

Trabajo Fin de Grado

Ingeniería Electrónica, Robótica y Mecatrónica

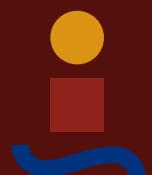
Simulación de un sistema de Pick and Place con un robot Braccio Tinkerkit de Arduino bajo ROS 2

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 de un sistema de Pick and Place con
un robot Braccio Tinkerkit de Arduino bajo ROS**
2

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 de un sistema de Pick and Place con un robot Braccio Tinkerkit de Arduino bajo ROS 2

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 llevo 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

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliquam quaerat voluptatem. Ut enim aequo doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distingue possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliquam quaerat voluptatem. Ut enim aequo doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

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
5. Entorno de simulación	17
5.1. Creación del mundo	17
5.2. Robot manipulador	17
5.3. Spawner de Cámara y Cubos	18
6. Control mediante PS4 controller	19
6.1. Arquitectura y filosofía de control	19
6.2. Mapeo de botones y joysticks	20
6.2.1. Joysticks	20

6.2.2.	Botones	20
6.3.	Flujo de datos y control	21
7.	Percepción y localización de objetivos	23
7.1.	Sensor de la cámara	24
7.2.	Detección de objetos	24
7.3.	Matriz de Homografía	25
8.	Planificación de agarre y manipulación	27
8.1.	Cinemática directa e inversa	27
8.2.	Repositorio attach/detach	27
8.3.	Transferencia sim-to-real y validación experimental	27
9.	Evaluación y métricas	29
10.	Huecos detectados y oportunidades para el TFG	31
	<i>Índice de Figuras</i>	33
	<i>Índice de Tablas</i>	35
	<i>Bibliografía</i>	37

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.

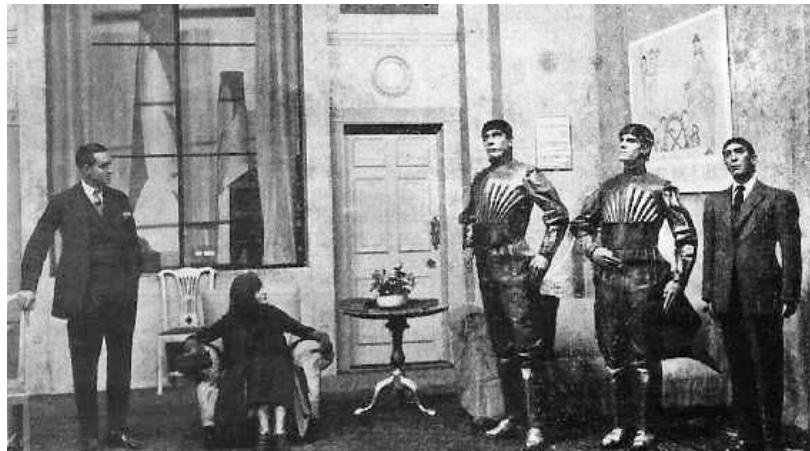


Figura 2.1 Representación original de la obra teatral de Karel Čapek, donde se observan un hombre junto a una mujer y tres robots.

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.

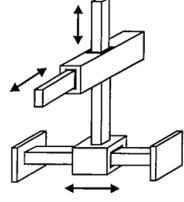
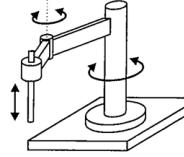
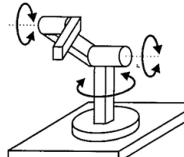
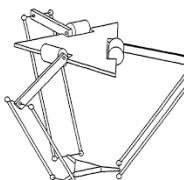
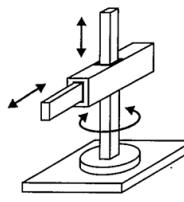
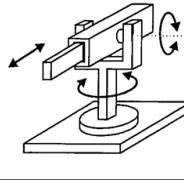
Nombre	Estructura mecánica	Imagen
Robot cartesiano	Manipulador que tiene tres articulaciones prismáticas, cuyos ejes forman un sistema de coordenadas cartesianas.	
Robot SCARA	Manipulador que tiene dos articulaciones rotatorias paralelas para proporcionar flexibilidad en un plano seleccionado.	
Robot 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.

