros repositorios. Finalmente, se ha implementado y probado el sistema tanto en simulación como en el robot físico, evaluando su rendimiento y realizando ajustes según sea necesario.

2 Estado del arte

2.1 Introducción a la robótica

La robótica se define como la técnica que aplica la informática al diseño y empleo de aparatos que, en sustitución de personas, realizan operaciones o trabajos, por lo general en instalaciones industriales (RAE).

El término “robot” fue acuñado por el escritor checo Karel Čapek en su obra de teatro “Rossum’s Universal Robots” en 1921 en el Teatro Nacional de Praga, en la cual se creaban humanos sintéticos para aligerar la carga de trabajo de los humanos. Cabe destacar que este término fue realmente ideado por el hermano del autor, el cual se basó en la palabra checa *robota*, que significa trabajo, en general, de la servidumbre [3].

De forma similar al descrito en la obra teatral, en un principio, los robots fueron concebidos como herramientas para sustituir a los humanos en tareas específicas debido a peligrosidad, precisión o repetitividad. Este hecho, unido con el auge de otros campos como la electrónica y la informática, ha permitido el desarrollo de robots cada vez más sofisticados y capaces de realizar tareas complejas, siendo éstos prácticamente indispensables en la automatización industrial moderna.

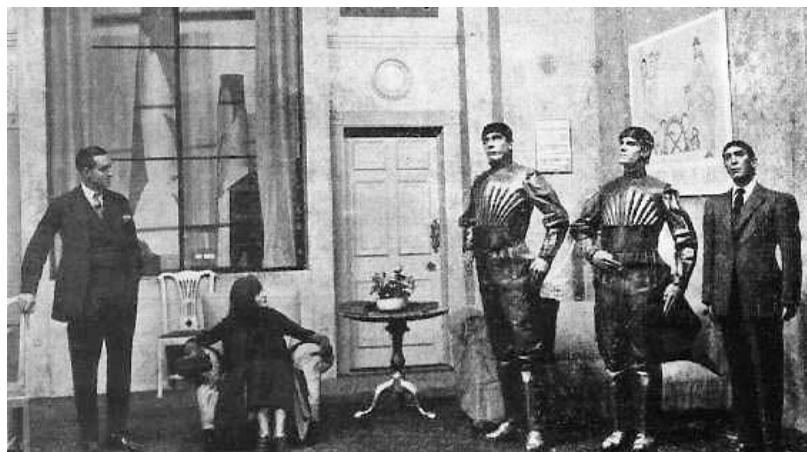


Figura 2.1 Representación original de la obra teatral de Karel Čapek, donde se observan un hombre junto a una mujer y tres robots.

2.2 Tipos de robots

Los robots son máquinas programables capaces de realizar tareas de forma autónoma o semiautónoma. Según la norma ISO (International Organization for Standardization) 8373 [4], se clasifican en tres grandes grupos en función de su uso. Los robots industriales son robots necesarios en tareas de automatización industrial, los de servicio realizan tareas útiles para las personas o los equipos; y los médicos están destinados a ser utilizados como equipo electromédico o sistema electromédico.

2.2.1 Robots industriales

La ISO define un robot industrial como un manipulador multipropósito reprogramable, controlado automáticamente, programable en tres o más ejes, que puede estar fijo en un lugar o fijado a una plataforma móvil para su uso en aplicaciones de automatización en un entorno industrial.

En base a ello, la IFR (International Federation of Robotics) clasifica estos según su estructura mecánica.

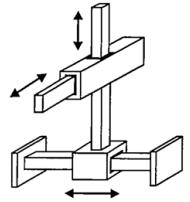
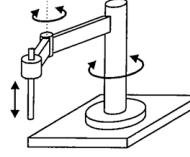
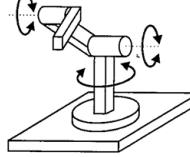
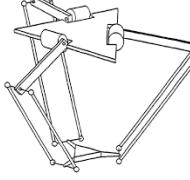
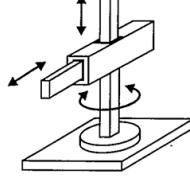
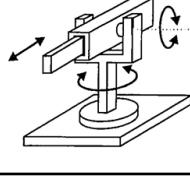
Nombre	Estructura mecánica	Imagen
Robot cartesiano	Manipulador que tiene tres articulaciones prismáticas, cuyos ejes forman un sistema de coordenadas cartesianas.	
Robot SCARA	Manipulador que tiene dos articulaciones rotatorias paralelas para proporcionar flexibilidad en un plano seleccionado.	
Robot articulado	Manipulador con tres o más articulaciones rotatorias.	
Robot paralelo / Delta	Manipulador cuyos brazos tienen enlaces que forman una estructura de bucle cerrado.	
Robot cilíndrico	Manipulador con al menos una articulación rotatoria y una prismática, cuyos ejes forman un sistema de coordenadas cilíndrico.	
Robot polar / esférico	Manipulador con dos articulaciones rotatorias y una articulación prismática, cuyos ejes forman un sistema de coordenadas polares.	

Tabla 2.1 Clasificación de los robots industriales en función de su estructura mecánica [5].

2.2.2 Robots de servicios

La IFR define un robot de servicio como un mecanismo accionado programable en dos o más ejes, que se mueve dentro de su entorno, para realizar tareas útiles para humanos o equipos, excluyendo aplicaciones de automatización industrial. Según su definición en la norma ISO, requieren de cierto grado de autonomía, yendo desde una autonomía parcial hasta una total autonomía, es decir, con cierto grado de interacción con un operador [6].

Esta institución distingue dos categorías de robots de servicio:

- Robots de servicio para el consumidor: destinados a ser utilizados por particulares en entornos domésticos. No requieren de formación específica para su uso. Algunos ejemplos son los robots de limpieza domésticos, las sillas de ruedas automatizadas o los robots de interacción social.
- Robots de servicio profesional: diseñados para realizar tareas específicas en entornos industriales o comerciales. Requieren de un operador con formación profesional. Algunos ejemplos son los robots de limpieza para espacios públicos, los robots de reparto o los robots de extinción de incendios. En la Figura 2.2 se puede observar un mayor catálogo del uso de éstos.

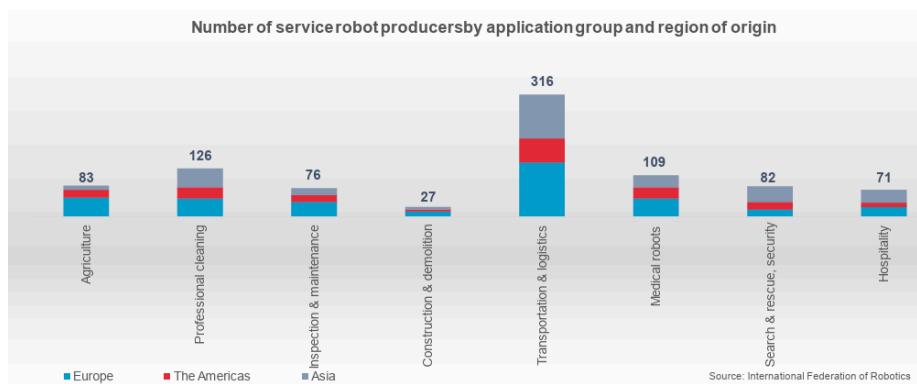


Figura 2.2 Representación gráfica del número de productores de robots de servicio por grupo de aplicación y origen en 2024 [7].

2.2.3 Robots médicos

Los robots médicos constituyen ahora una tercera área de aplicación, catalogándose anteriormente como una categoría especializada de robots de servicio. Sin embargo, tal como se documenta en este reportaje constituido por varias organizaciones sanitarias de Polonia [8], su definición aún se muestra un poco confusa si se considera la ofrecida por la ISO.

Pese a eso, basándose en la definición oficial, los robots médicos están diseñados para asistir en la atención médica y quirúrgica, pudiendo realizar tareas como la cirugía asistida, la rehabilitación de pacientes y la entrega de suministros médicos. Su uso en entornos clínicos requiere un alto grado de precisión y fiabilidad, así como la capacidad de interactuar de manera segura con pacientes y profesionales de la salud.

2.3 Estudio de mercado

En la actualidad, la presencia de robots industriales en el mercado está en constante crecimiento, impulsada por la demanda de automatización en diversos sectores. Según la Figura 2.3, más de 4 millones de robots industriales se encuentran operando en todo el mundo. Este crecimiento se debe a la adopción de tecnologías avanzadas, como la inteligencia artificial y la robótica colaborativa, que permiten a las empresas mejorar su eficiencia y competitividad.

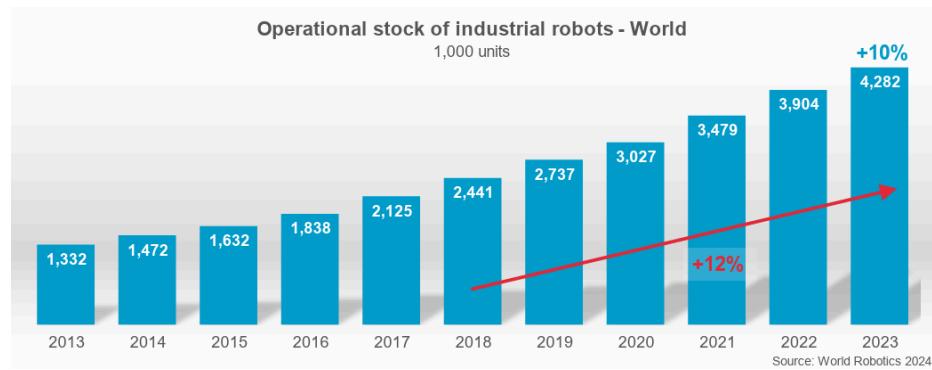


Figura 2.3 Representación gráfica del crecimiento en la cantidad de robots industriales operando en el mercado durante los últimos 10 años según World Robotics en 2024 [7].

Este incremento en la aplicación robótica se traduce en una oportunidad de mercado para aprender y desarrollar nuevas soluciones en el ámbito de la automatización y la robótica. En concreto, la versatilidad y funcionalidad que ofrecen los robots articulares, comúnmente llamados brazos robóticos ha impulsado a miles de estudiantes y profesionales del sector a contribuir en el desarrollo. Gracias a ello, se ha dado lugar a la creación de nuevas plataformas donde los usuarios pueden colaborar y compartir sus experiencias, enriqueciendo aún más el aprendizaje y la innovación en este campo.

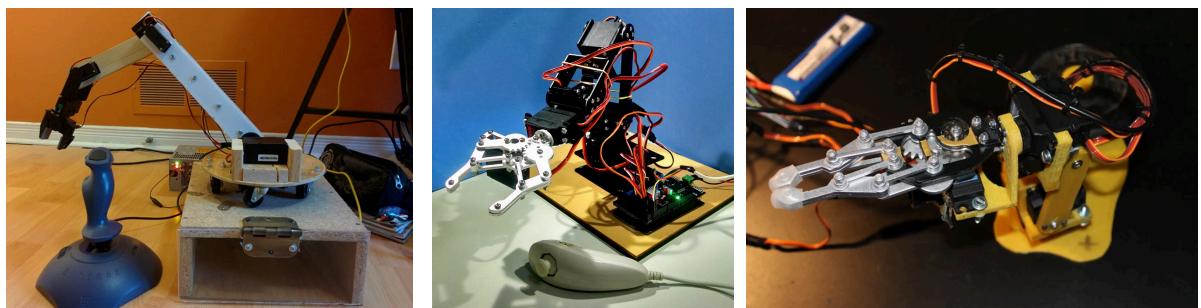


Figura 2.4 Proyectos compartidos por la comunidad de Autodesk Instructables, donde se explica mediante tutoriales y documentación cómo construir brazos robóticos [9].

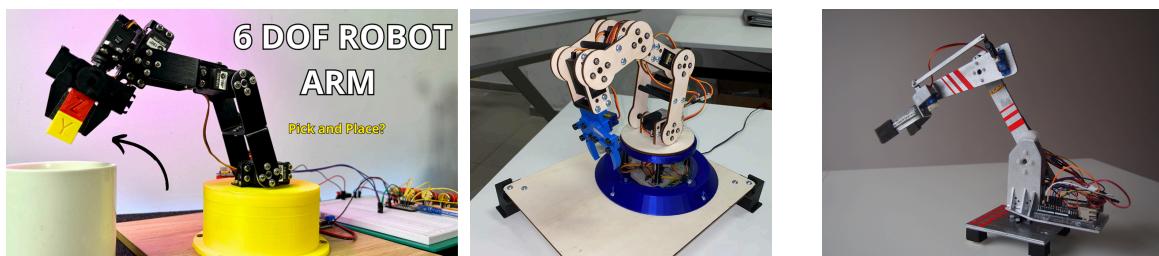


Figura 2.5 Proyectos compartidos por la comunidad de Arduino Project Hub, donde se explica mediante tutoriales y documentación la construcción y control de brazos robóticos mediante Arduino [10].

Con la proliferación de estas plataformas y la creciente demanda de soluciones robóticas, se ha generado un ecosistema vibrante y colaborativo que impulsa la innovación y el desarrollo en el campo de la robótica. Estas alternativas manuales también han servido como base para el desarrollo de kits educativos y plataformas de bajo coste, que permiten a estudiantes y entusiastas aprender sobre robótica y automatización de manera accesible y práctica, como el Braccio Tinkerkit de Arduino.

2.4 Braccio Tinkerkit

El Braccio Tinkerkit es un manipulador educativo de sobremesa diseñado para aprendizaje, prototipado y experimentación con control de robots manipuladores a bajo coste. Este kit de montaje ofrece una introducción versátil a la robótica, la mecánica y la electrónica, permitiendo a los usuarios ensamblar y programar el brazo para una variedad de tareas, como la manipulación de objetos, programación de trayectorias o control de articulaciones.

Destaca por su flexibilidad y enfoque educativo, constando de las siguientes características:

- Control por Arduino: Se integra perfectamente con el ecosistema de Arduino, lo que facilita su programación y control. Pese a que esta placa no se encuentra incluida en el kit, existen ofertas donde se incluye la placa junto con el kit a un precio competitivo.
- Múltiples Ejes de Movimiento: El brazo robótico cuenta con seis ejes controlados por servomotores, lo que le confiere una gran amplitud de movimiento y precisión.
- Diseño Versátil: Puede ser ensamblado de diversas maneras para realizar distintas funciones. Además de la pinza incluida, se le pueden acoplar otros elementos como una cámara, un teléfono o incluso un panel solar para seguir el movimiento del sol.
- Kit de Montaje completo: el kit incluye la estructura mecánica del brazo, un conjunto de servomotores de tipo hobby que actúan como actuadores para cada articulación, una pinza/gripper simple, la electrónica de control basada en una placa Arduino y el cableado y tornillería necesarios para su montaje.



Figura 2.6 Montajes posibles del Braccio Tinkerkit, incluyendo algunos sustitutos de la pinza.

A continuación, se presentan las especificaciones técnicas del Braccio Tinkerkit:

Peso	792 g
Rango máximo de distancia de operación	80 cm
Altura máxima	52 cm
Anchura de la base	14 cm
Anchura de la pinza	9 cm
Longitud del cable	40 cm
Peso máximo a una distancia de funcionamiento de 32 cm	150 g
Peso máximo en la configuración mínima del Braccio	400 g

Tabla 2.2 Especificaciones técnicas principales del Braccio Tinkerkit, obtenidas directamente de la web oficial de compra de arduino [11].

La estructura del robot se compone por:

- Una base circular que se mueve sobre un eje vertical, permitiendo la rotación completa del robot.
- Una articulación «hombro» que une la base con la pieza siguiente «brazo».
- Una articulación «codo», la cual enlaza dos piezas semiidénticas: «brazo» y «antebrazo».
- Una articulación «muñeca vertical», que une «antebrazo» con una pequeña pieza que contiene la estructura de la pinza.
- Una articulación «muñeca rotatoria», donde se acopla la pinza permitiendo una rotación sobre un eje en dirección de la garra.
- La pinza o «gripper» encargada de la sujeción de objetos, cuyos valores toman 10 grados cuando está completamente abierta y de 73 grados cuando se encuentra cerrada.

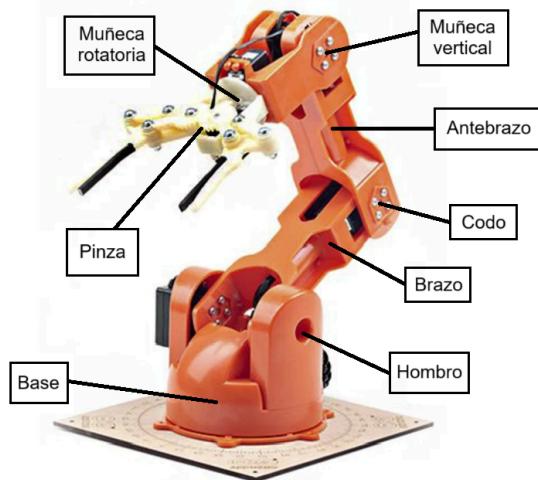


Figura 2.7 Estructura del Braccio Tinkerkit.

El movimiento de cada articulación es controlado por un servomotor de tipo hobby, los cuales son controlados mediante señales PWM (Pulse Width Modulation) generadas por la placa Arduino.

Los servomotores se encuentran catalogados por número ascendente desde la base hasta la pinza, siendo el SR 431 el encargado de mover las primeras cuatro articulaciones y el SR 311 los dos últimos, tal como se muestra en la Tabla 2.3.

Servomotor	Articulación	Rango de movimiento (°)	Modelo
M1	Base	0-180	SR 431
M2	Hombro	15-165	SR 431
M3	Codo	0-180	SR 431
M4	Muñeca vertical	0-180	SR 431
M5	Muñeca rotatoria	0-180	SR 311
M6	Pinza	10-73	SR 311

Tabla 2.3 Asignación de servomotores a las articulaciones del Braccio Tinkerkit, junto al rango de movimiento admisible de cada uno.

Las características de éstos se muestran en la Tabla 2.4, mostrando un mayor torque en el modelo dedicado a las zonas de mayor carga del dispositivo robot.

Característica	SR 431	SR 311
Voltaje operativo (V)	4.8-6.0	4.8-6.0
Torque (kg-cm) a 6 V	14.5	3.8
Torque (kg-cm) a 4.8 V	12.2	3.1
Peso (g)	62	27
Dimensiones (mm)	42 × 20.5 × 39.5	31.3 × 16.5 × 28.6
Velocidad (s/60°) a 6 V	0.18	0.14
Velocidad (s/60°) a 4.8 V	0.20	0.12

Tabla 2.4 Especificaciones comparativas de los servomotores utilizados en el Braccio Tinkerkit [11].

Adicionalmente a ello, en el kit se incluye una placa de expansión (shield) que permite conectar los servomotores y otros componentes electrónicos de manera sencilla y ordenada. Esta placa se conecta a una placa Arduino Uno y proporciona los pines necesarios para la conexión de los servos, así como una interfaz para la alimentación y el control de los mismos. Sus características técnicas se describen en la Tabla 2.5.

Versión	V4
Voltaje de funcionamiento	5 V
Consumo de potencia	20 mW
Corriente Máxima (M1-M4)	1.1 A
Corriente Máxima (M5-M6)	750 mA

Tabla 2.5 Especificaciones técnicas de la placa mostrada en la Figura 2.8.

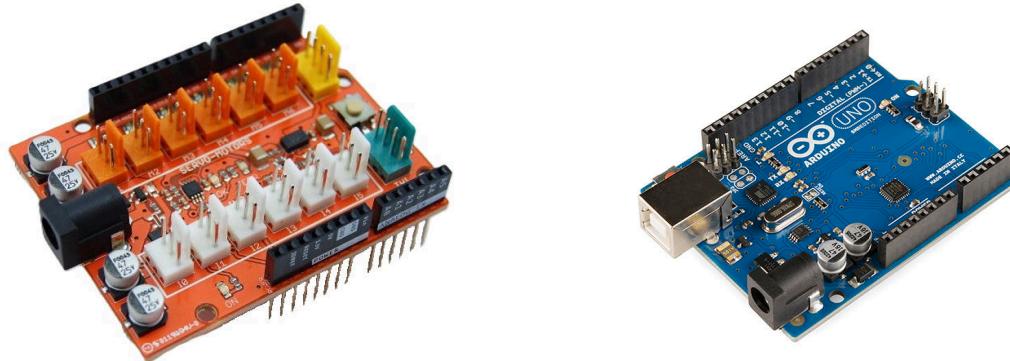


Figura 2.8 Placa de expansión (shield) utilizada para la conexión de los servomotores. En ella se puede visualizar la disposición de los pines naranjas etiquetados con la numeración correspondiente

Figura 2.9 Placa Arduino Uno utilizada como controlador principal del robot Braccio Tinkerkit.

La placa Arduino UNO es la base del sistema y encargada de la comunicación entre los diferentes componentes del robot. Esta placa se conecta a la shield y proporciona la interfaz necesaria para enviar las órdenes de los motores. Sus especificaciones técnicas se describen en la Tabla 2.6.

Microcontrolador	Microchip ATmega328P
Voltaje de funcionamiento	5 V
Voltaje de entrada	7-12 V
Pines de E/S digitales	14 (6 proporcionan salida PWM)
Pines de entrada analógica	6
Corriente DC por Pin de E/S	20 mA
Corriente CC para Pin de 3.3V	50 mA
Memoria Flash	32 KB
SRAM	2 KB
EEPROM	1 KB
Velocidad del reloj	16 MHz
Longitud	68.6mm
Ancho	53.4mm
Peso	25g

Tabla 2.6 Especificaciones técnicas de la placa mostrada en la Figura 2.9 [12].

3 Plataformas de desarrollo y simulación

El manipulador Braccio Tinkerkit forma parte del ecosistema Arduino, tal como se ha mencionado previamente. Debido a esta característica, puede ser simulado y controlado a través de diversas plataformas, destacando Matlab y ROS; siendo Gazebo, MoveIt y PyBullet las principales herramientas de simulación a destacar.

3.1 Plataformas de desarrollo

3.1.1 Matlab

Matlab/Simulink es un entorno de computación numérica y programación que ofrece un ecosistema integrado para el diseño, la simulación y la implementación de sistemas, incluyendo aplicaciones de robótica. A través de toolboxes específicos como *Robotics System Toolbox* y *Simscape* proporciona un entorno gráfico y basado en scripts para modelar y simular robots [13].

3.1.2 ROS

ROS (Robot Operating System) es un framework de código abierto caracterizado por ser el estándar para la investigación y el desarrollo en robótica. Facilita la comunicación y la gestión de procesos en un robot a través de un modelo de «nodos» que se comunican de forma centralizada [14].

3.1.3 ROS 2

ROS2 (Robot Operating System 2) es la nueva generación de ROS, diseñada para abordar las limitaciones de la primera versión y adaptarse a las necesidades actuales de la robótica. Está pensada para aplicaciones industriales, sistemas multi-robot y sistemas en tiempo real, mediante una arquitectura de comunicación descentralizada [2].

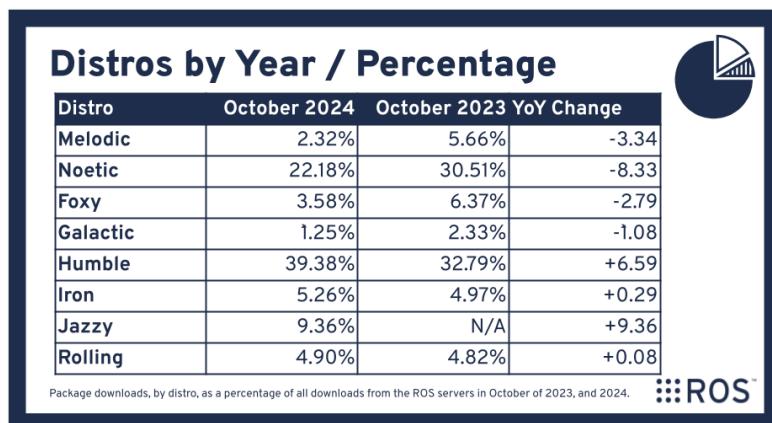


Figura 3.1 Tabla comparativa del incremento de descargas de ROS 2, siendo Humble y Jazzy sus exponentes, frente a la decadencia de las distribuciones anteriores, destacando Noetic como última versión de ROS 1 [15].

3.1.4 Comparativa de las plataformas

Característica	MATLAB	ROS	ROS2
Curva de Aprendizaje	Baja. Entorno gráfico muy intuitivo, visual y didáctico.	Media. Requiere aprender conceptos (nodos, tópicos, servicios), la compilación catkin y el uso de la terminal.	Alta. Similar a ROS1 pero con herramientas y conceptos más modernos.
Coste	Comercial: Requiere licencias para Matlab, Simulink y algunos toolboxes.	Gratis y Código Abierto.	Gratis y Código Abierto.
Arquitectura de Comunicación	Monolítica e integrada, simple y directa.	Centralizada: Depende de un nodo maestro (roscore), que actúa como servidor de nombres.	Descentralizada: Usa el protocolo DDS (Data Distribution Service); sin nodo maestro y mayor robustez y flexibilidad.
Simulación	Simscape Multibody para simulación dinámica y 3D. Buena pero no tan realista.	Gazebo para física realista.	Gazebo/Ignition y otros simuladores modernos.
Planificación de Movimiento	Robotics System Toolbox.	MoveIt: estándar maduro y con gran comunidad.	MoveIt2: versión para ROS2 en desarrollo activo.
Soporte para Braccio	Requiere importar URDF y configurar comunicación con Arduino. Escasos proyectos disponibles.	Excelente: variedad de paquetes y tutoriales disponibles.	Correcto: menor variedad de paquetes, con posibilidad de portar código de ROS1.
Comunidad	Fuerte soporte oficial (MathWorks) y comunidad académica.	Inmenso pero en declive: muchos paquetes obsoletos; sin nuevas versiones principales.	En crecimiento y activo: foco de la nueva investigación y desarrollo.

Tabla 3.1 Tabla comparativa entre MATLAB, ROS y ROS2 como opciones para la simulación y control del manipulador.

Tras comparar estas tres vertientes y en función de los objetivos formativos y profesionales planteados, la elección recomendada es ROS 2. El mayor problema del mismo recae en su complejidad, pues requiere de una mayor inversión inicial de tiempo para aprender sus herramientas y conceptos, así como de la instalación de un sistema operativo (Ubuntu) y los paquetes necesarios para su funcionamiento (ROS 2 Humble, Gazebo, RViz, MoveIt2, etc).

Sin embargo, aporta ventajas claras como su arquitectura descentralizada, un mejor soporte para requisitos de fiabilidad y tiempo real, integración con MoveIt2 y un ecosistema creciente orientado a la robótica profesional y de investigación.

3.2 Simuladores, planificadores y visores en ROS 2

A continuación se ofrece una comparativa práctica y orientada a decisiones entre las herramientas más relevantes del ecosistema ROS 2 en tres áreas: simulación física, planificación y visualización.

3.2.1 Simuladores físicos

- Gazebo: simulador con soporte para SDF/URDF, tratamiento de contactos y sensores, plugins en C++/Python y buena integración con ROS 2. Destaca por su alta fidelidad física y un ecosistema maduro de plugins y sensores, ideal para pruebas sim-to-real. Su principal limitación es una instalación y configuración más compleja, con un coste computacional mayor.
- PyBullet: motor ligero y rápido, fácil de usar desde Python, muy útil para prototipado rápido, simulación de grandes lotes y experimentos de aprendizaje por refuerzo. Destaca por su velocidad y simplicidad, pero con una integración con ROS 2 menos directa y una simulación de sensores más limitada.
- Webots: entorno todo-en-uno con fuerte enfoque educativo y ejemplos listos. Destaca por su rápido arranque y buen soporte de sensores. Su mayor limitante es constar de un modelo de licencias para ciertas características, con un ecosistema menor optimizado para investigación avanzada.

3.2.2 Planificación

- MoveIt2: framework principal para planificación de movimiento en ROS 2, integra OMPL, planificación de trayectoria, planificación de agarres y soporte para `ros2_control`. Pese a ser un sistema complejo, la integración directa con ROS 2 y sus herramientas para planificación y ejecución lo posicionan como una opción robusta para aplicaciones robóticas, mejorando a su predecesor.
- OMPL (Open Motion Planning Library): librería de planificadores sampling-based (RRT, PRM, etc.) usada por MoveIt2. Ofrece una amplia gama de algoritmos, siendo muy configurable. Sin embargo, requiere de un framework que gestione escenas y la ejecución de planes, por lo que se usa típicamente junto a MoveIt2.

3.2.3 Visualización y depuración

- RViz2: visor 3D estándar en ROS 2; muestra TF, tópicos, nubes, planes y estados del robot. Su principal ventaja es la integración nativa, siendo extensible mediante displays y plugins. Sin embargo, no ofrece grandes diferencias frente a Rviz, compartiendo su interfaz clásica.
- Foxglove Studio: herramienta moderna de visualización basada en web/desktop con soporte para ROS 2. Ofrece una interfaz moderna, con trazado de datos y vistas configurables. Se encuentra peor integrada en MoveIt2, pero su popularidad está en aumento y se están desarrollando más tutoriales y recursos para su uso.

3.2.4 Elección de herramientas

En base a las características y limitaciones de cada herramienta, se puede diseñar un mapa de flujos de trabajo estratégicos para el desarrollo en la robótica moderna. El ecosistema ROS2 se nutre de una caja de herramientas modular donde la elección correcta depende directamente de la fase y el objetivo del proyecto.

Para un desafío como el «pick and place» del Braccio Tinkerkit, utilizar un prototipo rápido en PyBullet puede ser útil para la validación de un nuevo algoritmo de agarre en minutos, así como la familiarización con el entorno de los robots manipuladores. Una vez validado, la simulación de alta fidelidad en Gazebo garantiza que la trayectoria es físicamente coherente y que los sensores responderían correctamente, construyendo la confianza necesaria para el paso al hardware. Finalmente, la abstracción que provee `ros2_control` actúa como el puente crucial que permite que el mismo código, planificado con MoveIt2, opere de forma idéntica en el simulador y en el robot físico. Este enfoque, visualizado con la claridad de RViz2 y las herramientas de GUI que nos ofrece, completa un sistema perfectamente integrado en el entorno de la robótica industrial, pudiendo ampliar horizontes con la potencia de datos de Foxglove Studio.

No obstante, para el proyecto actual, se ha optado por la combinación de Gazebo, MoveIt2 y RViz2, dada su integración nativa y comodidad, dejando la puerta abierta a Foxglove para futuras iteraciones. En la Figura 3.2 se muestra un ejemplo del flujo de datos y herramientas relevantes durante el desarrollo del sistema.

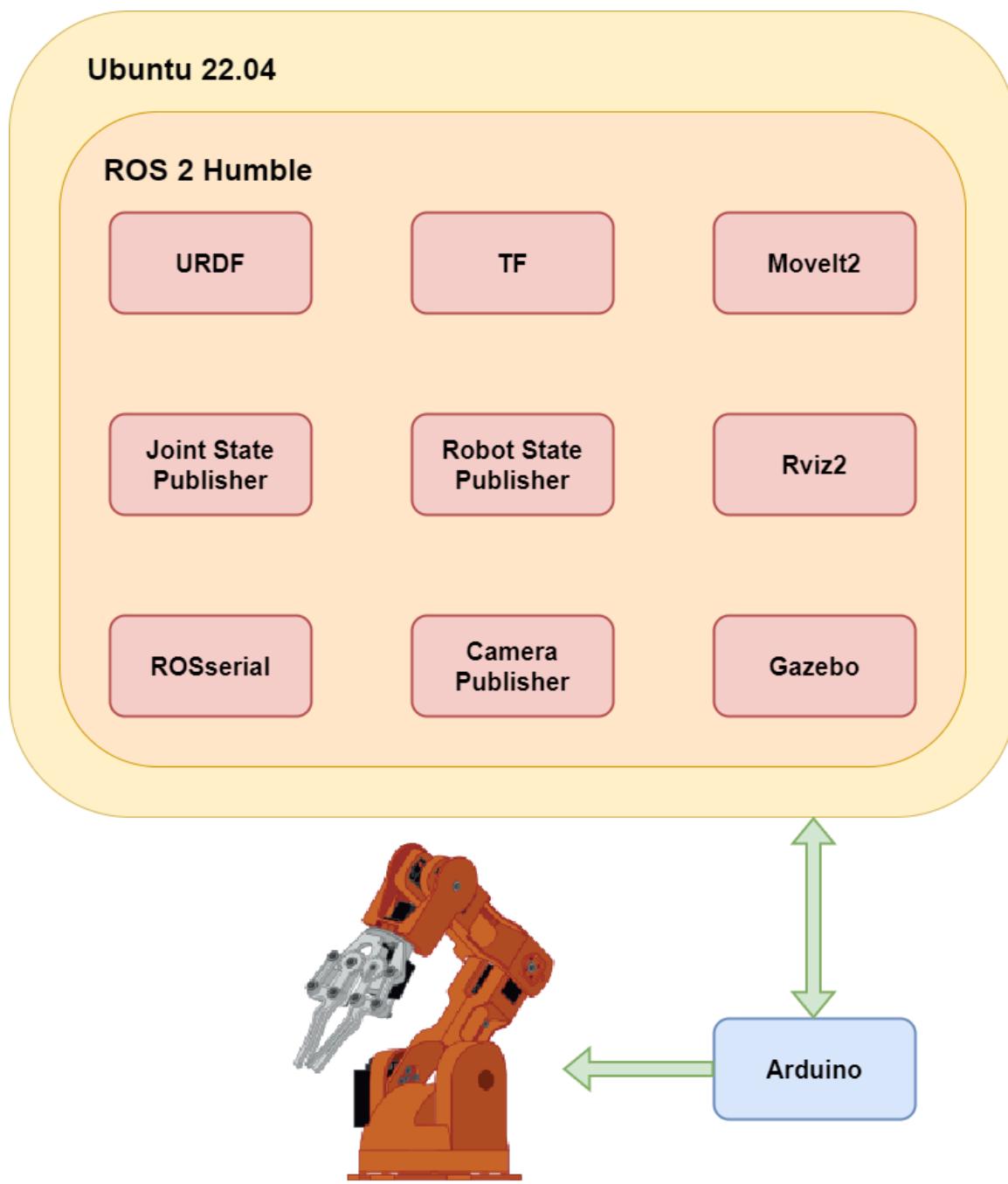


Figura 3.2 Diagrama de flujo del sistema ROS2 con MoveIt2, Gazebo y el resto de herramientas utilizadas para la simulación y control del proyecto.

4 Diseño del sistema

El sistema propuesto integra un robot manipulador Braccio Tinkerkit, controlado por una placa Arduino UNO, con un entorno de simulación en ROS 2 Humble. El flujo de datos y control se estructura en varios nodos ROS 2 que gestionan la percepción, planificación y ejecución de movimientos, tanto en simulación como en el robot físico.

4.1 Repositorio ROS2 Braccio

El repositorio base seleccionado para el desarrollo del sistema es el creado por el usuario jaMulet [2], debido principalmente a su estructura modular y funcional para la simulación y control del Braccio Tinkerkit en ROS 2.

Está organizado en varios paquetes que gestionan diferentes aspectos del sistema, siendo éstos:

1. Braccio_bringup: Paquete principal para lanzar los nodos necesarios para controlar el robot, tanto en simulación como en el hardware real.
2. Braccio_description: Contiene la descripción del robot en formato URDF, lo que permite su visualización y uso en herramientas como RViz.
3. Braccio_hardware: Define la interfaz de hardware para la comunicación con el robot real. Se basa en la comunicación serie USB para enviar mensajes a la plataforma Arduino y controlar el robot físico.
4. Braccio_moveit_config: Configuración para el uso de MoveIt2, mediante la implementación de los controladores del brazo y la pinza.
5. Braccio_ROS_Arduino: Contiene la biblioteca para implementar la interfaz de hardware del robot, basada en comunicación serial. Para el control de tareas se encuentra implementado un programador de éstas basado en la biblioteca TaskScheduler [16].

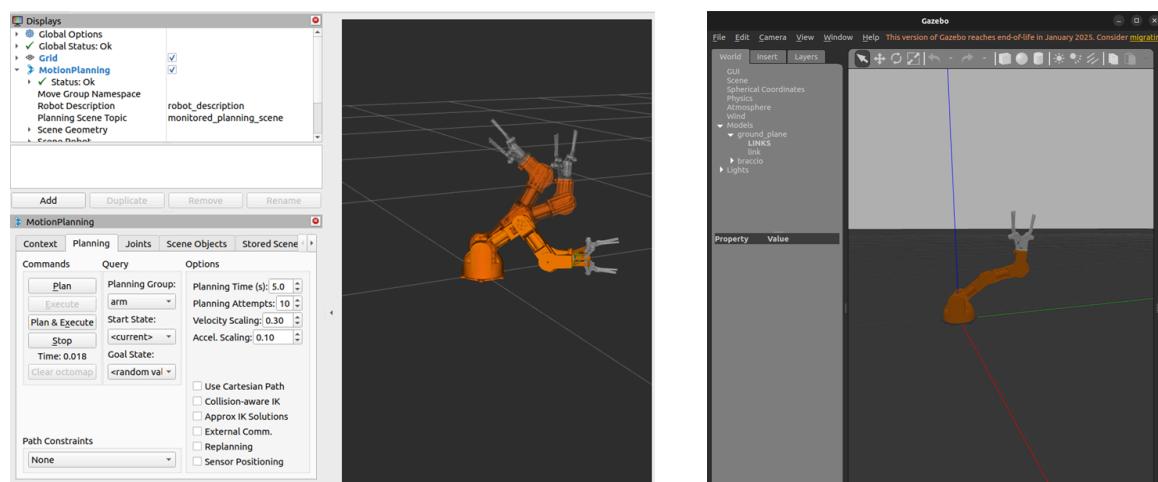


Figura 4.1 Ilustración de la simulación en RViz y Gazebo del robot manipulador realizando una trayectoria. En la izquierda, se puede observar el brazo en sus posiciones inicial, actual y final. A su derecha, la representación de Gazebo mostrando esa posición actual junto al entorno simulado.

El repositorio está diseñado para funcionar con dos modos, seleccionables en el lanzamiento del sistema:

- Simulación: Utiliza Gazebo para crear un entorno virtual con el robot Braccio. Esto permite probar la lógica de control y la planificación de movimientos sin necesidad del hardware físico. Se activa con el argumento «sim:=true».
- Hardware Real: Se comunica directamente con el robot Braccio a través de una conexión serie con una placa Arduino. El paquete *braccio_hardware* gestiona esta comunicación, mediante el argumento «sim:=false». Adicionalmente incluye una opción para probar la comunicación con el hardware, mediante «hw_test:=true».

4.2 Extensiones y mejoras implementadas

El repositorio original ha sido modificado y ampliado para incluir las nuevas funcionalidades que permitan lograr los objetivos propuestos, mejorar la experiencia de usuario y mantener esa modularidad característica, reflejando ese trabajo en el siguiente repositorio [17].

Se han añadido dos nuevos paquetes principales:

1. Braccio_gamepad_teleop: Permite el control del robot mediante un mando conectado a través del puerto serie. Se ha implementado el mapeo de los botones y joysticks del mando a comandos específicos para mover las articulaciones del robot, tanto en simulación como en el hardware real, utilizando un mando de PlayStation 4 para las pruebas.
2. Braccio_vision: Incluye los nodos y scripts necesarios para tareas de visión por computadora y control. En éste se abordan aspectos como la detección de objetos, calibración de cámaras y aplicaciones de «pick and place».
3. Sim-to-real: Contiene los scripts necesarios para la transferencia de la simulación a la realidad, permitiendo que los modelos y controladores desarrollados en Gazebo se apliquen al robot físico. Se nutre de la mayoría de scripts de visión y los complementa con adaptaciones específicas para el hardware.

Estas implementaciones se tratarán en detalle en las siguientes secciones, explicando su diseño, integración con el sistema y los beneficios que aportan al proyecto global.

4.3 Entorno de simulación

El entorno donde el robot opera se define en el archivo *braccio.world*, ubicado en la carpeta *gazebo* de *braccio_description*. Este archivo actúa como el escenario virtual en el que el robot Braccio interactúa con otros objetos y el entorno. Su función es establecer todo lo que existe en el mundo antes de que el manipulador aparezca.

En el siguiente se especifican varios elementos clave:

- Define la gravedad del mundo, estableciendo el terreno y la luminosidad del mismo a través de un modelo de sol.
- Incluye dos modelos estáticos de forma hexagonal de colores verde y azul, como superficies para el depósito de los objetos.
- Registra los plugins *gazebo_ros_state* y *gazebo_link_attacher*, necesarios para la obtención de la posición de cada objeto en tiempo real y para la simulación del agarre de los objetos, respectivamente.

4.3.1 Robot manipulador

El robot manipulador Braccio se describe principalmente en la carpeta *braccio_description* comentada previamente. En el interior de la misma se encuentra una carpeta llamada *urdf*, que contiene el archivo *braccio.urdf.xacro*, entre otros. Éste es el archivo de configuración principal y se nutre del resto de archivos, donde se definen las diferentes partes del robot, como los enlaces y las articulaciones, así como sus propiedades físicas y visuales.

Adicionalmente, se encuentra *braccio.ros2_control.xacro*, quién apunta a *braccio_controllers.yml*, donde se especifican los controladores PID para cada articulación del robot, así como la configuración de los actuadores y sensores.

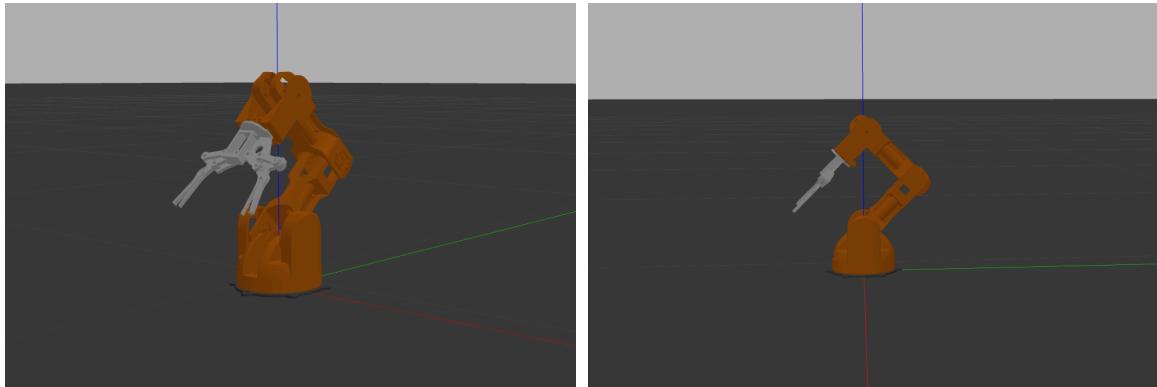


Figura 4.2 Representación del robot Braccio Tinkerkit en Gazebo, mostrando dos perspectivas diferentes del manipulador.

4.3.2 Spawner de Cámara y Cubos

Para la simulación de tareas de percepción y manipulación, se han añadido varios elementos al entorno de Gazebo. El ejecutable encargado de esta acción es *vision_simulation.launch.py*, ubicado en la carpeta *launch* del paquete *braccio_vision*. Este ejecutable lanza el mundo junto al robot y, pasado un tiempo, inicia el spawner de la cámara y los cubos.