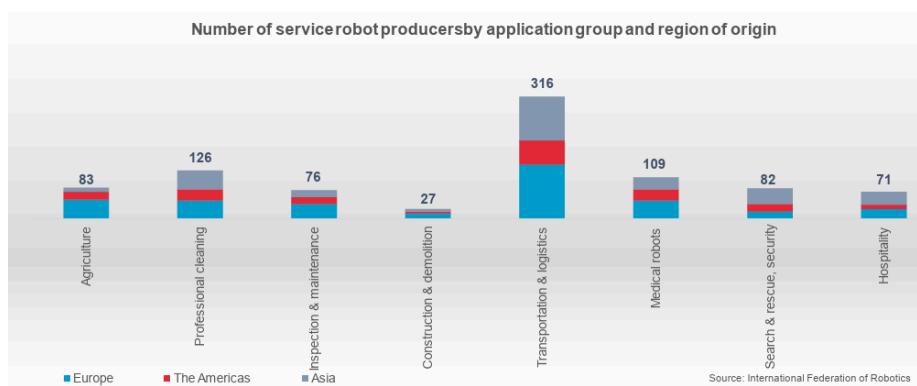


Figura 2.2 Representación gráfica del número de productores de robots de servicio por grupo de aplicación y origen en 2024 [7].

2.2.3 Robots médicos

Los robots médicos constituyen ahora una tercera área de aplicación, catgorizándose 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 ofertada 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.

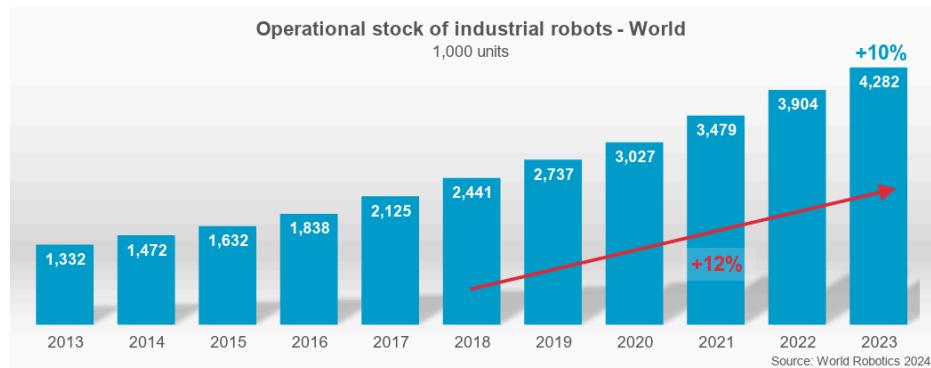


Figura 2.3 Representación gráfica del crecimiento en la cantidad de robots industriales operando en el mercado durante los últimos 10 años según World Robotics en 2024 [7].

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.

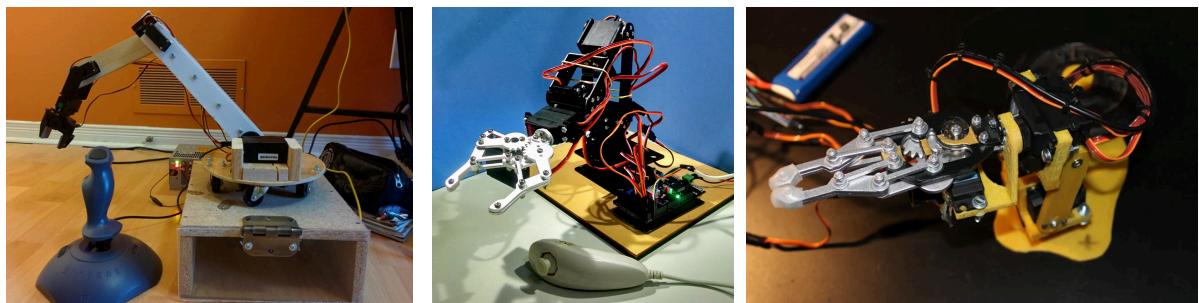


Figura 2.4 Proyectos compartidos por la comunidad de Autodesk Instructables, donde se explica mediante tutoriales y documentación cómo construir brazos robóticos [9].