En primer lugar, se ha implementado un nodo que simula una cámara ubicada en el centro del mundo, a 0.6m de altura, proporcionando una vista cenital del entorno. Las físicas y plugins de ésta se encuentran definidas mediante *overhead_camera.urdf.xacro*, quien, junto a *camera_spawner.py*, se encarga de su inicialización en el mundo y de la publicación de los datos de la cámara en los tópicos correspondientes.

Posteriormente, se han creado varios modelos de cubos de diferentes colores (rojo, verde y azul) y un tamaño de 3cm que actúan como objetos a manipular. Estos modelos están definidos mediante una plantilla SDF y se pueden instanciar en el mundo a través de *object_spawner.py*. Los cubos tienen propiedades físicas realistas, como masa y fricción, para asegurar una interacción coherente con el robot.

Estos elementos permiten simular escenarios de pick-and-place donde el robot debe identificar, agarrar y mover los cubos a ubicaciones específicas dentro del entorno simulado.

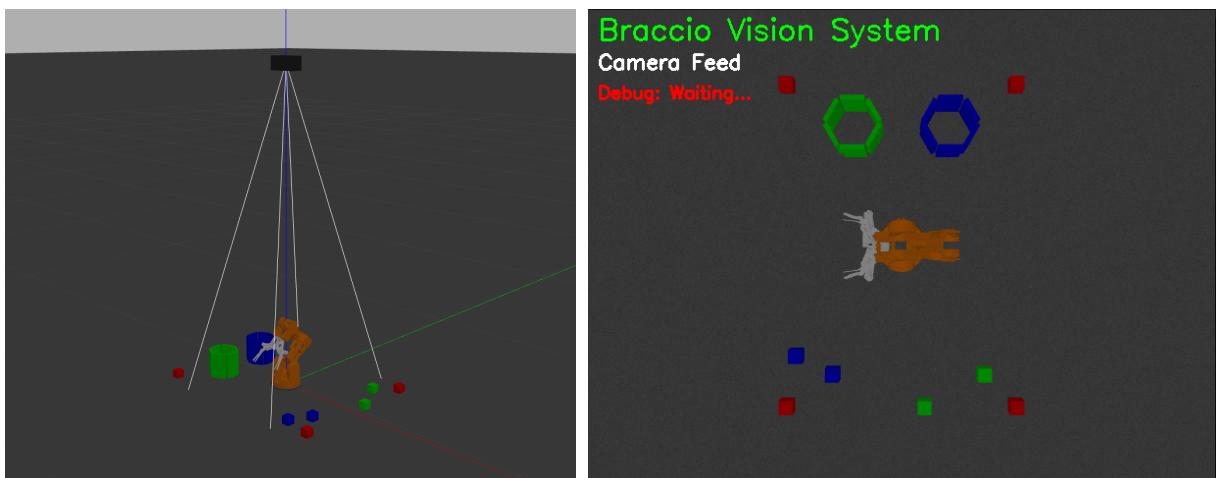


Figura 4.3 Representación completa del entorno de simulación, incluyendo el robot, los recipientes, la cámara y los objetos manipulables. A la derecha, la vista cenital de la cámara simulada 0.6m sobre el suelo, mostrando los cubos de diferentes colores.

5 Control mediante PS4 controller

El sistema actual propone la implementación de un sistema de control basado en la retroalimentación visual, utilizando la cámara simulada para ajustar dinámicamente los comandos de movimiento del robot. Sin embargo, previa a ésta, se ha optado por un sistema de control manual mediante un mando de PS4, con el objetivo de familiarizarse con el entorno y realizar las pruebas pertinentes que verifiquen el correcto funcionamiento del mismo. En la carpeta *braccio_gamepad_teleop* se encuentra el nodo principal encargado de esta tarea, *gamepad_teleop.py*, y un launch que arranca el sistema.

5.1 Arquitectura y filosofía de control

Para la realización de este sistema de teleoperación se ha optado por una filosofía basada en el control incremental directo en el espacio de articulación. Este sistema implica dos acciones fundamentales:

1. Los joysticks no controlan el movimiento de la pinza en un eje XYZ, sino que controlan directamente la velocidad de rotación de las articulaciones individuales del robot. En [18], se explica la diferencia entre utilizar este control directo (JointJog), frente al uso de un sistema basado en la cinemática inversa (TwistStamped).
2. El nodo mantiene una variable interna, *self.current_joint_positions*, que almacena la posición objetivo actual de cada articulación. De este modo, cada vez que se mueve el joystick, el nodo no establece una posición fija, sino que añade o resta un pequeño incremento a la posición actual, permitiendo un movimiento del robot mucho más fluido y suave. En la Figura 5.1 se muestra dicho incremento de una forma mucho más clara.

De este modo, la arquitectura del sistema es muy sencilla, teniendo un único nodo encargado de:

1. Recibir los datos crudos del gamepad.
2. Interpretar los movimientos de los joysticks y los botones.
3. Mantener un registro del estado de las articulaciones del robot.
4. Calcular los nuevos comandos de posición para cada articulación.
5. Gestionar la lógica de «pick and place» interactuando con los servicios de Gazebo.
6. Publicar los comandos directamente a los controladores del robot.

```
[gamepad teleop]: [Input: LX=0.94 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [1.9540665677487845, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
[gamepad teleop]: [Input: LX=0.95 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [1.9635715249478811, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
[gamepad teleop]: [Input: LX=0.97 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [1.9732416037023062, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
[gamepad teleop]: [Input: LX=0.95 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [1.9827465609014028, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
[gamepad teleop]: [Input: LX=0.95 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [1.992416396558278, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
[gamepad teleop]: [Input: LX=0.95 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [2.0019215968549244, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
[gamepad teleop]: [Input: LX=0.95 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [2.011426554054021, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
[gamepad teleop]: [Input: LX=0.95 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [2.0209315112531177, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
[gamepad teleop]: [Input: LX=0.95 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [2.0304364684522143, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
[gamepad teleop]: [Input: LX=0.95 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [2.039941425651311, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
[gamepad teleop]: [Input: LX=0.95 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [2.0494463828504075, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
[gamepad teleop]: [Input: LX=0.95 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [2.058951340049504, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
[gamepad teleop]: [Input: LX=0.95 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [2.0684562972486007, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
[gamepad teleop]: [Input: LX=0.95 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [2.0779612544476973, 0.7687281474769116, 3.14, 3.127, 1.572, 0.1]
```

Figura 5.1 Lectura del terminal de Ubuntu. Representa el incremento en la posición de la articulación base al mantener el joystick inclinado hacia la izquierda durante unos segundos, siendo LX el valor del joystick izquierdo y el primer término de POS, la posición angular de la base del robot en radianes. El resto de valores nulos corresponden al estado de los botones que no se han pulsado.

5.2 Mapeo de botones y joysticks

El control de las acciones del robot se realiza mediante un mapeo de los botones y joysticks del mando de PS4 a los comandos de movimiento del robot, tal como se puede observar en la Figura 5.2.

5.2.1 Joysticks

Se encuentran habitualmente dos joysticks en el mando, el stick izquierdo y el derecho, cada uno con los ejes horizontal y vertical. La asignación de los ejes a las articulaciones del robot es la siguiente:

- Stick izquierdo:
 - Eje X: Joint_base (base)
 - Eje Y: Joint_1 (hombro)
- Stick derecho:
 - Eje X: Joint_3 (antebrazo)
 - Eje Y: Joint_2 (codo)

A esta configuración se incluye un deadzone para evitar movimientos no deseados por pequeñas variaciones en la posición de los joysticks.

5.2.2 Botones

La cantidad de botones en el mando es ampliamente superior, decidiendo destinar los gatillos para el control de la pinza y articulación restante, y los botones frontales para acciones específicas de «pick and place»:

- Gatillos L1 y R1: incrementan y decrementan Joint_4 (muñeca).
- Gatillos L2 y R2: abren y cierran Gripper_Joint (pinza).
- Triángulo: realiza la acción de «pick», intentando agarrar el objeto más cercano.
- Cruz: realiza la acción de «place», intentando soltar el objeto agarrado.

Asimismo, se ha configurado una variable global, llamada *velocity_factor*, que permite ajustar la velocidad de movimiento del robot, afectando a la suavidad y sensibilidad de los movimientos.



Figura 5.2 Diagrama del mando de PS4 con el mapeo de los botones y joysticks a las articulaciones y acciones del robot mencionadas previamente.

5.3 Flujo de datos y control

Tal como se ha comentado previamente, esta implementación prioriza la sencillez y respuesta inmediata. De este modo, a continuación se detalla el flujo de datos con el fin de comprender adecuadamente su funcionamiento:

1. El launch arranca el nodo *joy_node* [19], encargado de la publicación de datos en crudo del gamepad en /joy y *gamepad_teleop.py* se suscribe a este tópico.
2. El nodo principal:
 - Recibe los datos del gamepad a través del tópico, la lista de modelos en simulación de /gazebo/model_states y obtiene de las TF la posición del gripper para calcular posteriormente su proximidad a los objetos.
 - Calcula y publica los comandos para los 5 joints del brazo y el gripper en los tópicos correspondientes. Sin embargo, previo a ello, verifica que ninguna de las posiciones objetivo calculadas exceda los límites definidos para cada articulación.
 - Conecta con los servicios de Gazebo para la lógica de «pick and place», llamando a los servicios de *attach* y *detach* del plugin *libgazebo_link_attacher.so*. Éstos:
 - Detectan el objeto más cercano utilizando la información de los sensores y la posición del gripper.
 - Comprueban que la distancia entre el gripper y el objeto es adecuada, inferior a 15 cm.
 - Si ambas condiciones se cumplen, envía la petición de *attach/detach* al servicio correspondiente, mostrado en la Figura 5.3
3. El sistema *ros2_control* recibe estas trayectorias y las ejecuta en el robot simulado o real.

Este control proporciona una vía rápida y segura para validar la arquitectura de control, probar el mapeo de ejes y calibrar parámetros como el *velocity_factor* antes de automatizar. Es una herramienta útil para detectar límites de articulación, comprobar la comunicación con Gazebo/rosl2_control y afinar la experiencia de teleoperación, previa a la implementación de un sistema de control basado en visión del manipulador real.

```
[python3-2] [INFO] [1756212919.326506584] [gamepad_teleop]: Modelos en model_states: ['ground_plane', 'zona_hexagono', 'zona_hexagono_azul', 'braccio', 'overhead_camera', 'corner1', 'corner2', 'corner3', 'corner4', 'green_cubel', 'green_cube2', 'blue_cubel']
[python3-2] [INFO] [1756212919.32703247] [gamepad_teleop]: Posición gripper TF: -0.18761242586964225, -0.12309485637507234, 0.05376704555478389
[python3-2] [INFO] [1756212919.327558657] [gamepad_teleop]: Comparando con modelo: zona_hexagono, posición: (0.0, 2.7755575615628914e-17, 0.0)
[python3-2] [INFO] [1756212919.328024207] [gamepad_teleop]: Comparando con modelo: zona_hexagono_azul, posición: (2.7755575615628914e-17, -4.163336342344337e-17, 0.0)
[python3-2] [INFO] [1756212919.328293691] [gamepad_teleop]: Comparando con modelo: zona_hexagono, posición: (0.2307415463118462, 0.0, 0.0)
[python3-2] [INFO] [1756212919.328503457] [gamepad_teleop]: Comparando con modelo: overhead_camera, posición: (0.0, 0.0, 1.2)
[python3-2] [INFO] [1756212919.328709994] [gamepad_teleop]: Comparando con modelo: corner1, posición: 1.167990047844116
[python3-2] [INFO] [1756212919.328911275] [gamepad_teleop]: Comparando con modelo: corner1, posición: (-0.35, -0.25, 0.014999999999755)
[python3-2] [INFO] [1756212919.329114418] [gamepad_teleop]: Distancia a corner1: 0.5537462593543822
[python3-2] [INFO] [1756212919.329296701] [gamepad_teleop]: Comparando con modelo: corner2, posición: (0.35, -0.25, 0.014999999999755)
[python3-2] [INFO] [1756212919.329481990] [gamepad_teleop]: Distancia a corner2: 0.2097081866105175
[python3-2] [INFO] [1756212919.329481990] [gamepad_teleop]: Comparando con modelo: corner3, posición: (0.35, 0.25, 0.014999999999755)
[python3-2] [INFO] [1756212919.329663833] [gamepad_teleop]: Distancia a corner3: 0.4087448836457087
[python3-2] [INFO] [1756212919.329846217] [gamepad_teleop]: Comparando con modelo: corner4, posición: (-0.35, 0.25, 0.014999999999755)
[python3-2] [INFO] [1756212919.330028688] [gamepad_teleop]: Comparando con modelo: green_cubel, posición: (0.65592704753598, 0.0, 0.0)
[python3-2] [INFO] [1756212919.330208688] [gamepad_teleop]: Comparando con modelo: green_cubel, posición: (0.35, 0.0499999999999996, 0.014999999999755)
[python3-2] [INFO] [1756212919.330391895] [gamepad_teleop]: Distancia a green_cubel: 0.24048791519848772
[python3-2] [INFO] [1756212919.330573297] [gamepad_teleop]: Comparando con modelo: green_cubel2, posición: (0.28, 0.18, 0.014999999999755)
[python3-2] [INFO] [1756212919.330753907] [gamepad_teleop]: Distancia a green_cubel2: 0.3192253743607857
[python3-2] [INFO] [1756212919.330943294] [gamepad_teleop]: Comparando con modelo: blue_cubel, posición: (0.28083722158545094, -0.1506971845372727, 0.014999999999755)
[python3-2] [INFO] [1756212919.331127250] [gamepad_teleop]: Distancia a blue_cubel: 0.10425522039243953
[python3-2] [INFO] [1756212919.331310125] [gamepad_teleop]: Comparando con modelo: blue_cubel2, posición: (0.23976770213997542, -0.24903742142327173, 0.014999274085567619)
[python3-2] [INFO] [1756212919.331530520] [gamepad_teleop]: Distancia a blue_cubel2: 0.1417202971999568
[python3-2] [INFO] [1756212919.3317721921] [gamepad_teleop]: Modelo más cercano: blue_cubel, distancia: 0.10425522039243953
[python3-2] [INFO] [1756212919.331968175] [gamepad_teleop]: Pick por proximidad: blue_cubel (distancia: 0.104m)
[python3-2] [INFO] [1756212919.332289160] [gamepad_teleop]: Intentando attach: blue_cubel/link
```

Figura 5.3 Servicio de Gazebo para el pick and place de objetos mediante el plugin *gazebo_link_attacher*. En la imagen se observa la petición de attach al servicio, el cálculo de las distancias respecto la posición del gripper y los cubos; y tras la verificación del umbral de cercanía, la ejecución de la acción entre la pinza y el blue_cubel.

6 Percepción y localización de objetivos

La percepción y detección de objetos es un componente elemental en el sistema de pick-and-place, ya que proporciona la información necesaria para que el robot identifique y localice los objetos a manipular sin necesidad de intervención humana directa, de forma automática. Ante ello, se ha investigado la metodología empleada por Will Stedden en su proyecto [20], basado en un flujo de trabajo modular y reproducible. Sin embargo, debido al enfoque de calibración manual, se ha optado por la implementación de un sistema de calibración automática basado en homografía, similar al explicado por Nathan Naerts [21], donde se han utilizado arucos para delimitar el espacio de trabajo y la conversión de coordenadas. No obstante, en este caso, se ha optado por sustituir sus arucos por cubos rojos referenciales.

El sistema de percepción sigue el flujo modular mostrado en la Figura 6.1 donde se facilita la validación en simulación y la transferencia al robot real. Todos estos archivos se encuentran en la carpeta *braccio_vision*.

- Los parámetros intrínsecos del sensor de la cámara, así como la configuración de los umbrales de detección se almacenan en *vision_config.yaml*.
- El sensor de la cámara publica las imágenes, pudiendo visualizar la imagen desde *camera_viewer.py*.
- El detector de objetos procesa las imágenes detectando contornos por color y calculando centroides en píxeles. Luego, publica la información en la imagen debug para su visualización y en un tópico con las coordenadas de los objetos detectados.
- Mediante la posición de los objetos detectados en píxeles y los objetos de referencia en metros, se realiza el cálculo de la matriz de homografía, que se almacena en *camera_calibration.json*.
- Esta matriz se utiliza para convertir las coordenadas de píxeles a coordenadas del mundo real de aquellos nuevos objetos que el sistema desconoce su posición.

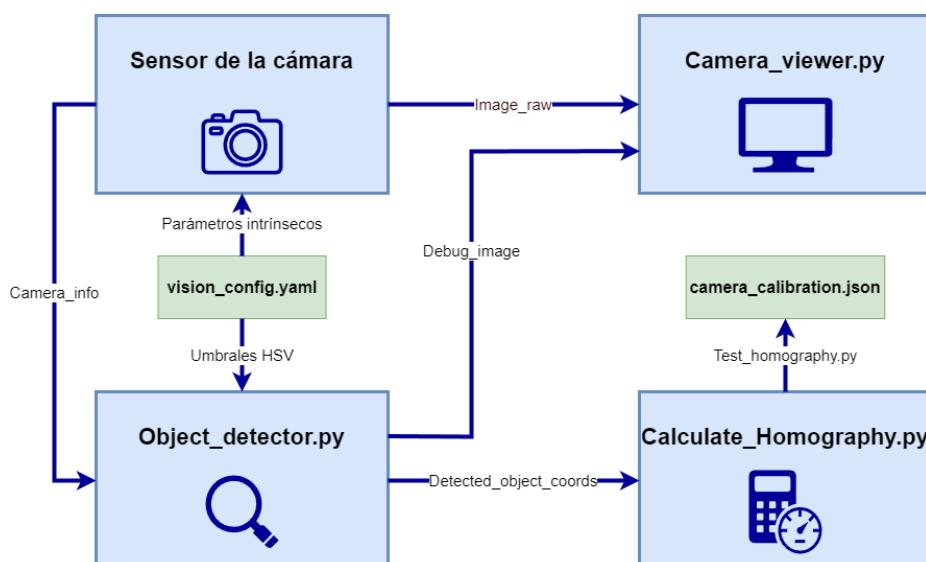


Figura 6.1 Esquema del flujo de percepción y detección de objetos para el sistema de pick-and-place.

6.1 Sensor de la cámara

El sensor de la cámara se configura para simular una cámara RGB convencional, con parámetros ajustables como resolución, campo de visión (FOV), tasa de frames y posición fija en el entorno. Tal como se describió en la Sección 4.3.2, la configuración principal se realiza en `overhead_camera.urdf.xacro` mediante `camera_spawner.py`.

Esta configuración se complementa con `vision_config.yaml`, donde se definen los siguientes parámetros clave del sensor:

- Resolución: 640x480 px.
- FOV: 80°.
- Tasa de frames: 30 FPS.
- Posición: fija en el entorno.
- Altura: 0.6 metros.

De los cuales se obtienen algunos valores como las distancias focales: $fx=381.96$ y $fy=381.96$ o el centro de la cámara $cx=320$ y $cy=240$.

La cámara simulada, por su parte, publica imágenes en el tópico `/overhead_camera/image_raw` e información de la misma en `/overhead_camera/camera_info`. Estas imágenes son consumidas por el nodo de detección de objetos para su procesamiento y por el subsistema de visualización `camera_viewer.py`. Esta última permite la visualización de la imagen en tiempo real y la imagen procesada con la información de los objetos detectados, mostrado en la Figura 6.2.

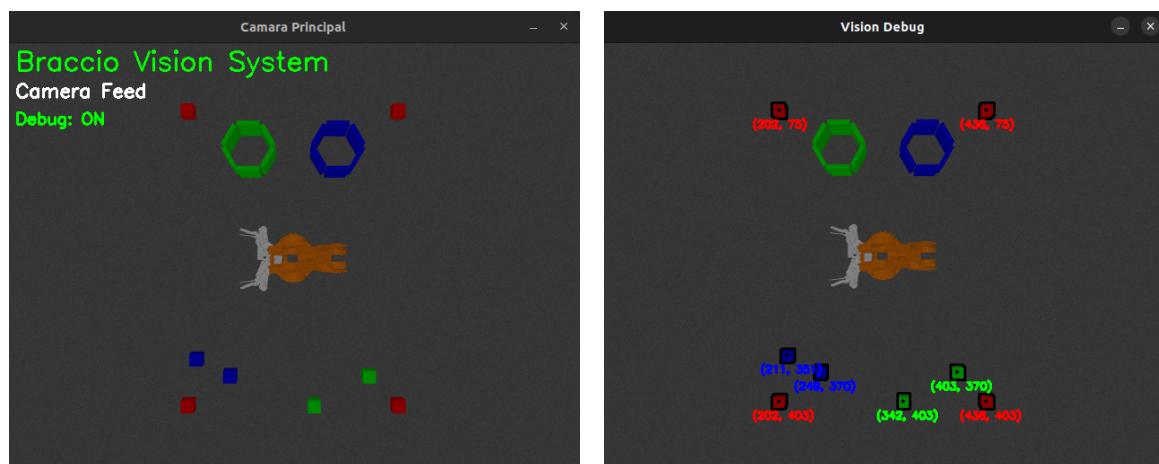


Figura 6.2 Representación de las cámaras simuladas, mostrando la vista del sensor raw y la vista de depuración tras la detección de los objetos. En esta última, se puede observar el marcador de cada objeto detectado junto con las coordenadas de su centro en sus respectivos colores.

6.2 Detección de objetos

El subsistema de detección de objetos tiene por objetivo localizar los cubos presentes en el área de trabajo y publicar su posición en un formato utilizable por el resto de la cadena de control y por las herramientas de visualización empleadas en el proyecto.

El sistema recibe información de la cámara simulada y de su configuración, así como los umbrales HSV y parámetros de filtrado de área definidos en `vision_config.yaml`. El procesamiento de las imágenes sigue un flujo clásico de visión por computadora basado en OpenCV, que incluye:

- Conversión BGR → HSV y aplicación de rangos colorimétricos definidos en el archivo de configuración mencionado, detectando así los cubos rojos, verdes y azules.
- Operaciones morfológicas y filtrado por área para eliminar ruido y detecciones espurias.
- Detección de contornos y cálculo del centroide en píxeles (cx , cy) para cada objeto relevante.
- Conversión píxel → mundo mediante la función `pixel_to_world`. Esta conversión se apoya en un modelo geométrico simple, Figura 6.3, pudiendo ser sustituido posteriormente por la homografía calibrada almacenada en `camera_calibration.json` para aumentar la precisión.

$$X_{\text{norm}} = \frac{\text{pixel}_x - c_x}{f_x} \quad Y_{\text{norm}} = \frac{\text{pixel}_y - c_y}{f_y}$$

$$X_{\text{world}} = X_{\text{norm}} \cdot Z$$

$$Y_{\text{world}} = Y_{\text{norm}} \cdot Z$$

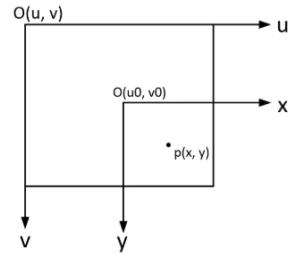


Figura 6.3 Ecuaciones para el cálculo de las coordenadas normalizadas y su proyección al plano, basadas en el modelo geométrico mostrado en la derecha [22]. Los parámetros utilizados se corresponden con la posición del objeto en píxeles (pixel_x , pixel_y), el centro de la cámara (c_x , c_y), las distancias focales (f_x , f_y) y la altura de la cámara (Z).

Tras esto, el nodo publica la siguiente información:

- `/vision/debug_image`: imagen anotada con máscaras y centroides, destinada a diagnóstico y visualización.
- `/vision/object_markers`: marcadores para representación en RViz.
- `/detected_object_coords`: coordenadas en el sistema de referencia del mundo para cada detección.

Esta información proporcionada permite al usuario depurar discrepancias entre el simulador y el hardware, y actuar como enlace entre la detección de objetos y los subsistemas de planificación y ejecución MoveIt2 o teleop. Por otra parte, mantener una separación clara entre canales de diagnóstico (`debug_image`, `markers`) y canales de decisión (`detected_object_coords`) reduce el acoplamiento entre componentes y facilita las pruebas.

```
[INFO] [1756322959.830451255] [object_detector]: ⚙️ Objetos detectados: 8
[INFO] [1756322959.830753790] [object_detector]: 📸 Publicando markers para 8 objetos
[INFO] [1756322959.831971035] [object_detector]: 🎨 CUBOS ROJOS DETECTADOS: 4
[INFO] [1756322959.832175620] [object_detector]: ⚡ Cubo rojo 1: pixel(436, 403)
[INFO] [1756322959.832368194] [object_detector]: ⚡ Cubo rojo 2: pixel(202, 403)
[INFO] [1756322959.832575204] [object_detector]: ⚡ Cubo rojo 3: pixel(436, 75)
[INFO] [1756322959.832760454] [object_detector]: ⚡ Cubo rojo 4: pixel(202, 75)
[INFO] [1756322959.832962455] [object_detector]: 🟢 CUBOS VERDES DETECTADOS: 2
[INFO] [1756322959.833161831] [object_detector]: ⚡ Cubo verde 1: pixel(342, 403)
[INFO] [1756322959.833332233] [object_detector]: ⚡ Cubo verde 2: pixel(403, 370)
[INFO] [1756322959.833497387] [object_detector]: 🔵 CUBOS AZULES DETECTADOS: 2
[INFO] [1756322959.833661949] [object_detector]: ⚡ Cubo azul 1: pixel(249, 370)
[INFO] [1756322959.833823185] [object_detector]: ⚡ Cubo azul 2: pixel(211, 351)
[INFO] [1756322959.833995150] [object_detector]: ⚡ Cubo verde en pixel(342, 403) temporalmente bloqueado
[INFO] [1756322959.834163840] [object_detector]: ⚡ Cubo verde en pixel(403, 370) temporalmente bloqueado
[INFO] [1756322959.834369217] [object_detector]: ⚡ Cubo azul en pixel(249, 370) temporalmente bloqueado
[INFO] [1756322959.834553264] [object_detector]: ⚡ Cubo azul en pixel(211, 351) temporalmente bloqueado
```

Figura 6.4 Lectura del terminal de Ubuntu tras la ejecución de `object_detector.py`. Representa la cantidad de objetos detectados, clasificados por su color, junto a sus coordenadas en píxeles, coincidentes con los datos de la Figura 6.2. Los cubos temporalmente bloqueados son aquellos destinados a la manipulación en el entorno simulado, estudiados en secciones posteriores.

6.3 Matriz de Homografía

La matriz de homografía se utiliza para mapear puntos de la imagen a coordenadas del mundo, teniendo en cuenta la perspectiva de la cámara. Bien es cierto que tiene infinidad de usos y, al tratarse de una cámara cenital, su relevancia es menor pues se trabaja en un escenario 2D ideal, sin distorsión. No obstante, para este proyecto y, de cara a la implementación real, es importante contar con esta matriz en términos de precisión.

El cálculo de esta matriz es bastante sencillo. Para ello, se han colocado 4 cubos rojos en posiciones conocidas dentro del entorno simulado. Mediante el script de detección de objetos anterior se han detectado éstos y calculado sus centroides en píxeles, tal como se muestra en la Figura 6.2, obteniendo las posiciones indicadas en la Tabla 6.1.

Cubo rojo	Posición real XY (m)	Posición en píxeles XY (px)
1	-0.35, -0.25	202, 75
2	-0.35, 0.25	436, 75
3	0.35, 0.25	436, 403
4	0.35, -0.25	202, 403

Tabla 6.1 Posiciones de los cubos rojos en el entorno simulado y sus centroides capturados por la cámara en píxeles.

Combinando esta posición en píxeles con su ubicación en el mundo real, se ha aplicado la función `cv2.findHomography` con el fin de la obtención de esta matriz.

La matriz obtenida se puede visualizar en la Figura 6.5 y se almacena en `camera_calibration.json` para su uso posterior en la conversión de coordenadas de la cinemática inversa.

```
ivan@Vlctus-by-HP-Ivan:~/Escritorio/Braccio-Tinkerkit-Ardutino/braccio_vision/scripts$ python3 calculate_homography.py
Matriz de homografía calculada:
[[ -2.62593536e-08  2.12423960e-03 -5.08588300e-01]
 [ 2.12680544e-03 -1.03919144e-08 -6.79031349e-01]
 [-6.73234617e-06 -1.04218054e-05  1.00000000e+00]]
Homografía guardada en braccio_vision/config/camera_calibration.json
```

Figura 6.5 Lectura del terminal de Ubuntu tras la ejecución de `Calculate_homography.py`. Representa la matriz de homografía calculada .

Finalmente, se ha probado dicha matriz mediante un script de validación que compara las posiciones conocidas de los cubos con las posiciones calculadas a partir de sus centroides en píxeles, así como con posiciones interiores a ese área definida. Los resultados muestran un error prácticamente insignificante, lo cual es perfecto para las tareas de pick-and-place previstas.

```
ivan@Vlctus-by-HP-Ivan:~/Escritorio/Braccio-Tinkerkit-Ardutino/braccio_vision/scripts$ python3 test_homography.py
Matriz de homografía cargada:
[[ -2.62593536e-08  2.12423960e-03 -5.08588300e-01]
 [ 2.12680544e-03 -1.03919144e-08 -6.79031349e-01]
 [-6.73234617e-06 -1.04218054e-05  1.00000000e+00]]

Proyección de pixel a mundo:
Pixel [202, 75] -> Mundo estimado: (-0.3500, -0.2500) | Mundo real: [-0.35, -0.25]
Pixel [436, 75] -> Mundo estimado: (-0.3506, 0.2492) | Mundo real: [-0.35, 0.25]
Pixel [436, 403] -> Mundo estimado: (0.3500, 0.2500) | Mundo real: [0.35, 0.25]
Pixel [202, 403] -> Mundo estimado: (0.3494, -0.2508) | Mundo real: [0.35, -0.25]
Pixel [320, 240] -> Mundo estimado: (0.0012, 0.0016) | Mundo real: [0.0, 0.0]

Proyección inversa de mundo a pixel:
Mundo [-0.35, -0.25] -> Pixel estimado: (202.0, 75.0) | Pixel real: [202, 75]
Mundo [-0.35, 0.25] -> Pixel estimado: (436.4, 75.3) | Pixel real: [436, 75]
Mundo [0.35, 0.25] -> Pixel estimado: (436.0, 403.0) | Pixel real: [436, 403]
Mundo [0.35, -0.25] -> Pixel estimado: (202.4, 403.0) | Pixel real: [202, 403]
Mundo [0.0, 0.0] -> Pixel estimado: (319.3, 239.4) | Pixel real: [320, 240]

Errores de reproyección (pixel -> mundo -> pixel):
Pixel [202, 75] -> Mundo (-0.3500, -0.2500) -> Pixel (202.0, 75.0) | Error: 0.00
Pixel [436, 75] -> Mundo (-0.3506, 0.2492) -> Pixel (436.0, 75.0) | Error: 0.00
Pixel [436, 403] -> Mundo (0.3500, 0.2500) -> Pixel (436.0, 403.0) | Error: 0.00
Pixel [202, 403] -> Mundo (0.3494, -0.2508) -> Pixel (202.0, 403.0) | Error: 0.00
Pixel [320, 240] -> Mundo (0.0012, 0.0016) -> Pixel (320.0, 240.0) | Error: 0.00

Error medio de reproyección: 0.00 pixeles

Test manual: Pixel [342, 403] -> Mundo estimado: (0.349745, 0.048648)
Mundo real: (0.350000, 0.050000)
Error componente: Δx=+0.000255 m, Δy=+0.001352 m
Error euclíadiano total: 0.001375 m (1.38 mm)
```

Figura 6.6 Lectura del terminal de Ubuntu tras la ejecución de `test_homography.py`. Representa los resultados de la validación de la matriz de homografía donde el error en la reproyección es muy bajo. Al comparar la posición real de un cubo adicional (0.35, 0.05) con la posición estimada mediante la matriz (0.3497, 0.04864) se observa un error de 1.38 mm, ampliamente asumible para un robot manipulador.

7 Planificación de agarre y manipulación

El sistema de planificación se basa en el cálculo de la cinemática inversa para determinar las posiciones y orientaciones necesarias del efecto final del robot. Posteriormente, se traspasa esta información a un archivo de configuración donde se ubican el resto de posiciones y el robot procede a ejecutar la tarea de manipulación.

7.1 Cinemática inversa

La cinemática inversa es el proceso de determinar las posiciones articulares necesarias para que el efecto final del robot alcance una posición y orientación deseadas en el espacio cartesiano. En este proyecto, se ha implementado un enfoque basado en la geometría del robot Braccio Tinkerkit, utilizando las longitudes de sus eslabones y las restricciones de sus articulaciones.

7.1.1 Fundamentos teóricos

El sistema utilizado para el cálculo de la cinemática inversa se basa en el marco teórico empleado por Will Stedden [20]. Este enfoque se centra en la geometría del brazo y las limitaciones de sus ángulos articulares mostrados en la Figura 7.1.

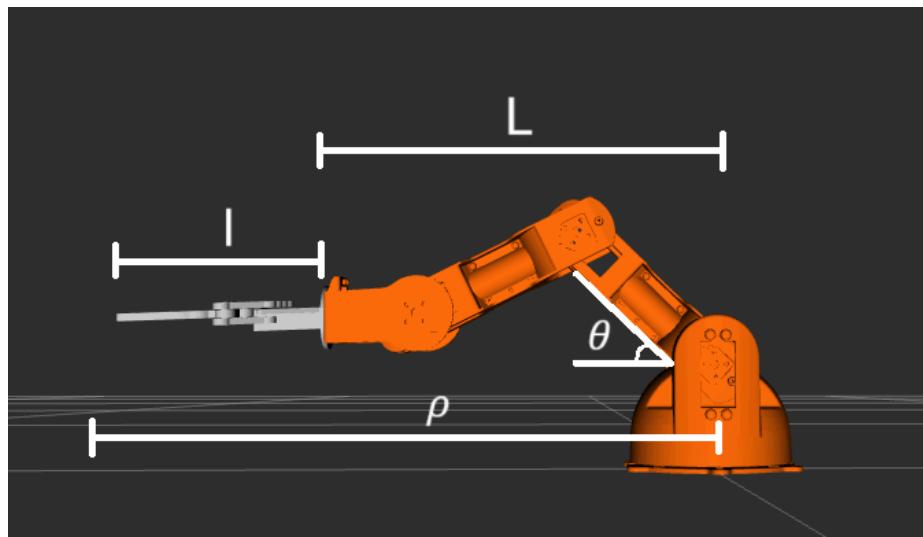


Figura 7.1 Diagrama esquemático del brazo robótico, mostrando las longitudes de los eslabones implicados y las articulaciones relevantes para el cálculo de la cinemática inversa descrito posteriormente.

En primer lugar se ha instanciado la longitud del brazo total en su máxima extensión, siendo ésta $L=0.3025\text{m}$, junto al Offset desde la muñeca hasta el efecto final, que se ha establecido en $l=0.064\text{m}$.

Luego, se transforman las posiciones (x,y) en coordenadas polares, se verifica la altura y se estudia la alcanzabilidad del radio de trabajo, calculado mediante la relación entre las longitudes anteriores y el ángulo máximo y mínimo del hombro en radianes.

$$\rho_{\max} = 0.064 + 0.3025 \cdot \cos(0.27) = 0.356[\text{m}]$$

$$\rho_{\min} = 0.064 + 0.3025 \cdot \cos\left(\frac{\pi}{4}\right) = 0.278[m]$$

«Nota»: el ángulo máximo del codo se establece en 45° para evitar colisiones con la base del robot y el resto de articulaciones, pese a que su rango de funcionamiento real es más amplio $[15^\circ, 165^\circ]$. Este hecho se debe en parte al método de recolección planteado, donde se ha optado por un agarre directo o frontal, frente a un método basado en un agarre desde arriba, donde se cogen objetos desde una vista superior, pero el rango de actuación es más limitado.

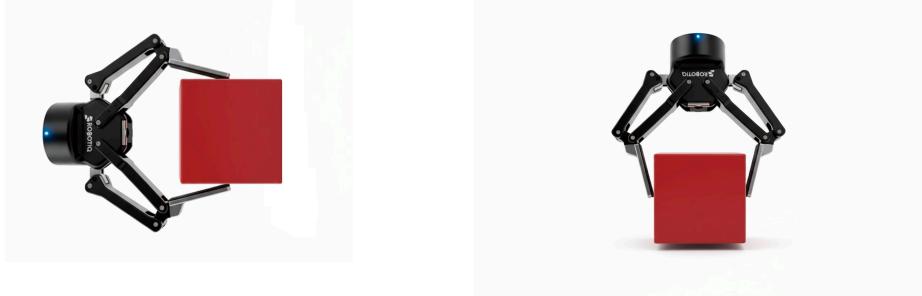


Figura 7.2 Representación de las pinzas del robot, mostrando los dos tipos de agarre. El primero es un agarre en pinza o directo, ideal para objetos de media y larga distancia, mientras que el segundo es un agarre en forma de gancho, ideal para objetos de corta distancia.

Definidos los límites de trabajo, se procede al cálculo de los ángulos articulares (Figura 7.3):

1. Se convierten las coordenadas cartesianas en polares (ρ, φ) y se aplica el cálculo del ángulo del hombro, verificando que se encuentra dentro de los límites establecidos.
2. Se aplica el cálculo del ángulo del codo y muñeca basados en el ángulo del hombro con la finalidad de mantener la orientación vertical.
3. Se devuelven los ángulos articulares calculados en el orden adecuado para el brazo, junto a φ , utilizada para la rotación de la base y θ_{gripper} , que se mantiene constante.

$$\theta_{\text{shoulder}} = \arccos\left(\frac{\rho - l}{L}\right)$$

$$\theta_{\text{wrist}} = \theta_{\text{shoulder}} + \frac{\pi}{2}$$

$$\theta_{\text{elbow}} = \frac{\pi}{2} - 2 \cdot \theta_{\text{shoulder}}$$

$$\text{Ángulos} = [\varphi, \theta_{\text{shoulder}}, \theta_{\text{elbow}}, \theta_{\text{wrist}}, \theta_{\text{gripper}}]$$

Figura 7.3 Ecuaciones para el cálculo de los ángulos articulares de un brazo robótico.

7.1.2 Configuración simétrica

Previo al cálculo final de la cinemática, donde la altura del efecto final es importante de cara a la posición de agarre y aproximación, es necesario aclarar un aspecto clave respecto al ángulo de la base. Según la Tabla 2.3, se establece que el ángulo de la base tiene un rango comprendido entre 0 y 180° . El problema recae cuando, debido a la lógica implementada, el sistema interpreta al obtener las coordenadas polares que dicho ángulo es negativo y, por ende, se encuentra fuera de rango.

La solución propuesta para este problema ha sido implementar una lógica de φ simétrica, de modo que si φ es negativo, el código intenta reflejar el ángulo en la otra cara del robot para que la base pueda alcanzar dicha posición girando hacia el otro lado y «simular» un rango de movimiento de 360° del robot. Es imprescindible explicar y entender esta metodología pues afecta directamente al cálculo de la cinemática inversa y a la planificación de las trayectorias del robot explicadas posteriormente.



Figura 7.4 Representación del problema de la base y su rango de movimiento. En la imagen izquierda se identifica una posición objetivo que se encuentra en el rango $(0, 180^\circ)$. En la derecha, el objeto se encuentra fuera de este rango, lo que implica una solución donde el robot oriente su base como en el primer caso, y aplique simetría en el codo y demás articulaciones del robot.

La metodología implicada es la siguiente:

1. Se obtiene el ángulo φ en coordenadas polares.
2. Si $\varphi < 0$, se refleja el ángulo sumando 180° : $\varphi_{\text{sim}} = \varphi + \pi$.
3. Si $\varphi > 180^\circ$, se refleja restando 180° : $\varphi_{\text{sim}} = \varphi - \pi$.
4. Se normaliza a un rango de $[0, 2\pi]$ si fuese necesario.
5. Se obtienen los ángulos articulares explicados anteriormente, con la única excepción de θ_{shoulder} , el cual se calcula su espejo antes de ser enviado por el vector de ángulos: $\theta_{\text{shoulder}} = \pi - \theta_{\text{shoulder}}$.

7.1.3 Cálculo de las posiciones de aproximación y agarre

Las posiciones se determinan a partir de la cinemática inversa y los ángulos articulares calculados previamente. Sin embargo, éstos se calcularon bajo unos estándares 2D, siendo necesario adaptarlos ahora a un sistema 3D donde existe una altura Z. Se definen así dos posiciones clave:

- Posición de aproximación: Es la posición inicial desde la cual el efecto final se mueve hacia el objeto a manipular. Esta posición se calcula teniendo en cuenta la posición del objeto en el espacio, incluyendo un incremento de altura que establezca un movimiento seguro hacia el objeto, en este caso, 0.03 m.
- Posición de agarre: Es la posición final en la que el efecto final debe estar para realizar la tarea de agarre. Esta posición se determina a partir de la posición del objeto y la configuración del brazo robótico.

El cálculo de estas posiciones se realiza mediante la función `calculate_ik_xyz` en `inverse_kinematics_calculator.py`, que toma como entrada las coordenadas del objeto (x, y, z) y devuelve los ángulos articulares necesarios para alcanzar dichas posiciones. Su finalidad es ajustar principalmente el codo para poder controlar dicha altura efectiva. Para este cálculo, se extraen los ángulos obtenidos previamente y se ajustan según la lógica de simetría descrita anteriormente:

- Configuración estándar: se establece una altura de referencia estándar de $Z_N = 0.05m$ y se configura un factor de sensibilidad dinámico basado en la extensión del hombro:
 - Si el hombro está muy extendido, es decir, es un elemento alejado, $Z_f = 12$.
 - Si el hombro está en una posición intermedia, $Z_f = 10$.
 - Si el hombro está en una posición cercana, $Z_f = 8$.

Es importante destacar que estos valores son aproximados y pueden ajustarse según las necesidades específicas de la tarea y la configuración del brazo robótico.