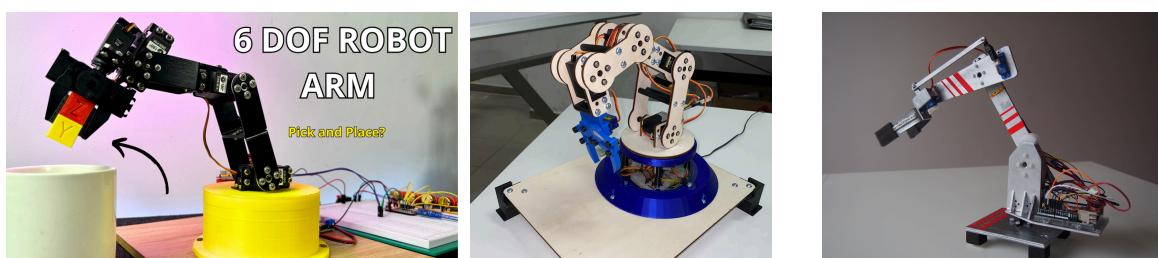


Figura 2.5 Proyectos compartidos por la comunidad de Arduino Project Hub, donde se explica mediante tutoriales y documentación la construcción y control de brazos robóticos mediante Arduino [10].

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.

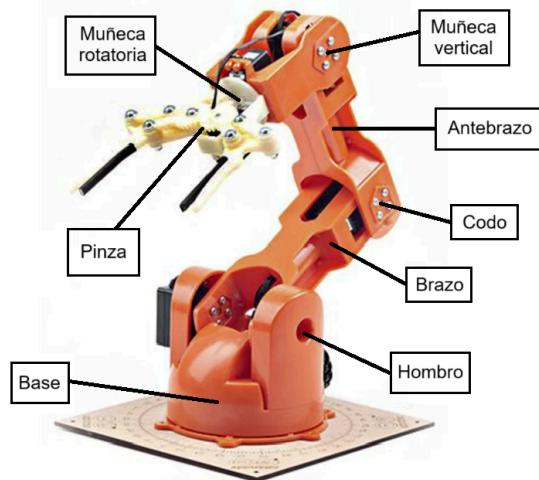


Figura 2.7 Estructura del Braccio Tinkerkit.

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)	$42 \times 20.5 \times 39.5$	$31.3 \times 16.5 \times 28.6$
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.

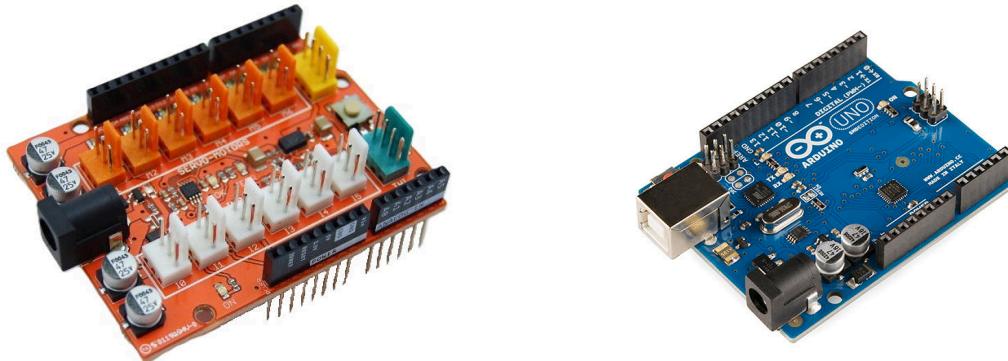


Figura 2.8 Placa de expansión (shield) utilizada para la conexión de los servomotores. En ella se puede visualizar la disposición de los pines naranjas etiquetados con la numeración correspondiente

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].

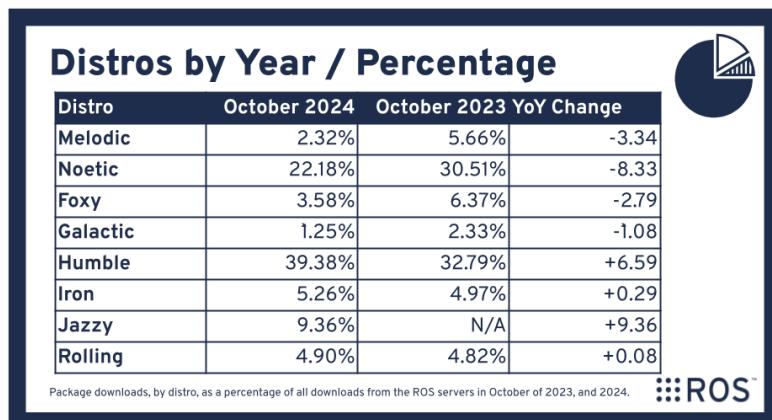


Figura 3.1 Tabla comparativa del incremento de descargas de ROS 2, siendo Humble y Jazzy sus exponentes, frente a la decadencia de las distribuciones anteriores, destacando Noetic como última versión de ROS 1 [15].

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.

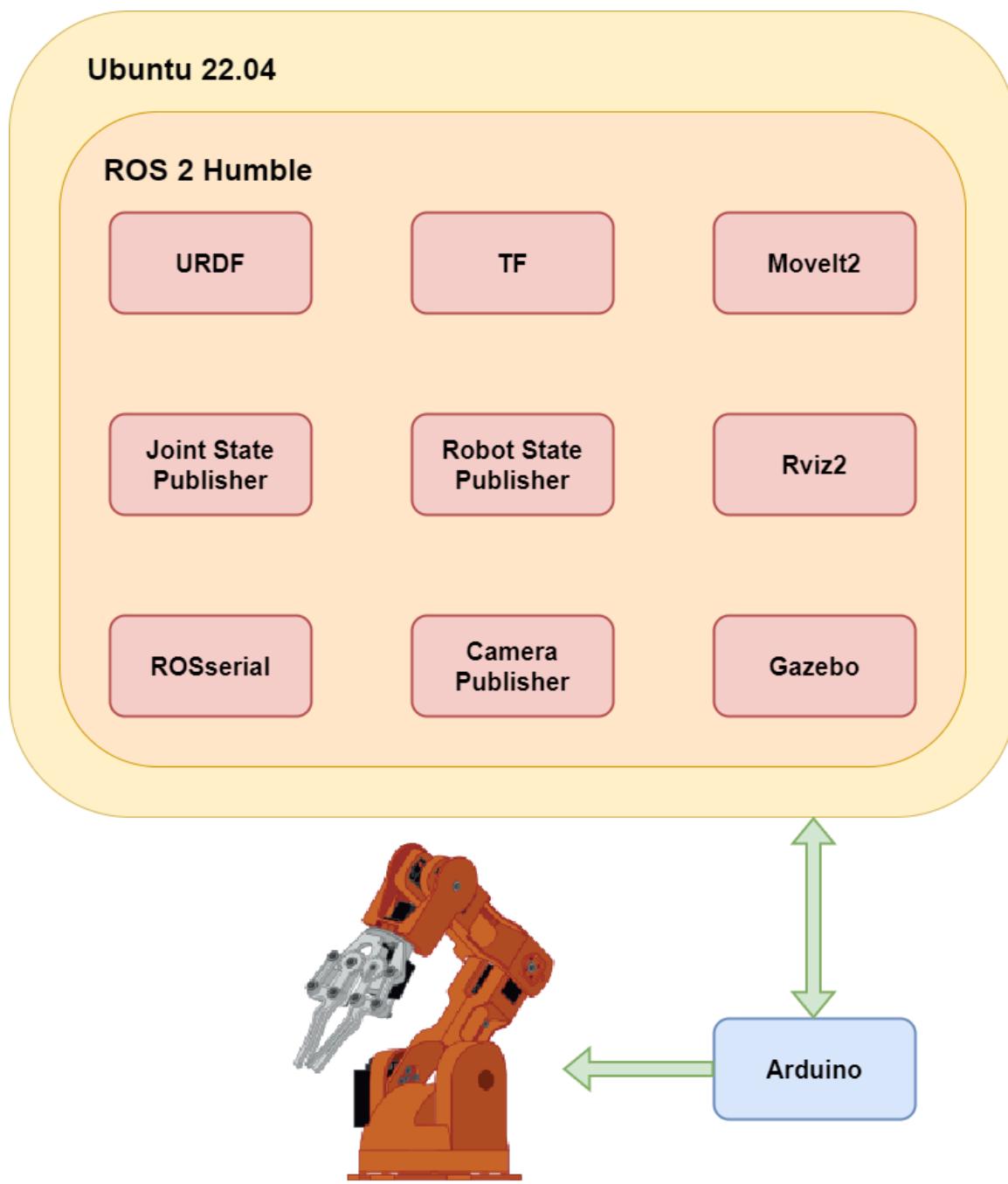


Figura 3.2 Diagrama de flujo del sistema ROS2 con MoveIt2, Gazebo y el resto de herramientas utilizadas para la simulación y control del proyecto.

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].