Con ellos, se calcula la altura ajustada del codo:

$$Z_a = (Z - Z_N) \cdot Z_f$$

$$\theta_{\text{elbow}} = \theta_{\text{elbow}} + Z_a$$

Y se ajusta la posición de la muñeca para mantener la orientación y compensar esta diferencia entre el ángulo original y final del codo:

$$\theta_{\text{wrist}} = \theta_{\text{wrist}} + \Delta\theta_{\text{elbow}} \cdot \frac{1}{2}$$

```
CALCULADORA DE CINEMÁTICA INVERSA PARA BRACCIO
=====
# Objeto en: (0.35, 0.05, 0.025)
[INFO] [1756742662.057610665] [inverse_kinematics_calculator]: === ANALIZANDO ESTRATEGIA ===
[INFO] [1756742662.057796358] [inverse_kinematics_calculator]: === CALCULANDO POSICIONES DE PICK ===
[INFO] [1756742662.058032935] [inverse_kinematics_calculator]: Workspace: rho=0.354m, rango=[0..278, 0..356]m
[INFO] [1756742662.058244045] [inverse_kinematics_calculator]: IK calculado para (0.350, 0.050) - Config: ORIGINAL
[INFO] [1756742662.058447491] [inverse_kinematics_calculator]: Ángulos: ['8.1°', '16.8°', '56.4°', '106.8°', '90.0°']
[INFO] [1756742662.058637372] [inverse_kinematics_calculator]: Shoulder bajo (16.8°) - usando z_factor agresivo: 12.0
[INFO] [1756742662.059000514] [inverse_kinematics_calculator]: Ajuste Z: 0.0000 rad (3.4°) en ELBOW
[INFO] [1756742662.059182330] [inverse_kinematics_calculator]: Ángulos finales: ['8.1°', '16.8°', '59.8°', '108.5°', '90.0°']
[INFO] [1756742662.059364647] [inverse_kinematics_calculator]: Pick approach calculado para Z=0.055m
[INFO] [1756742662.059561681] [inverse_kinematics_calculator]: IK calculado para (0.350, 0.050) - Config: ORIGINAL
[INFO] [1756742662.059746442] [inverse_kinematics_calculator]: Ángulos: ['8.1°', '16.8°', '56.4°', '106.8°', '90.0°']
[INFO] [1756742662.059932897] [inverse_kinematics_calculator]: Shoulder bajo (16.8°) - usando z_factor agresivo: 12.0
[INFO] [1756742662.060117578] [inverse_kinematics_calculator]: IK 3D calculado para (0.350, 0.050, 0.025) - Config: ORIGINAL
[INFO] [1756742662.060298222] [inverse_kinematics_calculator]: Ajuste Z: 0.0050 rad (-17.2°) en ELBOW
[INFO] [1756742662.060490467] [inverse_kinematics_calculator]: Ángulos finales: ['8.1°', '16.8°', '39.2°', '98.2°', '90.0°']
[INFO] [1756742662.060673996] [inverse_kinematics_calculator]: Pick position calculado para Z=0.025m (objeto en 0.025m + 5mm de seguridad)
[INFO] [1756742662.060855372] [inverse_kinematics_calculator]: === CÁLCULOS DE PICK COMPLETADOS ===
[INFO] [1756742662.061034819] [inverse_kinematics_calculator]: Diferencia de altura: 0.030m entre approach y position
[INFO] [1756742662.061212441] [inverse_kinematics_calculator]: Approach Z: 0.055m | Pick Z: 0.025m | Objeto Z: 0.025m
[INFO] [1756742662.061423912] [inverse_kinematics_calculator]: Estrategia: PICK DIRECTO

=====
POSICIONES CALCULADAS:
# Formato: [radianes] | [grados]
pick_approach: [0.1419, 0.2936, 1.0435, 1.8944, 1.5708] | [8.1, 16.8, 59.8, 108.5, 90.0]
pick_position: [0.1419, 0.2936, 0.6835, 1.7144, 1.5708] | [8.1, 16.8, 39.2, 98.2, 90.0]
```

Figura 7.5 Lectura del terminal de Ubuntu. Muestra el cálculo de la cinemática inversa para una posición objetivo x=0.35, y=0.05 y z=0.025, donde se observa el procedimiento ejecutado hasta el cálculo final, mostrando finalmente la posición de agarre y aproximación en radianes y grados.

- Configuración simétrica: para este caso, aplicar los mismos conceptos, así como la simetría es un proceso complejo puesto que se debe forzar al efecto a bajar, compensando que la cadena cinemática se alarga. Por ello, se establecen dos estrategias:
 - Alturas muy bajas: se aumenta el ángulo del hombro, se alinea el codo y se baja la muñeca drásticamente para que el efecto final pueda bajar lo máximo posible.
 - Alturas moderadas: el efecto es similar a la configuración estándar. Se reduce el ángulo del codo moderadamente y se compensa éste con la muñeca para mantener la orientación.

```
CALCULADORA DE CINEMÁTICA INVERSA PARA BRACCIO
=====
# Objeto en: (0.28, -0.15, 0.025)
[INFO] [1756743878.253141140] [inverse_kinematics_calculator]: === ANALIZANDO ESTRATEGIA ===
[INFO] [1756743878.253338163] [inverse_kinematics_calculator]: === CALCULANDO POSICIONES DE PICK ===
[INFO] [1756743878.253592344] [inverse_kinematics_calculator]: Workspace: rho=0.318m, rango=[-0.278, 0.356]m
[INFO] [1756743878.253806860] [inverse_kinematics_calculator]: Configuración simétrica: φ -28.2° → 151.8°
[INFO] [1756743878.254004725] [inverse_kinematics_calculator]: Configuración simétrica: Invertiendo SOLO base y shoulder...
[INFO] [1756743878.254186889] [inverse_kinematics_calculator]: Configuración simétrica aplicada:
[INFO] [1756743878.254374484] [inverse_kinematics_calculator]: Base: -28.2° → 151.8°
[INFO] [1756743878.254559054] [inverse_kinematics_calculator]: Shoulder: 33.0° → 147.0°
[INFO] [1756743878.254738363] [inverse_kinematics_calculator]: Elbow: SIN CAMBIOS (123.0°)
[INFO] [1756743878.254915408] [inverse_kinematics_calculator]: Wrist: SIN CAMBIOS (123.0°)
[INFO] [1756743878.255093375] [inverse_kinematics_calculator]: Gripper: SIN CAMBIOS (0.0°)
[INFO] [1756743878.255272184] [inverse_kinematics_calculator]: IK calculado para (0.280, -0.150) - Config: SIMÉTRICA
[INFO] [1756743878.255467373] [inverse_kinematics_calculator]: Ángulos: ['151.8°', '147.0°', '24.0°', '123.0°', '90.0°']
[INFO] [1756743878.255664958] [inverse_kinematics_calculator]: Configuración simétrica detectada - ajuste dinámico basado en Z=0.055m
[INFO] [1756743878.255838335] [inverse_kinematics_calculator]: Altura baja (0.055m) - aumentando shoulder AGRESIVAMENTE en 5.7°
[INFO] [1756743878.256020921] [inverse_kinematics_calculator]: Alineando elbow con shoulder: factor=0.06
[INFO] [1756743878.256200120] [inverse_kinematics_calculator]: Elbow alineado: 24.0° → 31.7° (similar a shoulder 152.7°)
[INFO] [1756743878.256387024] [inverse_kinematics_calculator]: Wrist bajado extra 28.6° para mejor agarre
[INFO] [1756743878.256567005] [inverse_kinematics_calculator]: Z=0.055m, ref=0.080m, diff=-0.025m
[INFO] [1756743878.256766212] [inverse_kinematics_calculator]: Shoulder: 147.0° → 152.7° (ajuste: 5.7°)
[INFO] [1756743878.256966280] [inverse_kinematics_calculator]: Elbow reducido en 93.1° (base=57.3° + altura)
[INFO] [1756743878.257100588] [inverse_kinematics_calculator]: Elbow: 24.0° → 31.7°
[INFO] [1756743878.257334658] [inverse_kinematics_calculator]: IK 3D calculado para (0.288, -0.150, 0.055) - Config: SIMÉTRICA
[INFO] [1756743878.257523215] [inverse_kinematics_calculator]: Ajuste Z: -0.1348 rad (-7.7°) en ELBOW
[INFO] [1756743878.257716991] [inverse_kinematics_calculator]: Ángulos finales: ['151.8°', '152.7°', '31.7°', '155.1°', '90.0°']
[INFO] [1756743878.257901631] [inverse_kinematics_calculator]: Pick approach calculado para Z=0.055m
[INFO] [1756743878.259618290] [inverse_kinematics_calculator]: IK calculado para (0.280, -0.150) - Config: SIMÉTRICA
[INFO] [1756743878.259963951] [inverse_kinematics_calculator]: Ángulos: ['151.8°', '147.0°', '24.0°', '123.0°', '90.0°']
[INFO] [1756743878.261700772] [inverse_kinematics_calculator]: Configuración simétrica detectada - ajuste dinámico basado en Z=0.025m
[INFO] [1756743878.261894489] [inverse_kinematics_calculator]: Altura baja (0.025m) - aumentando shoulder AGRESIVAMENTE en 40.1°
[INFO] [1756743878.262076563] [inverse_kinematics_calculator]: Alineando elbow con shoulder: factor=0.42
[INFO] [1756743878.262256784] [inverse_kinematics_calculator]: Elbow alineado: 24.0° → 87.1° (similar a shoulder 174.3°)
[INFO] [1756743878.262441294] [inverse_kinematics_calculator]: Wrist bajado extra 28.6° para mejor agarre
[INFO] [1756743878.262621164] [inverse_kinematics_calculator]: Z=0.025m, ref=0.080m, diff=-0.055m
[INFO] [1756743878.262800473] [inverse_kinematics_calculator]: Shoulder: 147.0° → 147.4° (ajuste: 40.1°)
[INFO] [1756743878.262979432] [inverse_kinematics_calculator]: Elbow reducido en 114.6° (base=57.3° + altura)
[INFO] [1756743878.263157459] [inverse_kinematics_calculator]: Elbow: 24.0° → 87.1°
[INFO] [1756743878.263341497] [inverse_kinematics_calculator]: IK 3D calculado para (0.288, -0.150, 0.025) - Config: SIMÉTRICA
[INFO] [1756743878.263524343] [inverse_kinematics_calculator]: Ajuste Z: -1.1018 rad (-63.1°) en ELBOW
[INFO] [1756743878.263705606] [inverse_kinematics_calculator]: Ángulos finales: ['151.8°', '174.3°', '87.1°', '176.1°', '90.0°']
[INFO] [1756743878.263883383] [inverse_kinematics_calculator]: Pick position calculado para Z=0.025m (objeto en 0.025m + 5mm de seguridad)
[INFO] [1756743878.264062141] [inverse_kinematics_calculator]: === CÁLCULOS DE PICK COMPLETADOS ===
```

Figura 7.6 Lectura del terminal de Ubuntu. Muestra el cálculo de la cinemática inversa para una posición objetivo simétrica: x=0.28, y=-0.15 y z=0.025, donde se observa el procedimiento ejecutado hasta el cálculo final. La posición de agarre se establece en [151.8, 174.3, 87.1, 176.1, 90.0][°], siendo la posición de aproximación [151.8, 147.0, 24.0, 123.0, 90.0][°].

Los cálculos de esta metodología se basan principalmente en constantes empíricas obtenidas tras múltiples pruebas y errores, las cuales pueden ser ajustadas según las necesidades específicas de la tarea y la configuración del brazo robótico. Por ello, no se detallan las ecuaciones específicas en este documento. Sin embargo, en el repositorio [17] se pueden visualizar cada una de ellas en el archivo correspondiente.

Las posiciones calculadas mediante la cinemática inversa se almacenan en variables específicas para la aproximación y el agarre. Posteriormente, se abre el archivo *pick_and_place_config.yaml*, donde se encuentran las posiciones de home, reposo, depósito y objetivo. Entonces, se actualizan las entradas correspondientes a las posiciones objetivo del robot, sobre escribiendo los valores previos con los nuevos ángulos articulares obtenidos. Este proceso garantiza que el sistema de planificación y ejecución utilice siempre las posiciones más precisas y actualizadas para cada tarea de pick-and-place.

7.1.4 Test cinemática inversa

En la misma carpeta donde se ha desarrollado la cinemática inversa se encuentra un script de validación llamado *ik_workspace_tester_py*. Este script permite verificar la correcta implementación de la cinemática inversa, haciendo uso de un rango ampliado de posiciones en los ejes XY y comprobando que el robot puede alcanzar dichas posiciones sin errores. La ejecución del script proporciona información sobre las posiciones probadas, junto a los resultados obtenidos, indicando si es apto, demasiado cerca/lejos o el problema encontrado. Junto a este, proporciona un porcentaje de éxito en las pruebas realizadas.

La base del uso de este script se basa en encontrar posiciones de trabajo adecuadas para el robot, asegurando que pueda operar siempre en las mejores posiciones posibles, donde se minimicen los errores de posicionamiento y se maximice la eficiencia en las tareas de pick-and-place.

```
== TESTER DE WORKSPACE DE PICK DIRECTO ==
x (m)   y (m)   Resultado           Ángulos (grados)
0.210   0.150   Demasiado cerca (theta_shoulder > THETA_RET)
0.210   0.160   Demasiado cerca (theta_shoulder > THETA_RET)
0.210   0.170   Demasiado cerca (theta_shoulder > THETA_RET)
0.210   0.180   Demasiado cerca (theta_shoulder > THETA_RET)
0.210   0.190   PICK DIRECTO      [42.1, 43.6, 2.9, 133.6, 90.0]
0.210   0.200   PICK DIRECTO      [43.6, 41.7, 6.7, 131.7, 90.0]
0.210   0.210   PICK DIRECTO      [45.0, 39.6, 10.7, 129.6, 90.0]
0.210   0.220   PICK DIRECTO      [46.3, 37.5, 15.1, 127.5, 90.0]
0.210   0.230   PICK DIRECTO      [47.6, 35.1, 19.8, 125.1, 90.0]
0.210   0.240   PICK DIRECTO      [48.8, 32.6, 24.8, 122.6, 90.0]
0.210   0.250   PICK DIRECTO      [50.0, 29.8, 30.4, 119.8, 90.0]
0.210   0.260   PICK DIRECTO      [51.1, 26.7, 36.6, 116.7, 90.0]
0.210   0.270   PICK DIRECTO      [52.1, 23.2, 43.6, 113.2, 90.0]
0.210   0.280   PICK DIRECTO      [53.1, 19.0, 52.0, 109.0, 90.0]
0.210   0.290   Demasiado lejos (theta_shoulder < THETA_EXT)
0.220   0.150   Demasiado cerca (theta_shoulder > THETA_RET)
0.220   0.160   Demasiado cerca (theta_shoulder > THETA_RET)
0.220   0.170   PICK DIRECTO      [37.7, 45.0, 0.1, 135.0, 90.0]
0.220   0.180   PICK DIRECTO      [39.3, 43.3, 3.5, 133.3, 90.0]
0.220   0.190   PICK DIRECTO      [40.8, 41.5, 7.1, 131.5, 90.0]
0.220   0.200   PICK DIRECTO      [42.3, 39.5, 10.9, 129.5, 90.0]
0.220   0.210   PICK DIRECTO      [43.7, 37.5, 15.1, 127.5, 90.0]
0.220   0.220   PICK DIRECTO      [45.0, 35.2, 19.6, 125.2, 90.0]
0.220   0.230   PICK DIRECTO      [46.3, 32.8, 24.4, 122.8, 90.0]
0.220   0.240   PICK DIRECTO      [47.5, 30.1, 29.7, 120.1, 90.0]
0.220   0.250   PICK DIRECTO      [48.7, 27.2, 35.6, 117.2, 90.0]
0.220   0.260   PICK DIRECTO      [49.8, 23.9, 42.2, 113.9, 90.0]
0.220   0.270   PICK DIRECTO      [50.8, 20.0, 50.0, 110.0, 90.0]
0.220   0.280   Demasiado lejos (theta_shoulder < THETA_EXT)
0.220   0.290   Demasiado lejos (theta_shoulder < THETA_EXT)
0.230   0.150   Demasiado cerca (theta_shoulder > THETA_RET)
```

Figura 7.7 Lectura del terminal de Ubuntu tras la ejecución del script *ik_workspace_tester_py*. Muestra el resultado de las pruebas de cinemática inversa, indicando si las posiciones son aptas o si presentan problemas. En este fragmento, se muestra un rango de posición X comprendido entre 0.21 y 0.23, mientras que el rango Y se encuentra entre 0.15 y 0.30, con un paso de 0.02, mostrando así las posiciones cómodas para la ejecución del robot en un rango determinado.

7.2 Repositorio attach/detach

Durante todo el desarrollo del proyecto se ha hablado sobre la importancia de la lógica de pick-and-place, donde el robot debe ser capaz de identificar, agarrar y mover objetos dentro del entorno simulado. Sin embargo, el sistema de simulación de agarre del robot Braccio Tinkerkit no es nativo de Gazebo, por lo que no se puede simular el agarre de objetos directamente.

En versiones anteriores de ROS y Gazebo, se utilizaba el plugging de Planning Scene de MoveIt para gestionar la colisión y el contacto entre el robot y los objetos en la escena [23]. No obstante, este plugin no es compatible con ROS 2 y Gazebo, lo que ha llevado a la búsqueda de alternativas para implementar esta funcionalidad en el entorno actual, puesto que la web oficial de MoveIt2 aún no ha implementado una solución. Como consecuencia, tras un intento fallido de adaptación del plugin, se ha optado por indagar en la red proyectos similares que puedan ofrecer una solución viable.

Ante ello, la solución adoptada ha sido emplear el plugin *libgazebo_link_attacher.so*, implementado por la universidad de Cranfield [24]. Este plugin permite simular el agarre y la liberación de objetos en el entorno de Gazebo, a través de la instalación del mismo y mediante un sencillo sistema de envío y recepción de mensajes.

La interfaz proporcionada gestiona las colisiones y los contactos entre el robot y los objetos, por medio de una llamada al servicio /ATTACHLINK y /DETACHLINK. Estos servicios únicamente requieren del *model_name* y *link_name* de ambos elementos a unir o separar, es decir, el robot y el objeto a manipular. El problema de esto ha recaído en que, pese a estar muy bien explicado en su repositorio, no se ha mostrado una vía clara para un sistema donde el objeto a manipular cambia dinámicamente, como es el caso de este proyecto. Es por ello que, tal como se explicó anteriormente en la Sección 5.3, ha sido necesario implementar un sistema dinámico basado en la distancia entre los elementos implicados.

```
[python3-2] [INFO] [1756213044.56351676] [gamepad_teleop]: Modelos en model_states: ['ground_plane', 'zona_hexagono', 'zona_hexagono_azul', 'braccio', 'overhead_camera', 'corner1', 'corner2', 'corner3', 'corner4', 'green_cube1', 'green_cube2', 'blue_cube1', 'blue_cube2']
[python3-2] [INFO] [1756213044.56445267] [gamepad_teleop]: Comparando con modelos: zona_hexagono_azul, posición: (0.0, -0.09159123305363656, 0.15466432781284215)
[python3-2] [INFO] [1756213044.56445267] [gamepad_teleop]: Comparando con modelos: zona_hexagono, posición: (0.0, 2.7755575615628914e-17, 0.0)
[python3-2] [INFO] [1756213044.564718761] [gamepad_teleop]: Distancia a zona_hexagono: 0.25877253615629791
[python3-2] [INFO] [1756213044.564973592] [gamepad_teleop]: Comparando con modelo: zona_hexagono_azul, posición: (2.7755575615628914e-17, -4.163336342344337e-17, 0.0)
[python3-2] [INFO] [1756213044.565225057] [gamepad_teleop]: Distancia a zona_hexagono_azul: 0.25877253615629793
[python3-2] [INFO] [1756213044.565565575] [gamepad_teleop]: Comparando con modelo: overhead_camera, posición: (0.0, 0.0, 1.2)
[python3-2] [INFO] [1756213044.565736650] [gamepad_teleop]: Distancia a overhead_camera: 1.0657245694366736
[python3-2] [INFO] [1756213044.565962596] [gamepad_teleop]: Comparando con modelo: corner1, posición: (-0.35, -0.25, 0.014999999999755)
[python3-2] [INFO] [1756213044.566185968] [gamepad_teleop]: Distancia a corner1: 0.5762407365629631
[python3-2] [INFO] [1756213044.566410542] [gamepad_teleop]: Comparando con modelo: corner2, posición: (0.35, -0.25, 0.014999999999755)
[python3-2] [INFO] [1756213044.566845256] [gamepad_teleop]: Comparando con modelo: corner3, posición: (0.35, 0.25, 0.014999999999755)
[python3-2] [INFO] [1756213044.567043424] [gamepad_teleop]: Distancia a corner3: 0.4037775251583654
[python3-2] [INFO] [1756213044.567245515] [gamepad_teleop]: Comparando con modelo: corner4, posición: (-0.35, 0.25, 0.014999999999755)
[python3-2] [INFO] [1756213044.567451724] [gamepad_teleop]: Distancia a corner4: 0.6508851929895944
[python3-2] [INFO] [1756213044.567653063] [gamepad_teleop]: Comparando con modelo: green_cube1, posición: (0.35, 0.04999999999999996, 0.01499999999999755)
[python3-2] [INFO] [1756213044.567850976] [gamepad_teleop]: Distancia a green_cube1: 0.2576815799547396
[python3-2] [INFO] [1756213044.568039271] [gamepad_teleop]: Comparando con modelo: green_cube2, posición: (0.28, 0.18, 0.01499999999999755)
[python3-2] [INFO] [1756213044.568223449] [gamepad_teleop]: Distancia a green_cube2: 0.319491849192509
[python3-2] [INFO] [1756213044.568410112] [gamepad_teleop]: Comparando con modelo: blue_cube1, posición: (0.2885912169297311, -0.10961402782969638, 0.15985862336583423)
[python3-2] [INFO] [1756213044.568607885] [gamepad_teleop]: Distancia a blue_cube1: 0.1414046889385649
[python3-2] [INFO] [1756213044.568763795] [gamepad_teleop]: Comparando con modelos blue_cube1, blue_cube2, posición: (0.23976770213997542, -0.24903742142327173, 0.014999274085567619)
[python3-2] [INFO] [1756213044.568964866] [gamepad_teleop]: Distancia a blue_cube2: 0.21718675314063462
[python3-2] [INFO] [1756213044.569155445] [gamepad_teleop]: Modelo más cercano: blue_cube1, distancia: 0.10414046889385649
[python3-2] [INFO] [1756213044.569458617] [gamepad_teleop]: Modelo más cercano para detach: blue_cube1, link: link
[python3-2] [INFO] [1756213044.569895130] [gamepad_teleop]: ⚡ Intentando detach: blue_cube1/link
```

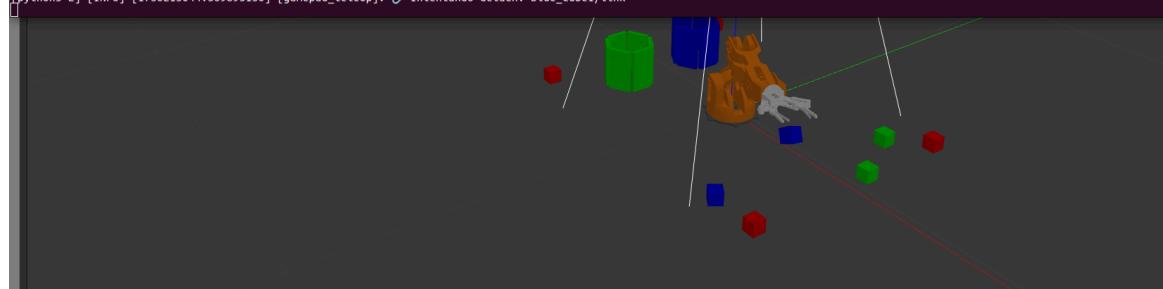


Figura 7.8 Lectura del terminal de Ubuntu. Representa la petición de detach al servicio, el cálculo de las distancias respecto la posición del gripper y los cubos; y la ejecución de la acción de detach, donde se puede mostrar el cubo azul cayendo desde la pinza.