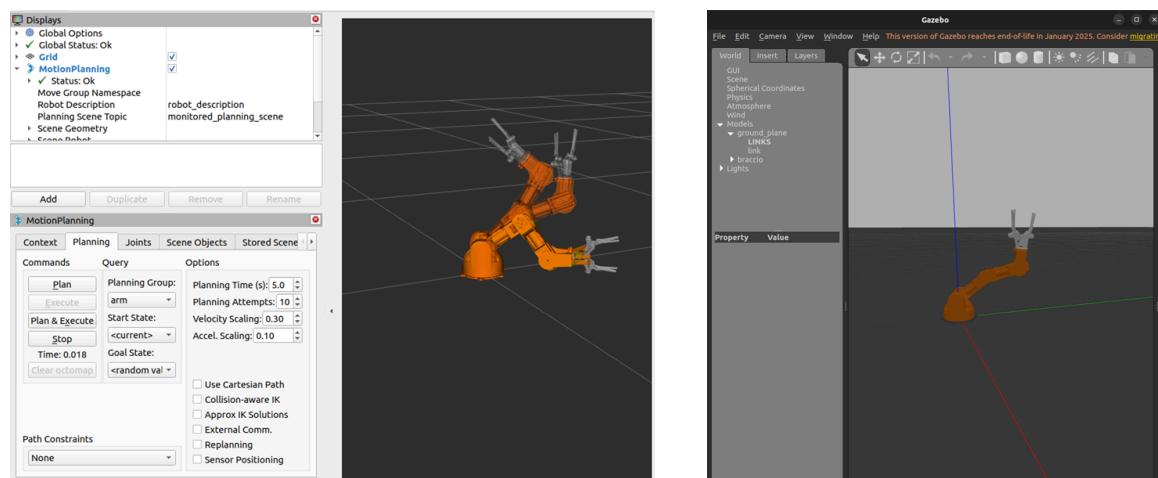


Figura 4.1 Ilustración de la simulación en RViz y Gazebo del robot manipulador realizando una trayectoria. En la izquierda, se puede observar el brazo en sus posiciones inicial, actual y final. A su derecha, la representación de Gazebo mostrando esa posición actual junto al entorno simulado.

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 mediante el 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».

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.

5 Entorno de simulación

5.1 Creación del mundo

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.

5.2 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.

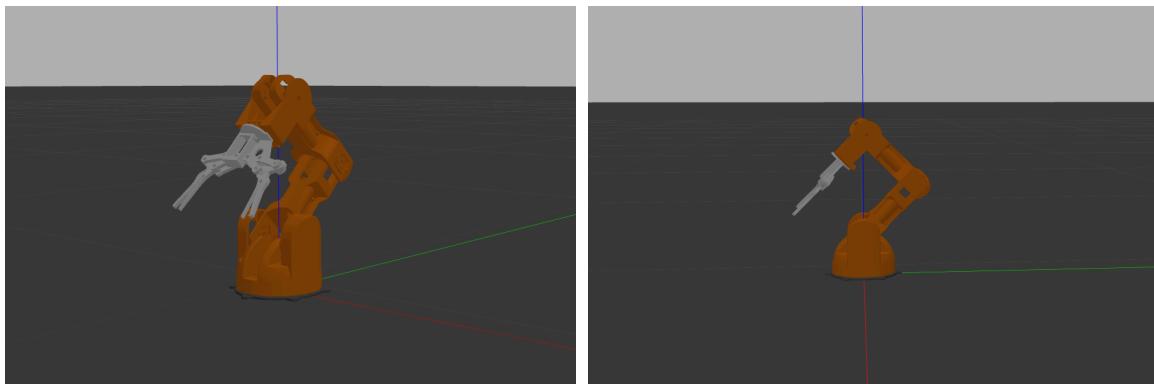


Figura 5.1 Representación del robot Braccio Tinkerkit en Gazebo, mostrando dos perspectivas diferentes del manipulador.

5.3 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.

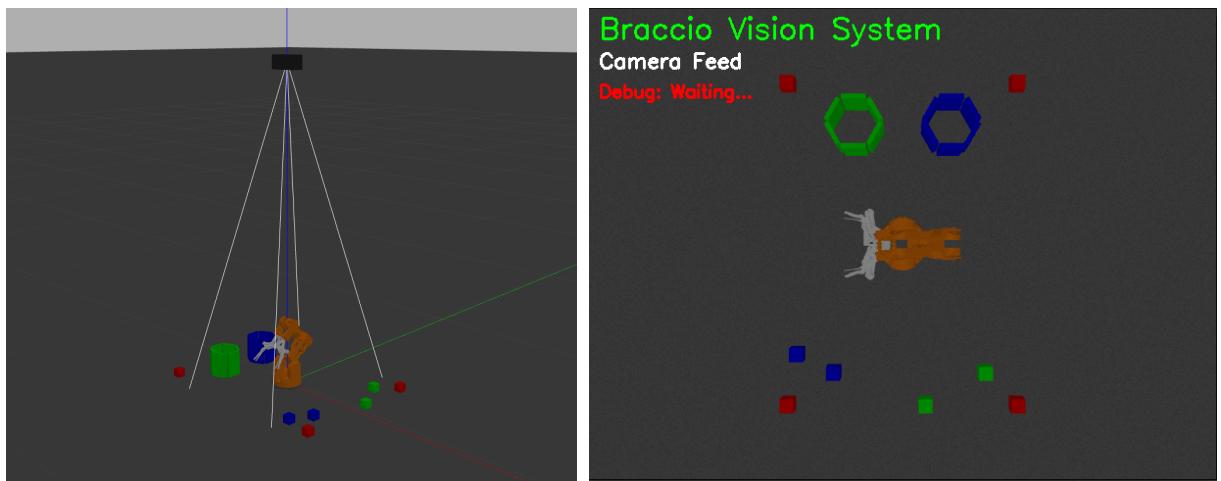


Figura 5.2 Representación completa del entorno de simulación, incluyendo el robot, los recipientes, la cámara y los objetos manipulables. A la derecha, la vista cenital de la cámara simulada 0.6m sobre el suelo, mostrando los cubos de diferentes colores.

6 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.

6.1 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 6.1 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: [1.9540665077487845, 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: [1.9732416837023962, 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=0.97 LY=0.00 RX=0.00 RY=0.00 J4=0.00 G=0.00 → Pos: [1.9924166396558278, 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: [2.02099315112531177, 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: [2.0399414256513111, 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: [2.04944638285804075, 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: [2.058951340649504, 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: [2.06845629724866007, 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 6.1 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.

6.2 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 6.2.

6.2.1 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.

6.2.2 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 6.2 Diagrama del mando de PS4 con el mapeo de los botones y joysticks a las articulaciones y acciones del robot mencionadas previamente.