En la Figura 5.3 mostrada anteriormente, se puede observar la acción inversa, el proceso de coger el cubo.

7.3 Flujo de acción

El flujo de acción del sistema de pick-and-place se basa en la integración de los subsistemas de percepción, planificación y control. A continuación, se detalla el proceso completo desde la detección del objeto hasta la ejecución de la tarea, descrito por el script `vision_auto_pick_and_place.py`:

- El nodo arranca y se suscribe a los tópicos de detección de objetos y estados del robot. Carga la calibración de la cámara obtenida mediante homografía y crea los clientes encargados de la lógica de *attach* y *detach*.
- Recibe un *PointStamped* con las coordenadas del objeto a manipular. Se realiza un filtrado de detecciones ya procesadas para evitar repeticiones y se encola o procesa, según corresponda.
- Transforma las coordenadas del objeto (px) a coordenadas del mundo (m) mediante la matriz de homografía obtenida de `camera_calibration.json`.
- Llama a la lógica de la cinemática inversa para calcular los ángulos articulares necesarios para alcanzar la posición de agarre y aproximación del objeto, e importarlos en `pick_and_place_config.yaml`.
- El ejecutor carga las posiciones desde el archivo de configuración, planifica con Moveit2 y ejecuta la trayectoria.
- Se monitorea el estado del robot y se gestionan las colisiones y contactos mediante los servicios /ATTACHLINK y /DETACHLINK en base al cierre y apertura de la pinza.
- Finalmente, se retorna a la posición de home, marca el objeto como procesado y se espera la siguiente detección.

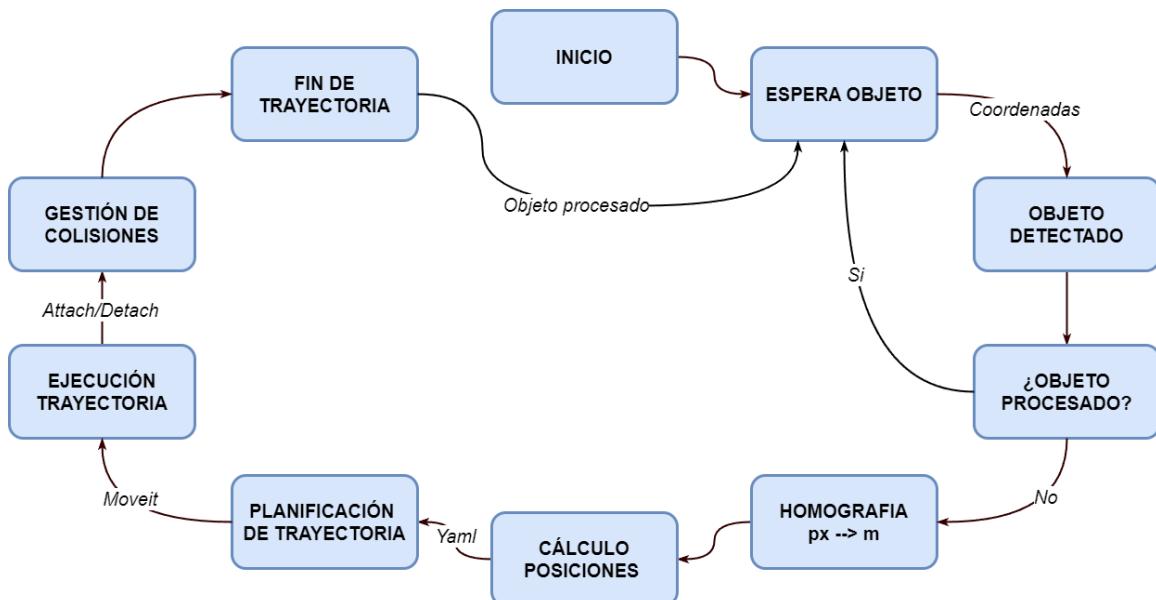


Figura 7.9 Diagrama del flujo de acción del sistema de pick-and-place, mostrando la interacción entre los subsistemas de percepción, planificación y control.

En la Figura 7.10 se muestra la ejecución del sistema de pick-and-place para la recolección de un cubo, unificando todos los subsistemas y etapas del proceso explicadas. En las siguientes, se detallan imágenes del proceso de ejecución del robot, mostrando las posiciones alcanzadas en cada etapa del proceso.

```
Victus-by-HP-Ivan:~/Escritorio/Braccio-Tinkerkit-Arduino$ ros2 launch braccio_vision vision_auto_pick_and_place.launch.py
[INFO] [launch]: All log files can be found below /home/ivan/.ros/log/2025-09-01-18-16-25-472345-Victus-by-HP-Ivan-7793
[INFO] [launch]: Default logging verbosity is set to INFO
[INFO] [vision_auto_pick_and_place.py-1]: process started with pid [7795]
[vision_auto_pick_and_place.py-1] [WARN] [1756743386.101930450] [rcl.logging.rosout]: Publisher already registered for provided node name. If this is due to multiple nodes with the same name then all logs for that logger name will go out over the existing publisher. As soon as any node with that name is destroyed it will unregister the publisher, preventing any further logs for that name from being published on the rosout topic.
[vision_auto_pick_and_place.py-1] [INFO] [1756743386.125437090] [vision_auto_pick_and_place]: Configuración cargada desde: /home/ivan/Escritorio/Braccio-Tinkerkit-Arduino/braccio_moveit_config/config/pick_and_place_config.yaml
[vision_auto_pick_and_place.py-1] [INFO] [1756743386.149377838] [vision_auto_pick_and_place]: Nodo Configurable Pick and Place iniciado
[vision_auto_pick_and_place.py-1] [INFO] [1756743386.149662895] [vision_auto_pick_and_place]: Configuración cargada: 3 secuencias disponibles
[vision_auto_pick_and_place.py-1] [INFO] [1756743386.151532208] [vision_auto_pick_and_place]: ✓ Homografía cargada exitosamente
[vision_auto_pick_and_place.py-1] [INFO] [1756743386.151768325] [vision_auto_pick_and_place]: Matriz de homografía: [[-2.6259353610373147e-08, 0.0021242396000179836, -0.508588299906841], [0.002126805442632593, -1.039191438735743e-08, -0.6790313491379903], [-6.73234616623402e-06, -1.423180536143241e-05, 1.0]]
[vision_auto_pick_and_place.py-1] [INFO] [1756743386.151999631] [vision_auto_pick_and_place]: Vision-Based Pick and Place iniciado
[vision_auto_pick_and_place.py-1] [INFO] [1756743386.152207794] [vision_auto_pick_and_place]: Esperando detección de cubos verdes...
[vision_auto_pick_and_place.py-1] [INFO] [1756743396.153320841] [vision_auto_pick_and_place]: ESTADÍSTICAS: Ningún objeto procesado aún
[vision_auto_pick_and_place.py-1] [INFO] [1756743399.993677684] [vision_auto_pick_and_place]: RECIBIDO del object_detector: pixel(342, 403)
[vision_auto_pick_and_place.py-1] [INFO] [1756743399.993426161] [vision_auto_pick_and_place]: TRANSFORMADO: pixel(342,403) usando HOMOGRAFIA
[vision_auto_pick_and_place.py-1] [INFO] [1756743399.994824169] [vision_auto_pick_and_place]: Homografía: pixel(342,403) > mundo(0.358,0.049) rho=0.353m
[vision_auto_pick_and_place.py-1] [INFO] [1756743399.994288062] [vision_auto_pick_and_place]: Coordenadas detectadas (0.358, 0.049) > Objeto real: green_cube1 (distancia: 0.001m)
[vision_auto_pick_and_place.py-1] [INFO] [1756743399.994615193] [vision_auto_pick_and_place]: Nuevo cubo verde detectado: green_cube1 en pixeles (342, 403)
[vision_auto_pick_and_place.py-1] [INFO] [1756743399.994881756] [vision_auto_pick_and_place]: Total de cubos en lista: 1
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.089771713] [vision_auto_pick_and_place]: RECIBIDO del object_detector: pixel(403, 370)
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.090181847] [vision_auto_pick_and_place]: TRANSFORMADO: pixel(403,370) usando HOMOGRAFIA
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.090687050] [vision_auto_pick_and_place]: Homografía: pixel(403,370) > mundo(0.279,0.179) rho=0.332m
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.091058180] [vision_auto_pick_and_place]: Coordenadas detectadas (0.279, 0.179) > Objeto real: green_cube2 (distancia: 0.001m)
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.091299795] [vision_auto_pick_and_place]: Nuevo cubo verde detectado: green_cube2 en pixeles (403, 370)
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.153696260] [vision_auto_pick_and_place]: Total de cubos en lista: 2
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.154031693] [vision_auto_pick_and_place]: Procesando cubo verde green_cube1 en posición: (342, 403)
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.154307332] [vision_auto_pick_and_place]: Quedan 1 cubos en cola
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.154522548] [vision_auto_pick_and_place]: Procesando objeto green_cube1...
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.154800853] [vision_auto_pick_and_place]: TRANSFORMADO: pixel(342,403) usando HOMOGRAFIA
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.155094757] [vision_auto_pick_and_place]: Homografía: pixel(342,403) > mundo(0.350,0.049) rho=0.353m
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.155395034] [vision_auto_pick_and_place]: Objeto green_cube1 localizado en: (0.350, 0.049, 0.025)
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.155618765] [vision_auto_pick_and_place]: Coordenadas para IK: (0.350, 0.049, 0.025)
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.160846665] [vision_auto_pick_and_place]: Calculando cinemática inversa...
For this logger name will go out over the existing publisher. As soon as any node with that name is destroyed it will unregister the publisher, preventing any further logs for that name from being published on the rosout topic.
[vision_auto_pick_and_place.py-1] [WARN] [1756743408.160846665] [rcl.logging.rosout]: Publisher already registered for provided node name. If this is due to multiple nodes with the same name then all logs for that logger name will go out over the existing publisher. As soon as any node with that name is destroyed it will unregister the publisher, preventing any further logs for that name from being published on the rosout topic.
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.173057510] [vision_auto_pick_and_place]: Calculadora de IK analítica para Braccio inclinada
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.173382433] [vision_auto_pick_and_place]: Parametros: L=0.3025m, l=0.064m
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.173632805] [vision_auto_pick_and_place]: Workspace: rho=0.353m, rango=[0.278, 0.356]m
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.173829226] [vision_auto_pick_and_place]: --- CALCULANDO POSICIONES DE PICK ---
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.174622330] [vision_auto_pick_and_place]: Workspace: rho=0.353m, rango=[0.278, 0.356]m
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.174258860] [vision_auto_pick_and_place]: IK calculado para (0.350, 0.049) - Config: ORIGINAL
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.174508219] [vision_auto_pick_and_place]: Angulos: ['7.9°', '11.1°', '55.8°', '101.1°', '98.0°']
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.174773576] [vision_auto_pick_and_place]: Shoulder bajo (17.1°) - usando z_factor agresivo: 12.0
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.175036091] [vision_auto_pick_and_place]: IK 3D calculado para (0.350, 0.049, 0.105) - Config: ORIGINAL
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.17525188] [vision_auto_pick_and_place]: Ajuste Z: 0.6000 rad (37.8°) en ELBOW
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.175506089] [vision_auto_pick_and_place]: Angulos finales: ['7.9°', '17.1°', '93.6°', '120.0°', '90.0°']
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.175760529] [vision_auto_pick_and_place]: Plick approach calculado para Z=0.105m
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.176034684] [vision_auto_pick_and_place]: IK calculado para (0.350, 0.049) - Config: ORIGINAL
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.176234430] [vision_auto_pick_and_place]: Angulos: ['7.9°', '11.1°', '55.8°', '101.1°', '98.0°']
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.176660000] [vision_auto_pick_and_place]: Shoulder bajo (17.1°) usando z_factor agresivo: 12.0
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.176723340] [vision_auto_pick_and_place]: IK 3D calculado para (0.350, 0.049, 0.025) - Config: ORIGINAL
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.177474688] [vision_auto_pick_and_place]: Ajuste Z: 0.30008 rad (-17.2°) en ELBOW
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.177484340] [vision_auto_pick_and_place]: Angulos finales: ['7.9°', '17.1°', '58.6°', '98.5°', '90.0°']
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.177651363] [vision_auto_pick_and_place]: Pick position calculado para Z=0.025m (objeto en 0.025m + 5mm de seguridad)
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.178267044] [vision_auto_pick_and_place]: --- CÁLCULOS DE PICK COMPLETADOS --=
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.178479795] [vision_auto_pick_and_place]: Diferencia de altura: 0.080m entre approach y position
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.178684511] [vision_auto_pick_and_place]: Approach Z: 0.105m | Pick Z: 0.025m | Objeto Z: 0.025m
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.179155400] [vision_auto_pick_and_place]: DIAGNÓSTICO - Posiciones: IK calculadas para (0.350, 0.049, 0.025):
    • Pick approach: [0.1382, 0.2986, 1.6336, 2.1994, 1.5708]
    • Pick position: [0.1382, 0.2986, 0.0736, 1.7194, 1.5708]
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.179776731] [vision_auto_pick_and_place]: Recargando configuración actualizada...
[vision_auto_pick_and_place.py-1] [INFO] [1756743408.695628754] [vision_auto_pick_and_place]: Configuración actualizada en: /home/ivan/Escritorio/Braccio-Tinkerkit-Arduino/braccio_moveit_config/config/pick_and_place_config.yaml
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.695907070] [vision_auto_pick_and_place]: Cinemática inversa calculada y configuración guardada exitosamente
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.696143656] [vision_auto_pick_and_place]: Esperando estabilización del archivo de configuración...
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.698859696] [vision_auto_pick_and_place]: Iniciando secuencia pick and place para green_cube1...
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.699076733] [vision_auto_pick_and_place]: Usando secuencia para cubos verdes >destino: (0.25, 0.10)
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.699270883] [vision_auto_pick_and_place]: Iniciando pick and place para: green_cube1
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.699475347] [vision_auto_pick_and_place]: Recargando configuración actualizada...
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.707167312] [vision_auto_pick_and_place]: Configuración cargada desde: /home/ivan/Escritorio/Braccio-Tinkerkit-Arduino/braccio_moveit_config/config/pick_and_place_config.yaml
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.707487046] [vision_auto_pick_and_place]: === EJECUTANDO SECUENCIA: GREEN_CUBE_SEQUENCE PARA green_cube1 ===
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.707755692] [vision_auto_pick_and_place]: Paso 1/8
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.707904465] [vision_auto_pick_and_place]: Ejecutando: Posición inicial (objetivo: green_cube1)
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.708306480] [vision_auto_pick_and_place]: Moviendo a: home
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.708570839] [vision_auto_pick_and_place]: Posiciones: [0.0, 1.57, 0.0, 0.0, 0.0]
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.712523148] [vision_auto_pick_and_place]: Paso 2/8
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.712926048] [vision_auto_pick_and_place]: Ejecutando: Acerca al cubo verde (objetivo: green_cube1)
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.713472578] [vision_auto_pick_and_place]: Moviendo a: pick_approach
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.713748569] [vision_auto_pick_and_place]: Posiciones: [0.13820937553898294, 0.2986167548516211, 1.6335628170916545, 2.199413081646518, 1.570796326794896]
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.715234148] [vision_auto_pick_and_place]: Paso 3/8
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.717788684] [vision_auto_pick_and_place]: Ejecutando: Bajar al cubo verde (objetivo: green_cube1)
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.719254300] [vision_auto_pick_and_place]: Moviendo a: pick_position
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.719637673] [vision_auto_pick_and_place]: Posiciones: [0.13820937553898294, 0.2986167548516211, 1.6335628170916545, 1.7194130816465178, 1.570796326794896]
[vision_auto_pick_and_place.py-1] [INFO] [1756743409.719903594] [vision_auto_pick_and_place]: Recargando configuración actualizada...
[vision_auto_pick_and_place.py-1] [INFO] [1756743413.725349644] [vision_auto_pick_and_place]: Configuración cargada desde: /home/ivan/Escritorio/Braccio-Tinkerkit-Arduino/braccio_moveit_config/config/pick_and_place_config.yaml
[vision_auto_pick_and_place.py-1] [INFO] [1756743413.725696929] [vision_auto_pick_and_place]: Pasos 4/8
[vision_auto_pick_and_place.py-1] [INFO] [1756743413.726601728] [vision_auto_pick_and_place]: Ejecutando: Agarrar cubo verde (objetivo: green_cube1)
[vision_auto_pick_and_place.py-1] [INFO] [1756743417.721036600] [vision_auto_pick_and_place]: Cerrando gripper a posición: 0.75
[vision_auto_pick_and_place.py-1] [INFO] [1756743417.721016600] [vision_auto_pick_and_place]: ATTACHLINK exitoso para modelo: green_cube1
[vision_auto_pick_and_place.py-1] [INFO] [1756743417.721570103] [vision_auto_pick_and_place]: Paso 5/8
[vision_auto_pick_and_place.py-1] [INFO] [1756743417.725120151] [vision_auto_pick_and_place]: Ejecutando: Levantar cubo verde (objetivo: green_cube1)
[vision_auto_pick_and_place.py-1] [INFO] [1756743417.728415151] [vision_auto_pick_and_place]: Moviendo a: pick_approach
[vision_auto_pick_and_place.py-1] [INFO] [1756743417.730866549] [vision_auto_pick_and_place]: Posiciones: [0.13820937553898294, 0.2986167548516211, 1.6335628170916545, 2.199413081646518, 1.570796326794896]
[vision_auto_pick_and_place.py-1] [INFO] [1756743421.732711739] [vision_auto_pick_and_place]: Paso 6/8
[vision_auto_pick_and_place.py-1] [INFO] [1756743421.758877756] [vision_auto_pick_and_place]: Ejecutando: Ir hacia destino verde (objetivo: green_cube1)
[vision_auto_pick_and_place.py-1] [INFO] [1756743421.758853299] [vision_auto_pick_and_place]: Moviendo a: green_position
[vision_auto_pick_and_place.py-1] [INFO] [1756743421.759113159] [vision_auto_pick_and_place]: Posiciones: [0.4062, 2.06, 1.43, 2.78, 1.5708]
[vision_auto_pick_and_place.py-1] [INFO] [1756743425.763456497] [vision_auto_pick_and_place]: Paso 7/8
[vision_auto_pick_and_place.py-1] [INFO] [1756743425.764008491] [vision_auto_pick_and_place]: Ejecutando: Soltar cubo verde (objetivo: green_cube1)
[vision_auto_pick_and_place.py-1] [INFO] [1756743425.764222071] [vision_auto_pick_and_place]: Abriendo gripper a posición: 0.0
[vision_auto_pick_and_place.py-1] [INFO] [1756743425.971138998] [vision_auto_pick_and_place]: DETACHLINK exitoso para modelo: green_cube1
[vision_auto_pick_and_place.py-1] [INFO] [1756743426.472095741] [vision_auto_pick_and_place]: Paso 8/8
[vision_auto_pick_and_place.py-1] [INFO] [1756743426.472334301] [vision_auto_pick_and_place]: Ejecutando: Volver a home (objetivo: green_cube1)
[vision_auto_pick_and_place.py-1] [INFO] [1756743426.472822361] [vision_auto_pick_and_place]: Moviendo a: home
[vision_auto_pick_and_place.py-1] [INFO] [1756743426.477608491] [vision_auto_pick_and_place]: Posiciones: [0.0, 1.57, 0.0, 0.0, 0.0]
[vision_auto_pick_and_place.py-1] [INFO] [1756743430.477608491] [vision_auto_pick_and_place]: === SECUENCIA GREEN_CUBE_SEQUENCE COMPLETADA PARA green_cube1 ===
[vision_auto_pick_and_place.py-1] [INFO] [1756743430.477889130] [vision_auto_pick_and_place]: Pluck and place completado exitosamente para green_cube1
[vision_auto_pick_and_place.py-1] [INFO] [1756743430.478265371] [vision_auto_pick_and_place]: Objetos procesados: ['green_cube1']
[vision_auto_pick_and_place.py-1] [INFO] [1756743430.478978125] [vision_auto_pick_and_place]: Objetos restantes en cola: 1
[vision_auto_pick_and_place.py-1] [INFO] [1756743430.479958615] [vision_auto_pick_and_place]: Procesando cubo verde green_cube2 en posición: (403, 370)
[vision_auto_pick_and_place.py-1] [INFO] [1756743430.480307082] [vision_auto_pick_and_place]: Quedan 0 cubos en cola
[vision_auto_pick_and_place.py-1] [INFO] [1756743430.480307082] [vision_auto_pick_and_place]: Procesando objeto green_cube2...
```

Figura 7.10 Lectura del terminal de Ubuntu. Representa la ejecución completa del sistema de pick-and-place, desde la detección del objeto green_cube1 hasta la finalización de la tarea, mostrando las posiciones calculadas y los estados del robot en cada paso del proceso. Al finalizar, detecta el cubo green_cube2 y repite el proceso hasta que no existan más elementos.