6.3 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 *gazebo_link_attacher*. 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 6.3
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_cube1', 'green_cube2', 'blue_cube1', 'blue_cube2']
[python3-2] [INFO] [1756212919.32793247] [gamepad_teleop]: Posición gripper TF: (0.18761242586964223, -0.12309485637507234, 0.0537670455547838)
[python3-2] [INFO] [1756212919.32758657] [gamepad_teleop]: Comparando con modelo: zona_hexagono, posición: (0.0, 2.7755575615628914e-17, 0.0)
[python3-2] [INFO] [1756212919.32807567] [gamepad_teleop]: Comparando con modelo: zona_hexagono_azul, posición: (2.7755575615628914e-17, -4.163336342344337e-17, 0.0)
[python3-2] [INFO] [1756212919.32829567] [gamepad_teleop]: Comparando con modelo: overhead_camera, posición: (0.0, 0.0, 1.2)
[python3-2] [INFO] [1756212919.32870909] [gamepad_teleop]: Comparando con modelo: blue_cube1, posición: (0.2307415463118462, 0.2307415463118462, 0.2307415463118462)
[python3-2] [INFO] [1756212919.32891175] [gamepad_teleop]: Comparando con modelo: corner1, posición: (-0.35, -0.25, 0.014999999999755)
[python3-2] [INFO] [1756212919.32911448] [gamepad_teleop]: Distancia a corner1: 0.553746259354382
[python3-2] [INFO] [1756212919.32948199] [gamepad_teleop]: Comparando con modelo: corner2, posición: (0.35, -0.25, 0.014999999999755)
[python3-2] [INFO] [1756212919.32966383] [gamepad_teleop]: Distancia a corner2: 0.2997081866105175
[python3-2] [INFO] [1756212919.32984621] [gamepad_teleop]: Comparando con modelo: corner3: 0.4087448836457087
[python3-2] [INFO] [1756212919.33002774] [gamepad_teleop]: Comparando con modelo: corner4, posición: (-0.35, 0.25, 0.014999999999755)
[python3-2] [INFO] [1756212919.33028668] [gamepad_teleop]: Distancia a corner4: 0.6555377762753594
[python3-2] [INFO] [1756212919.33048668] [gamepad_teleop]: Comparando con modelo: green_cube1, posición: (0.35, 0.0499999999999999, 0.014999999999755)
[python3-2] [INFO] [1756212919.33056732] [gamepad_teleop]: Distancia a green_cube1: 0.240487448836457087
[python3-2] [INFO] [1756212919.33075398] [gamepad_teleop]: Comparando con modelo: blue_cube2, posición: (0.28, 0.18, 0.014999999999755)
[python3-2] [INFO] [1756212919.33075398] [gamepad_teleop]: Distancia a green_cube2: 0.3197253712607857
[python3-2] [INFO] [1756212919.33094329] [gamepad_teleop]: Comparando con modelo: blue_cube1, posición: (0.28003722158545094, -0.1506971845372727, 0.014999999999755)
[python3-2] [INFO] [1756212919.33112725] [gamepad_teleop]: Distancia a blue_cube1: 0.10425522039243953
[python3-2] [INFO] [1756212919.331310125] [gamepad_teleop]: Comparando con modelo: blue_cube2, posición: (0.23976770213997542, -0.24903742142327173, 0.014999274085567619)
[python3-2] [INFO] [1756212919.331530520] [gamepad_teleop]: Distancia a blue_cube2: 0.1417202971999568
[python3-2] [INFO] [1756212919.3317721921] [gamepad_teleop]: Modelo más cercano: blue_cube1, distancia: 0.10425522039243953
[python3-2] [INFO] [1756212919.331968175] [gamepad_teleop]: Pick por proximidad: blue_cube1 (distancia: 0.104m)
[python3-2] [INFO] [1756212919.332289166] [gamepad_teleop]: Intentando attach: blue_cube1/link]
```

Figura 6.3 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*.