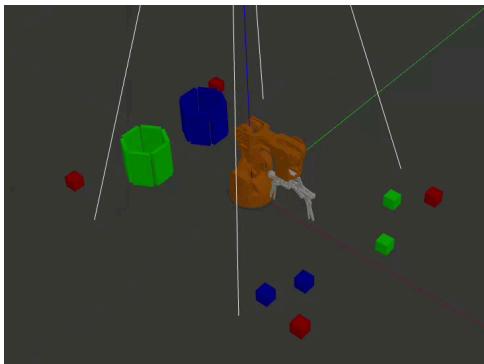


Figura 7.11 Posición home.

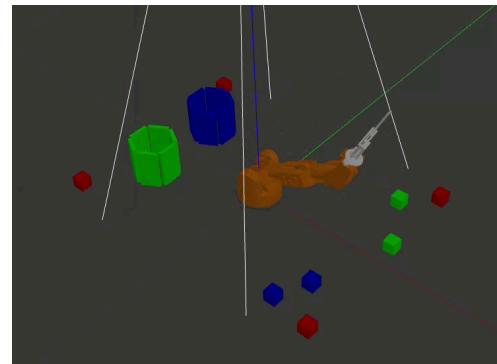


Figura 7.12 Posición de aproximación.

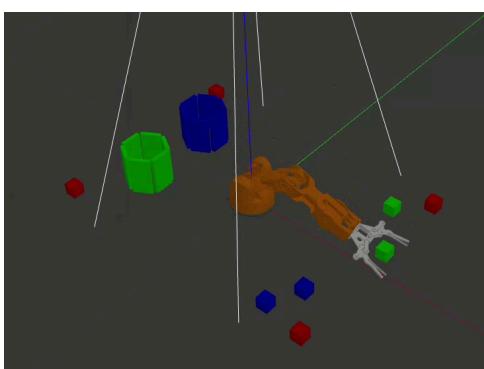


Figura 7.13 Posición de agarre. Gripper abierto

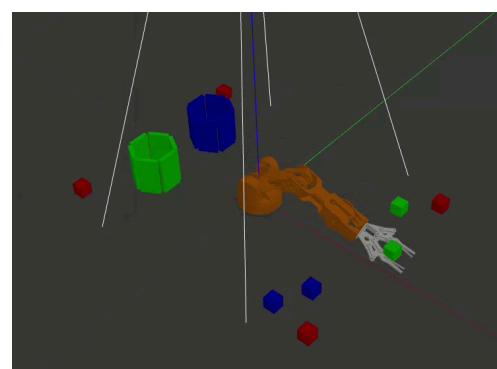


Figura 7.14 Posición de agarre. Gripper cerrado.

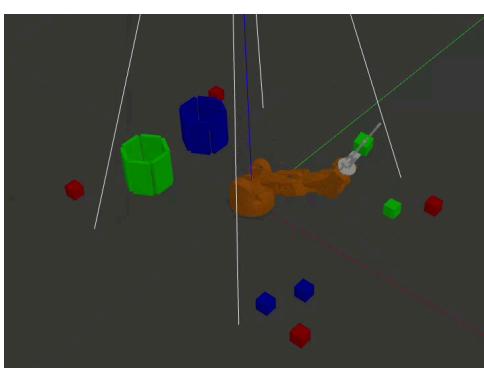


Figura 7.15 Posición de aproximación con objeto.

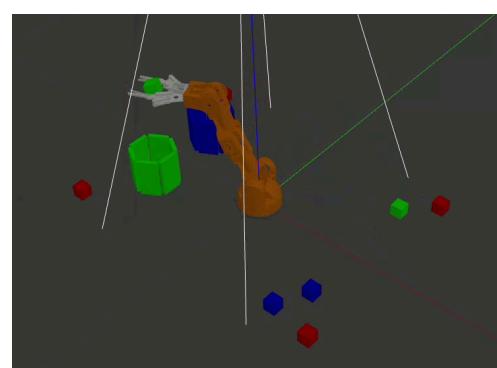


Figura 7.16 Posición depósito con objeto. Gripper cerrado.

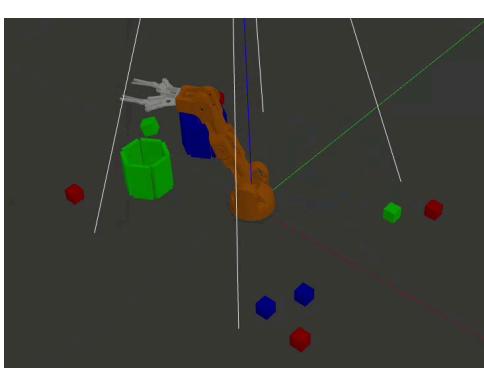


Figura 7.17 Posición depósito sin objeto. Gripper abierto.

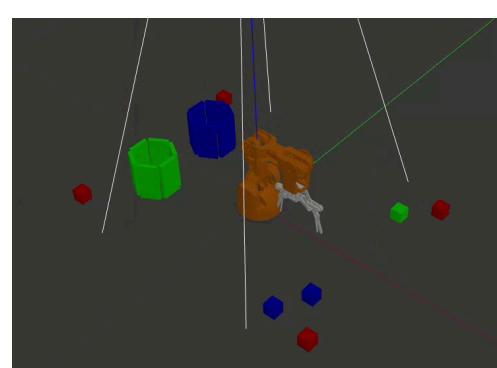


Figura 7.18 Retorno a posición home.

,

8 Sim to real y validación

El proceso de transferencia de un sistema desarrollado y probado en simulación a un entorno real, conocido como «sim to real», implica adaptar y validar el sistema para que funcione de manera efectiva en el mundo físico, enfrentándose a diversas dificultades y diferencias entre ambos entornos. Ante esto, ha sido necesario recrear el entorno simulado en la realidad y, posteriormente, aplicar técnicas de calibración y adaptación para el funcionamiento completo del sistema.

8.1 Montaje robot físico

El montaje del robot físico Braccio Tinkerkit se ha realizado siguiendo las instrucciones proporcionadas por el fabricante [11]. El proceso ha implicado el ensamblaje de las diferentes partes del robot, incluyendo la base, los eslabones y las articulaciones, así como la instalación de los servomotores y la electrónica necesaria para su control. Durante el mismo, se ha seguido un tutorial en vídeo disponible en Youtube que facilita la comprensión del proceso.

Sin embargo, al tratarse de un robot de bajo presupuesto, se han detectado ciertas limitaciones en cuanto a la precisión y robustez del sistema. Las principales limitaciones recaen sobre la calidad del material empleado en la fabricación de las piezas, siendo principalmente plástico, el cual, al estar conectadas dichas piezas mediante tornillos metálicos, puede experimentar deformaciones y holguras que afectan la precisión del robot. Como resultado, pese a que a primera vista no se noten desperfectos en la Figura 8.1, se han experimentado problemas con la adhesión de los componentes plásticos en los ejes de los servomotores, que unido a la baja firmeza que ofrece la pinza, ha generado dificultades en la tarea de trayectoria y manipulación de objetos.

Para mitigar estos problemas, se han realizado ajustes y mejoras en el montaje, como el refuerzo de ciertas uniones con adhesivos y la sustitución de piezas defectuosas. Además, se ha llevado a cabo un mantenimiento regular del robot para asegurar su correcto funcionamiento y prolongar su vida útil.



Figura 8.1 Imágenes finales del montaje del robot Braccio Tinkerkit, mostrando su estructura completa y una visión detallada de la estructura de la pinza.

8.2 Espacio de trabajo

El espacio de trabajo del robot se ha definido en función de las dimensiones del entorno físico y de los objetos a manipular. Se ha optado por un área de trabajo rectangular, delimitada por las posiciones alcanzables del efecto final del robot, obtenidas además de forma experimental en la simulación.

El entorno recreado consta de los siguientes elementos:

- Superficie de trabajo rectangular: se ha construido una plataforma plana y estable donde el robot pueda interactuar cómodamente con los elementos. Esta misma tiene un tamaño aproximado de 90 cm x 70 cm y contiene una serie de líneas paralelas y perpendiculares de unos 10 cm de ancho, que facilitan la calibración visual y la localización de los objetos.
- Marcadores visuales: se han utilizado cubos de colores similares a los empleados en la simulación, con dimensiones aproximadas de 3cm. Estos elementos se han colocado de forma estratégica para barrer el área de trabajo y facilitar la calibración visual.
- Objetos a manipular: se han dispuesto varios vasos de chupito de colores en el área de trabajo, los cuales serán utilizados en las pruebas de manipulación. La elección de estos objetos se ha realizado teniendo en cuenta su tamaño, forma y peso, asegurando que su recolección y manipulación sean factibles. Adicionalmente, se ha estudiado la posibilidad de utilizar otros objetos, como pelotas de ping pong.
- Cámara cenital: se ha instalado un trípode unido a un teléfono móvil que simula la cámara cenital empleada en la simulación. Sin embargo, debido a las limitaciones del trípode y la cámara, la vista ofrecida no es completamente cenital.



Figura 8.2 Imagen del entorno físico recreado para las pruebas del sistema, mostrando la superficie de trabajo con los marcadores visuales, los vasos y el manipulador en el centro del área de trabajo. Junto a éstos, se observa el trípode con el teléfono móvil simulando la cámara cenital.

8.3 Calibraciones y ajustes

Para asegurar un funcionamiento óptimo del sistema en el entorno real, se han llevado a cabo diversas calibraciones y ajustes, destacando las relacionadas con la visión.

El dispositivo móvil empleado como cámara cenital se ha instalado en un trípode, asegurando una posición estable y una vista adecuada del área de trabajo, intentando siempre obtener lo más cercano posible a una vista cenital. La cámara se enlaza al sistema a través de una conexión Wi-Fi, permitiendo la transmisión de video en tiempo real mediante una aplicación específica, llamada Droidcam [25].

Tras su instalación y configuración, se ha procedido a crear un script, *webcam_publisher.py*, encargado de publicar la imagen en el mismo nodo que lo hacía la simulación, permitiendo así un acceso directo al detector de objetos de la información del entorno real.

A continuación, se han modificado los umbrales de detección de objetos en el script, ajustando parámetros como el tamaño mínimo y máximo de los objetos a detectar, así como la colorimetría, pues en función de las condiciones de iluminación se ha observado que ciertos colores no son detectados correctamente.

Finalmente se ha procedido a la calibración de la cámara mediante homografía, siguiendo el mismo procedimiento explicado en la sección correspondiente. Para ello, se han colocado 4 cubos rojos en posiciones conocidas dentro del entorno real y se han capturado sus centroides en píxeles mediante el sistema de detección de objetos. Sin embargo, al no tratarse de una vista completamente cenital, ha resultado en valores muy imprecisos. La solución propuesta ha sido la adición de más marcadores visuales, logrando una precisión aceptable para las tareas de pick-and-place previstas.

Posición XY (m)	Posición XY (px)
-0.325, -0.215	115, 98
0.0, -0.215	297, 107
0.325, -0.215	497, 113
-0.2, -0.1	180, 165
0.2, -0.1	412, 179
-0.325, 0.0	106, 222
0.0, 0.0	288, 233
0.325, 0.0	489, 246
-0.2, 0.1	169, 282
0.2, 0.1	406, 301
-0.325, 0.215	97, 344
0.0, 0.215	279, 361
0.325, 0.215	461, 381



Tabla 8.1 Posiciones de los marcadores en el escenario creado y sus centroides capturados por la cámara del móvil en píxeles, junto con la detección de los marcadores en el entorno real. Tal como se observa, los píxeles de la tabla y la imagen coinciden, al igual que las posiciones en metros. Sin embargo, estas últimas están señaladas en el cartón con bolígrafo, por lo que no se llegan a distinguir perfectamente en la imagen.

Respecto a la calibración del brazo robótico, no ha sido necesario adaptar su configuración, puesto que el sistema de nodos, publicaciones y suscripciones se mantiene idéntico al de la simulación, permitiendo el movimiento del robot sin necesidad de ajustes adicionales.

8.4 Validación del sistema

La validación del sistema se ha llevado a cabo mediante una serie de pruebas diseñadas para evaluar su desempeño en el entorno real. Estas pruebas han incluido la detección y manipulación de objetos en diferentes condiciones, así como la evaluación de la precisión.

8.4.1 Visión

Respecto al apartado de la visión se ha observado que, pese a los ajustes realizados, la detección de objetos sigue siendo un desafío debido a las variaciones en la iluminación, siendo el sistema capaz de discernir entre un vaso de chupito rojo y uno verde, pero mostrando dificultades en la detección de los tonos de rojo, provocadas por ejemplo por la sombra que proyecta el trípode en éstos. Adicionalmente, cabe destacar la dificultad para la obtención de los parámetros de la cámara debido a la imposibilidad de acceder a ellos en el dispositivo móvil.

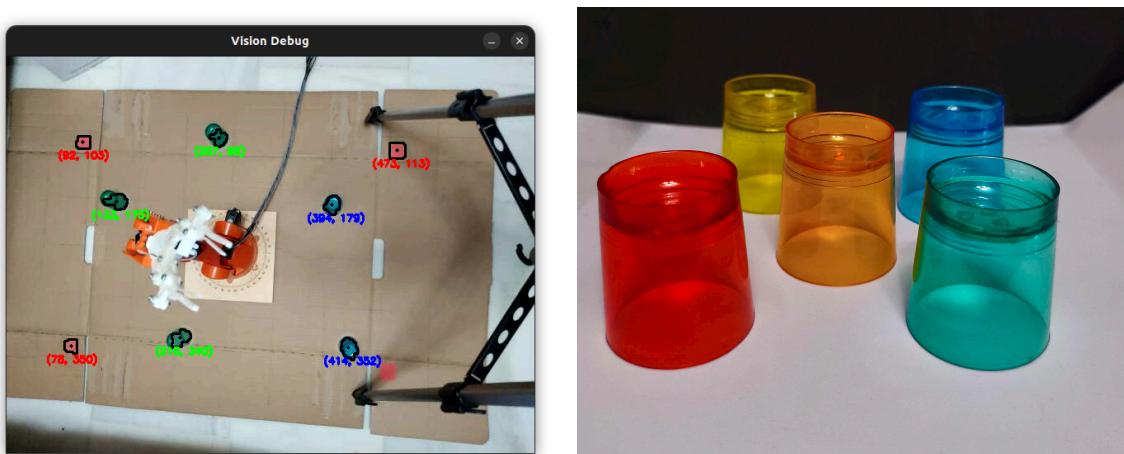


Figura 8.3 Imágenes de los vasos de chupito empleados en las pruebas de manipulación, mostrando la variedad de colores y la dificultad que presenta el sistema de visión para su detección. Durante la detección se puede observar los problemas causados por la perspectiva de la cámara y las variaciones en la iluminación, que afectan a la precisión del sistema para calcular el centroide y clasificar los objetos presentes.

8.4.2 Localización

Enlazado con lo anterior, en cuanto a la localización de los objetos, el sistema ha mostrado un rendimiento aceptable, siendo capaz de identificar la posición de los vasos de chupito en el espacio tridimensional de forma relativamente precisa. Sin embargo, se han observado algunos errores de localización, causados por la propia imprecisión al crear la superficie de trabajo o la colocación manual de los marcadores de referencia.

```
Matriz de homografía calculada :
[[ 1.84459602e-03  1.28172514e-04 -5.62022026e-01
  [-1.21147752e-04  1.82157430e-03 -3.88514485e-01]
  [ 2.52136693e-04  6.64246129e-05  1.00000000e+00]]

Verificación:
Pixel (460,156) -> Mundo [ 0.2721076  -0.14212061]
Esperado: (0.25, -0.15)
Error: 0.0235m
Homografía guardada en braccio_vision/config/camera_calibration.json
```

Figura 8.4 Lectura del terminal de Ubuntu. Representa el cálculo de la homografía en el entorno real, mostrando los puntos en píxeles y metros de un objeto seleccionado al azar, junto con la matriz de homografía obtenida y el error de reprojeción final, superior al obtenido en simulación pero aún aceptable para las tareas previstas.

8.4.3 Trayectoria

En lo que respecta a la trayectoria, el sistema ha encontrado limitaciones causados por los errores físicos producidos durante el montaje, unido a los errores explicados en las secciones anteriores. Como resultado, se han observado desviaciones en la trayectoria planificada, obteniendo un rendimiento más favorable cuando el control ha sido realizado mediante el mando a distancia.

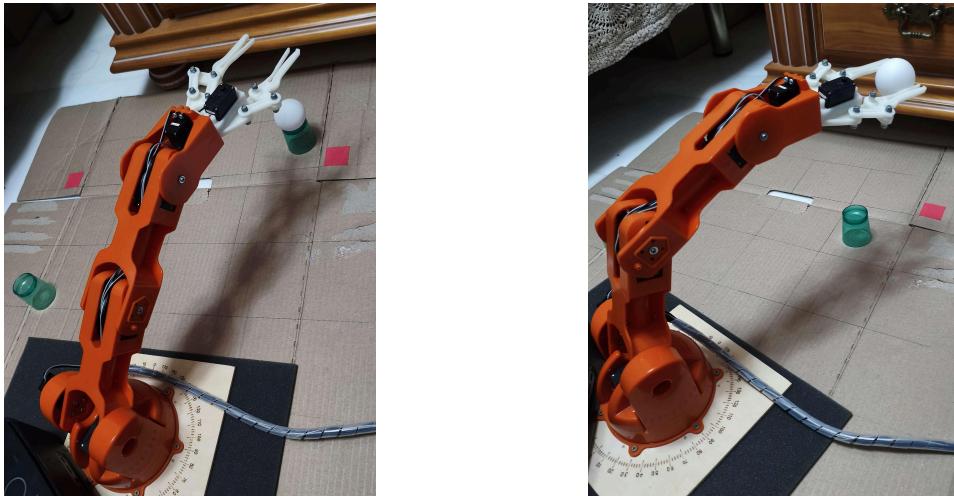


Figura 8.5 Imágenes del robot Braccio Tinkerkit en diferentes posiciones alcanzadas durante las pruebas de recolección en el entorno real, mostrando la capacidad del sistema para posicionarse y recoger objetos en el espacio de trabajo.

8.4.4 Manipulación

En cuanto a la manipulación de objetos, el sistema ha demostrado ser efectivo en la recolección y colocación de los vasos de chupito. La garra permite una sujeción mucho más firme de lo esperado, unido a la estructura ligera y compacta del vaso de plástico, conformando un proceso de recolección ágil y eficiente. Sin embargo, se han identificado algunas limitaciones en la precisión del agarre, causadas por la pobre amplitud de la garra cuando se encuentra abierta, siendo realmente tedioso recoger objetos de tamaño mediano. De igual forma, ha sido capaz de manipular objetos redondos como pelotas de ping pong, siempre y cuando se encuentren éstas en una superficie plana.

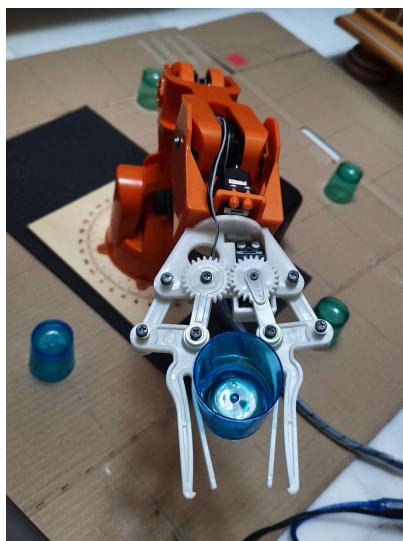


Figura 8.6 Imagen del robot Braccio Tinkerkit recogiendo un vaso de chupito en el entorno real, demostrando la capacidad del sistema para realizar tareas de agarre de objetos con éxito. En la Figura 8.5 se puede observar también el éxito durante la recolección de una pelota de ping pong

9 Conclusión

Este Trabajo de Fin de Grado se propuso bajo la finalidad de diseñar, simular y validar un sistema robótico de pick-and-place para el manipulador de bajo presupuesto Braccio Tinkerkit, utilizando el framework ROS 2. El propósito era claro y directo: crear una plataforma funcional y modular para la experimentación, que facilitara la adaptación al mundo real y, al mismo tiempo, adquirir competencias avanzadas en herramientas estándar de la industria robótica, como pueden ser ROS 2, Gazebo o GitHub.

La contribución principal de este proyecto ha sido la integración de múltiples subsistemas en un flujo de trabajo coherente, desde la simulación a la validación real. Se desarrollaron paquetes específicos como *braccio_vision*, dedicado a la percepción por computador y el cálculo de la cinemática, *braccio_gamepad_teleop* para el control manual o *sim-to-real* para la adaptación al mundo real; demostrando la extensibilidad de la arquitectura base.

La metodología de percepción, que combina la detección de objetos por color con una calibración por homografía, proporcionó la precisión necesaria para la localización de objetos en el espacio de trabajo, como valida el bajo error de reprojeción obtenido en la Figura 6.6. Esta metodología, enlazada con las técnicas del cálculo de la cinemática inversa y la planificación de trayectorias mediante MoveIt2, permitió la ejecución autónoma de tareas de manipulación en el entorno simulado, como se observa en la Figura 7.10.

Uno de los desafíos técnicos más relevantes fue el cálculo de la cinemática inversa, que requirió la implementación de algoritmos específicos para determinar las posiciones articulares necesarias para alcanzar un objetivo en el espacio tridimensional. Esto implicó un profundo entendimiento de la geometría del manipulador y la capacidad de modelar su comportamiento de manera precisa, pues durante largos días se experimentó con diferentes enfoques, logrando incluso que la orientación fuera la correcta pero las posiciones articulares no coincidieran como en la Figura 9.1. Asimismo, tras superar ese obstáculo y descubrir la falta de un plugin nativo en ROS 2 para el agarre, casi rompo a llorar. Agradecer finalmente a la herramienta *libgazebo_link_attacher*[24], que junto con el desarrollo de una lógica de detección de proximidad, se consiguió gestionar dinámicamente la unión y separación de objetos, y solventar así la mayoría de mis problemas.

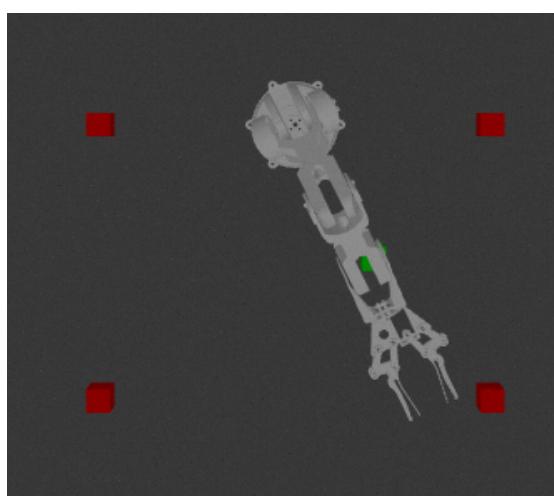


Figura 9.1 Fallo en la colocación del brazo robótico para la recolección del cubo verde. En esta imagen se puede observar la discrepancia entre la posición del cubo y la del brazo, compartiendo orientación pero no ubicación del efecto final.

A nivel personal, este proyecto ha consolidado mi formación como ingeniero, forzándome a superar la curva de aprendizaje de ROS 2 y a afrontar la brecha entre la simulación ideal y las complejidades del hardware real, como las holguras mecánicas, la variabilidad del entorno y la precisión de los sensores.

En simulación, el sistema demostró ser robusto, completando ciclos de manipulación de forma autónoma. Sin embargo, la transferencia al robot físico evidenció las limitaciones de un hardware educativo: la baja rigidez de los componentes plásticos y la precisión de los servomotores, introducen desviaciones que no se observan en el modelo ideal. Asimismo, el sistema de visión es sensible a cambios de iluminación, lo que exige un entorno controlado el cual, adicionalmente, tiene un error humano añadido pues todas las variables que influyen no se pueden controlar.

Como línea de trabajo futuro, se ha mejorado consideradamente la base ofrecida por Jaume Mulet [2], implementando una estructura mucho más compleja y modular. Por ello, se proponen varias extensiones que pueden dar un salto de calidad al proyecto:

En primer lugar destacaría la implementación de un sistema de detección de objetos mucho más robusto, basado en aprendizaje profundo, que permita la identificación de objetos en condiciones de iluminación variables y con mayor precisión. Esto podría lograrse mediante el entrenamiento de un modelo de red neuronal convolucional utilizando un conjunto de datos adecuado, ya sea entrenado previamente como Yolo, o bien un modelo personalizado adaptado a las necesidades específicas del entorno de trabajo.

En segundo lugar, se podría explorar la integración de sensores adicionales, como cámaras RGB-D, cuyos parámetros intrínsecos se podrían calibrar de manera más precisa. Esta mejora permitiría diseñar un espacio de trabajo más profesional, donde la luz sea proveniente de diferentes focos, minimizando las sombras y reflejos; donde el trípode sea mucho más regulable y estable.

Por último, se podría investigar la implementación de técnicas de aprendizaje por refuerzo para mejorar la planificación de trayectorias y la toma de decisiones en entornos dinámicos, donde existan multitud de obstáculos. Este caso se podría incluso extrapolar a un sistema mayor, como una línea de producción, donde existan varios robots y cintas transportadoras encargadas de la manipulación de objetos.

En definitiva, este TFG no solo ha producido un sistema funcional, sino que ha generado una plataforma de código abierto, documentada y modular. Todo el trabajo desarrollado se encuentra disponible en mi repositorio [17], donde se incluyen instrucciones detalladas para su ejecución y vídeos que reflejan el funcionamiento del mismo. Se espera que éste pueda servir como una valiosa herramienta educativa para futuros estudiantes que deseen iniciarse en la robótica avanzada y para quienes quieran tomar las riendas de este proyecto, aplicando las implementaciones propuestas.

Índice de Figuras

1.1.	Figura 1.1 Juguete para niños, kit de construcción de un vehículo todo-terreno Meccano	1
2.1.	Figura 2.1 Representación original de la obra teatral de Karel Čapek, donde se observan un hombre junto a una mujer y tres robots.	3
2.2.2.	Figura 2.2 Representación gráfica del número de productores de robots de servicio por grupo de aplicación y origen en 2024 [7].	5
2.3.	Figura 2.3 Representación gráfica del crecimiento en la cantidad de robots industriales operando en el mercado durante los últimos 10 años según World Robotics en 2024 [7].	6
2.3.	Figura 2.4 Proyectos compartidos por la comunidad de Autodesk Instructables, donde se explica mediante tutoriales y documentación cómo construir brazos robóticos [9].	6
2.3.	Figura 2.5 Proyectos compartidos por la comunidad de Arduino Project Hub, donde se explica mediante tutoriales y documentación la construcción y control de brazos robóticos mediante Arduino [10].	6
2.4.	Figura 2.6 Montajes posibles del Braccio Tinkerkit, incluyendo algunos sustitutos de la pinza.	7
2.4.	Figura 2.7 Estructura del Braccio Tinkerkit.	8
2.4.	Figura 2.8 Placa de expansión (shield) utilizada para la conexión de los servomotores. En ella se puede visualizar la disposición de los pines naranjas etiquetados con la numeración correspondiente	9
2.4.	Figura 2.9 Placa Arduino Uno utilizada como controlador principal del robot Braccio Tinkerkit.	9
3.1.3.	Figura 3.1 Tabla comparativa del incremento de descargas de ROS 2, siendo Humble y Jazzy sus exponentes, frente a la decadencia de las distribuciones anteriores, destacando Noetic como última versión de ROS 1 [15].	11
3.2.4.	Figura 3.2 Diagrama de flujo del sistema ROS2 con MoveIt2, Gazebo y el resto de herramientas utilizadas para la simulación y control del proyecto.	14
4.1.	Figura 4.1 Ilustración de la simulación en RViz y Gazebo del robot manipulador realizando una trayectoria. En la izquierda, se puede observar el brazo en sus posiciones inicial, actual y final. A su derecha, la representación de Gazebo mostrando esa posición actual junto al entorno simulado.	15
4.3.1.	Figura 4.2 Representación del robot Braccio Tinkerkit en Gazebo, mostrando dos perspectivas diferentes del manipulador.	17
4.3.2.	Figura 4.3 Representación completa del entorno de simulación, incluyendo el robot, los recipientes, la cámara y los objetos manipulables. A la derecha, la vista cenital de la cámara simulada 0.6m sobre el suelo, mostrando los cubos de diferentes colores.	17
5.1.	Figura 5.1 Lectura del terminal de Ubuntu. Representa el incremento en la posición de la articulación base al mantener el joystick inclinado hacia la izquierda durante unos segundos, siendo LX el valor del joystick izquierdo y el primer término de POS, la posición angular de la base del robot en radianes. El resto de valores nulos corresponden al estado de los botones que no se han pulsado.	19
5.2.2.	Figura 5.2 Diagrama del mando de PS4 con el mapeo de los botones y joysticks a las articulaciones y acciones del robot mencionadas previamente.	20

5.3.	Figura 5.3 Servicio de Gazebo para el pick and place de objetos mediante el plugin gazebo_link_attacher. En la imagen se observa la petición de attach al servicio, el cálculo de las distancias respecto la posición del gripper y los cubos; y tras la verificación del umbral de cercanía, la ejecución de la acción entre la pinza y el blue_cube1.	21
6.	Figura 6.1 Esquema del flujo de percepción y detección de objetos para el sistema de pick-and-place.	23
6.1.	Figura 6.2 Representación de las cámaras simuladas, mostrando la vista del sensor raw y la vista de depuración tras la detección de los objetos. En esta última, se puede observar el marcador de cada objeto detectado junto con las coordenadas de su centro en sus respectivos colores.	24
6.2.	Figura 6.3 Ecuaciones para el cálculo de las coordenadas normalizadas y su proyección al plano, basadas en el modelo geométrico mostrado en la derecha [22]. Los parámetros utilizados se corresponden con la posición del objeto en píxeles (pixel _x , pixel _y), el centro de la cámara (c _x , c _y), las distancias focales (f _x , f _y) y la altura de la cámara (Z).	25
6.2.	Figura 6.4 Lectura del terminal de Ubuntu tras la ejecución de <i>object_detector.py</i> . Representa la cantidad de objetos detectados, clasificados por su color, junto a sus coordenadas en píxeles, coincidentes con los datos de la Figura 6.2. Los cubos temporalmente bloqueados son aquellos destinados a la manipulación en el entorno simulado, estudiados en secciones posteriores.	25
6.3.	Figura 6.5 Lectura del terminal de Ubuntu tras la ejecución de <i>Calculate_homography.py</i> . Representa la matriz de homografía calculada .	26
6.3.	Figura 6.6 Lectura del terminal de Ubuntu tras la ejecución de <i>test_homography.py</i> . Representa los resultados de la validación de la matriz de homografía donde el error en la reprojeción es muy bajo. Al comparar la posición real de un cubo adicional (0.35, 0.05) con la posición estimada mediante la matriz (0.3497, 0.04864) se observa un error de 1.38 mm, ampliamente asumible para un robot manipulador.	26
7.1.1.	Figura 7.1 Diagrama esquemático del brazo robótico, mostrando las longitudes de los eslabones implicados y las articulaciones relevantes para el cálculo de la cinemática inversa descrito posteriormente.	27
7.1.1.	Figura 7.2 Representación de las pinzas del robot, mostrando los dos tipos de agarre. El primero es un agarre en pinza o directo, ideal para objetos de media y larga distancia, mientras que el segundo es un agarre en forma de gancho, ideal para objetos de corta distancia.	28
7.1.1.	Figura 7.3 Ecuaciones para el cálculo de los ángulos articulares de un brazo robótico.	28
7.1.2.	Figura 7.4 Representación del problema de la base y su rango de movimiento. En la imagen izquierda se identifica una posición objetivo que se encuentra en el rango (0, 180°). En la derecha, el objeto se encuentra fuera de este rango, lo que implica una solución donde el robot oriente su base como en el primer caso, y aplique simetría en el codo y demás articulaciones del robot.	29
7.1.3.	Figura 7.5 Lectura del terminal de Ubuntu. Muestra el cálculo de la cinemática inversa para una posición objetivo x=0.35, y=0.05 y z=0.025, donde se observa el procedimiento ejecutado hasta el cálculo final, mostrando finalmente la posición de agarre y aproximación en radianes y grados.	30
7.1.3.	Figura 7.6 Lectura del terminal de Ubuntu. Muestra el cálculo de la cinemática inversa para una posición objetivo simétrica: x=0.28, y=-0.15 y z=0.025, donde se observa el procedimiento ejecutado hasta el cálculo final. La posición de agarre se establece en [151.8, 174.3, 87.1, 176.1, 90.0]°, siendo la posición de aproximación [151.8, 147.0, 24.0, 123.0, 90.0]°.	30
7.1.4.	Figura 7.7 Lectura del terminal de Ubuntu tras la ejecución del script <i>ik_workspace_tester_py</i> . Muestra el resultado de las pruebas de cinemática inversa, indicando si las posiciones son aptas o si presentan problemas. En este fragmento, se muestra un rango de posición X comprendido entre 0.21 y 0.23, mientras que el rango Y se encuentra entre 0.15 y 0.30, con un paso de 0.02, mostrando así las posiciones cómodas para la ejecución del robot en un rango determinado.	31

7.2.	Figura 7.8 Lectura del terminal de Ubuntu. Representa la petición de detach al servicio, el cálculo de las distancias respecto la posición del gripper y los cubos; y la ejecución de la acción de detach, donde se puede mostrar el cubo azul cayendo desde la pinza.	32
7.3.	Figura 7.9 Diagrama del flujo de acción del sistema de pick-and-place, mostrando la interacción entre los subsistemas de percepción, planificación y control.	33
7.3.	Figura 7.10 Lectura del terminal de Ubuntu. Representa la ejecución completa del sistema de pick-and-place, desde la detección del objeto green_cube1 hasta la finalización de la tarea, mostrando las posiciones calculadas y los estados del robot en cada paso del proceso. Al finalizar, detecta el cubo green_cube2 y repite el proceso hasta que no existan más elementos.	34
7.3.	Figura 7.11 Posición home.	35
7.3.	Figura 7.12 Posición de aproximación.	35
7.3.	Figura 7.13 Posición de agarre. Gripper abierto	35
7.3.	Figura 7.14 Posición de agarre. Gripper cerrado.	35
7.3.	Figura 7.15 Posición de aproximación con objeto.	35
7.3.	Figura 7.16 Posición depósito con objeto. Gripper cerrado.	35
7.3.	Figura 7.17 Posición depósito sin objeto. Gripper abierto.	35
7.3.	Figura 7.18 Retorno a posición home.	35
8.1.	Figura 8.1 Imágenes finales del montaje del robot Braccio Tinkerkit, mostrando su estructura completa y una visión detallada de la estructura de la pinza.	37
8.2.	Figura 8.2 Imagen del entorno físico recreado para las pruebas del sistema, mostrando la superficie de trabajo con los marcadores visuales, los vasos y el manipulador en el centro del área de trabajo. Junto a éstos, se observa el trípode con el teléfono móvil simulando la cámara cenital.	38
8.4.1.	Figura 8.3 Imágenes de los vasos de chupito empleados en las pruebas de manipulación, mostrando la variedad de colores y la dificultad que presenta el sistema de visión para su detección. Durante la detección se puede observar los problemas causados por la perspectiva de la cámara y las variaciones en la iluminación, que afectan a la precisión del sistema para calcular el centroide y clasificar los objetos presentes.	40
8.4.2.	Figura 8.4 Lectura del terminal de Ubuntu. Representa el cálculo de la homografía en el entorno real, mostrando los puntos en píxeles y metros de un objeto seleccionado al azar, junto con la matriz de homografía obtenida y el error de reprojeción final, superior al obtenido en simulación pero aún aceptable para las tareas previstas.	40
8.4.3.	Figura 8.5 Imágenes del robot Braccio Tinkerkit en diferentes posiciones alcanzadas durante las pruebas de recolección en el entorno real, mostrando la capacidad del sistema para posicionarse y recoger objetos en el espacio de trabajo.	41
8.4.4.	Figura 8.6 Imagen del robot Braccio Tinkerkit recogiendo un vaso de chupito en el entorno real, demostrando la capacidad del sistema para realizar tareas de agarre de objetos con éxito. En la Figura 8.5 se puede observar también el éxito durante la recolección de una pelota de ping pong	41
9.	Figura 9.1 Fallo en la colocación del brazo robótico para la recolección del cubo verde. En esta imagen se puede observar la discrepancia entre la posición del cubo y la del brazo, compartiendo orientación pero no ubicación del efecto final.	43

Índice de Tablas

2.2.1.	Tabla 2.1 Clasificación de los robots industriales en función de su estructura mecánica [5].	4
2.4.	Tabla 2.2 Especificaciones técnicas principales del Braccio Tinkerkit, obtenidas directamente de la web oficial de compra de arduino [11].	7
2.4.	Tabla 2.3 Asignación de servomotores a las articulaciones del Braccio Tinkerkit, junto al rango de movimiento admisible de cada uno.	8
2.4.	Tabla 2.4 Especificaciones comparativas de los servomotores utilizados en el Braccio Tinkerkit [11].	8
2.4.	Tabla 2.5 Especificaciones técnicas de la placa mostrada en la Figura 2.8.	9
2.4.	Tabla 2.6 Especificaciones técnicas de la placa mostrada en la Figura 2.9 [12].	9
3.1.4.	Tabla 3.1 Tabla comparativa entre MATLAB, ROS y ROS2 como opciones para la simulación y control del manipulador.	12
6.3.	Tabla 6.1 Posiciones de los cubos rojos en el entorno simulado y sus centroides capturados por la cámara en píxeles.	26
8.3.	Tabla 8.1 Posiciones de los marcadores en el escenario creado y sus centroides capturados por la cámara del móvil en píxeles, junto con la detección de los marcadores en el entorno real. Tal como se observa, los píxeles de la tabla y la imagen coinciden, al igual que las posiciones en metros. Sin embargo, estas últimas están señaladas en el cartón con bolígrafo, por lo que no se llegan a distinguir perfectamente en la imagen.	39

Bibliografía

- [1] EMB Global, «Transforming Learning: The Impact of Robotics on Education». [En línea]. Disponible en: <https://blog.emb.global/transforming-learning-with-robotics/>
- [2] Jaume Mulet, «ROS2 Braccio». [En línea]. Disponible en: https://github.com/jaMulet/ROS2_braccio
- [3] Tomás Fernández y Elena Tamaro, «Biografía de Karel Čapek». [En línea]. Disponible en: <https://www.biografiasyvidas.com/biografia/c/capek.htm>
- [4] International Organization for Standardization, «ISO 8373:2021 - Robots and robotic devices – Vocabulary». [En línea]. Disponible en: <https://www.iso.org/obp/ui/#iso:std:iso:8373:ed-3:v1:en>
- [5] International Federation of Robotics, «Industrial Robots». [En línea]. Disponible en: <https://ifr.org/industrial-robots>
- [6] International Federation of Robotics, «Service Robots». [En línea]. Disponible en: <https://ifr.org/wr-service-robots/>
- [7] International Federation of Robotics, «World Robotics 2024 - Press Conference». [En línea]. Disponible en: https://ifr.org/img/worldrobotics/Press_Conference_2024.pdf
- [8] Zbigniew Nawrat, «Medical Robots. A medical robot – what is it?». [En línea]. Disponible en: https://yadda.icm.edu.pl/baztech/element/bwmeta1.element.baztech-49bb2863-0aca-44b4-a5fb-581a59639348/c/MRR10_101-104.pdf
- [9] Instructables, «Robotic Arm Projects». [En línea]. Disponible en: <https://www.instructables.com/search/?q=robotic%20arm&projects=featured>
- [10] Arduino, «Arduino Project Hub». [En línea]. Disponible en: <https://projecthub.arduino.cc/?category=Motors+%26+Robotics&sortBy=>
- [11] Arduino, «Braccio Tinkerkit». [En línea]. Disponible en: https://store.arduino.cc/products/tinkerkit-braccio-robot?srsltid=AfmBOop3_QGF-ekRRLRKSMZ3r3J3aKQ2ERPQLrD1_9bns6Bmv6cOvYZZ
- [12] Arduino, «Arduino Uno». [En línea]. Disponible en: https://es.wikipedia.org/wiki/Arduino_Uino
- [13] Murtaza Bohra, «Leveraging MATLAB®/Simulink® Toolboxes to Rapidly Deploy a Pick & Place Application». [En línea]. Disponible en: <https://www.quanser.com/blog/robotics-haptics/leveraging-matlab-simulink-toolboxes-to-rapidly-deploy-a-pick-place-application/>
- [14] Stedden, «Simulating the Braccio robotic arm with ROS and Gazebo». [En línea]. Disponible en: <https://opus.stedden.org/2020/08/braccio-moveit-gazebo/>
- [15] Katherine_Scott, Open Robotics, «2024 ROS Metrics Report». [En línea]. Disponible en: <https://discourse.openrobotics.org/t/2024-ros-metrics-report/42354>
- [16] arkhipenko, «TaskScheduler». [En línea]. Disponible en: <https://github.com/arkhipenko/TaskScheduler>
- [17] Iván Luque Valverde, «Braccio-Tinkerkit-Arduino». [En línea]. Disponible en: <https://github.com/Ivan-Luque-Valverde/Braccio-Tinkerkit-Arduino>
- [18] PickNik Robotics, «How to Teleoperate a Robotic Arm with a Gamepad». [En línea]. Disponible en: https://moveit.picknik.ai/main/doc/how_to_guides/controller_teleoperation/controller_teleoperation.html
- [19] ROS, «Joy Linux». [En línea]. Disponible en: https://index.ros.org/p/joy_linux/
- [20] Will Stedden, «su_chef prototype 1: My robotic arm that detects apples and picks them up». [En línea]. Disponible en: <https://opus.stedden.org/2020/07/su-chef-braccio-yolo/>
- [21] Nathan Naert, «Python-controlled-Braccio-robot-arm». [En línea]. Disponible en: <https://github.com/NNaert/Python-controlled-Braccio-robot-arm>

- [22] Sophia.feng, GoerMicro, «Machine vision system coordinate systems and parameters of camera». [En línea]. Disponible en: <https://industry.goermicro.com/blog/tech-briefs/machine-vision-coordinate-systems-and-parameters-of-camera.html>
- [23] ROS, «Planning Scene ROS API Tutorial». [En línea]. Disponible en: https://docs.ros.org/en/melodic/api/moveit_tutorials/html/doc/planning_scene_ros_api/planning_scene_ros_api_tutorial.html
- [24] IFRA Cranfield University, «IFRA Link Attacher». [En línea]. Disponible en: https://github.com/IFRA-Cranfield/IFRA_LinkAttacher
- [25] DroidCam, «DroidCam». [En línea]. Disponible en: <https://droidcam.app/>