7 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. Sin embargo, 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 7.1 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.

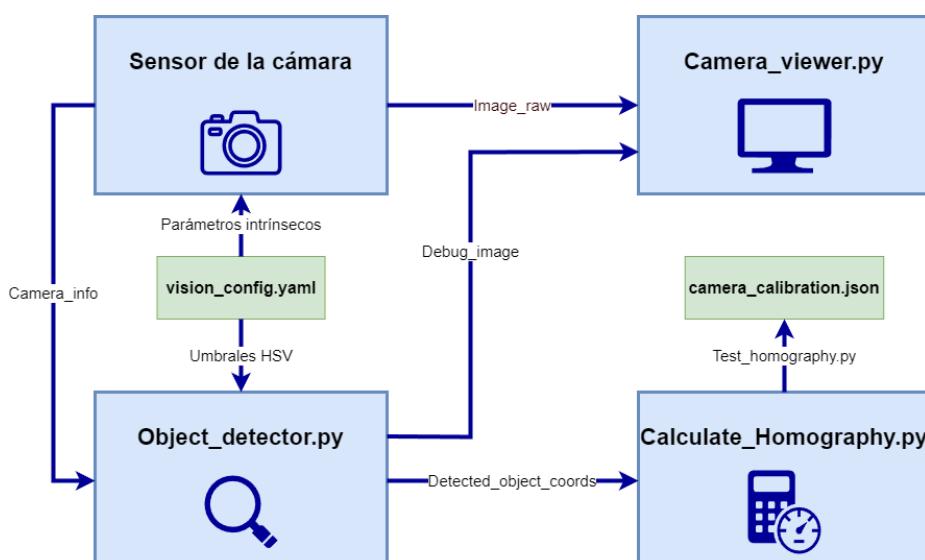


Figura 7.1 Esquema del flujo de percepción y detección de objetos para el sistema de pick-and-place.

7.1 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 5.3, 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 7.2.

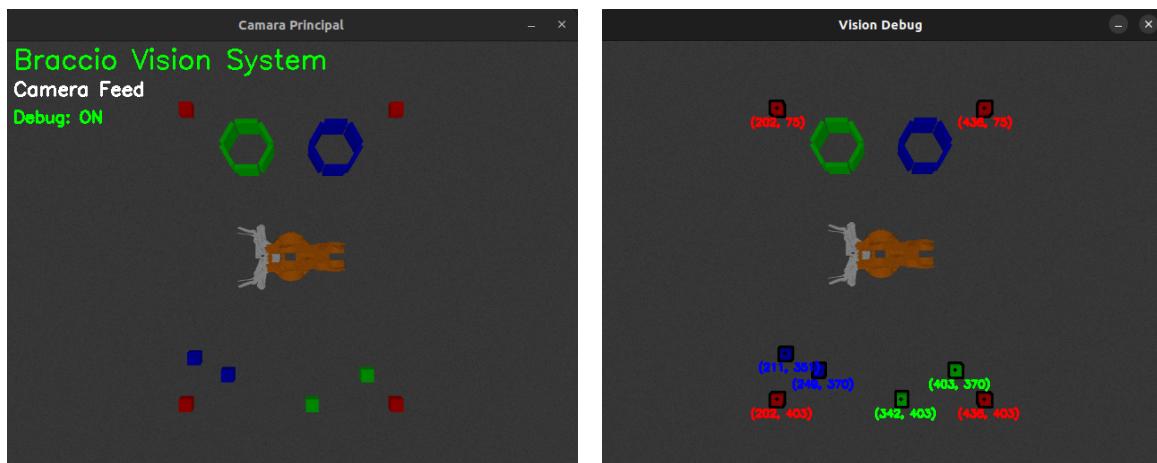


Figura 7.2 Representación de las cámaras simuladas, mostrando la vista del sensor raw y la vista de depuración tras la detección de los objetos. En esta última, se puede observar el marcador de cada objeto detectado junto con las coordenadas de su centro en sus respectivos colores.

7.2 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 7.3, pudiendo ser sustituido posteriormente por la homografía calibrada almacenada en *camera_calibration.json* para aumentar la precisión.

$$X_{\text{norm}} = \frac{\text{pixel}_x - c_x}{f_x}$$

$$Y_{\text{norm}} = \frac{\text{pixel}_y - c_y}{f_y}$$

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

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

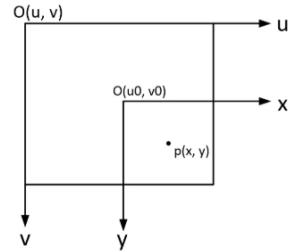


Figura 7.3 Ecuaciones para el cálculo de las coordenadas normalizadas y su proyección al plano, basadas en el modelo geométrico mostrado en la derecha [22]. Los parámetros utilizados se corresponden con la posición del objeto en píxeles ($\text{pixel}_x, \text{pixel}_y$), el centro de la cámara (c_x, 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] [1756322959.832575204] [object_detector]: 🎯 Cubo rojo 4: pixel(202, 75)
[INFO] [1756322959.832760454] [object_detector]: 🎨 CUBOS VERDES DETECTADOS: 2
[INFO] [1756322959.832962455] [object_detector]: 🎯 Cubo verde 1: pixel(342, 403)
[INFO] [1756322959.833161831] [object_detector]: 🎯 Cubo verde 2: pixel(403, 370)
[INFO] [1756322959.833332233] [object_detector]: 🎨 CUBOS AZULES DETECTADOS: 2
[INFO] [1756322959.833497387] [object_detector]: 🎯 Cubo azul 1: pixel(249, 370)
[INFO] [1756322959.833661949] [object_detector]: 🎯 Cubo azul 2: pixel(211, 351)
[INFO] [1756322959.833823185] [object_detector]: 🎯 Cubo verde en pixel(342, 403) temporalmente bloqueado
[INFO] [1756322959.833995150] [object_detector]: 🎯 Cubo verde en pixel(403, 370) temporalmente bloqueado
[INFO] [1756322959.834163840] [object_detector]: 🎯 Cubo azul en pixel(249, 370) temporalmente bloqueado
[INFO] [1756322959.834369217] [object_detector]: 🎯 Cubo azul en pixel(211, 351) temporalmente bloqueado
[INFO] [1756322959.834553264] [object_detector]: 🎯 Cubo azul en pixel(211, 351) temporalmente bloqueado
```

Figura 7.4 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 7.2. Los cubos temporalmente bloqueados son aquellos destinados a la manipulación en el entorno simulado, estudiados en secciones posteriores.

7.3 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 7.2, obteniendo las posiciones indicadas en la Tabla 7.1.

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 7.1 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 7.5 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 7.5 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) | Píxel 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, 403.3) | 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 7.6 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.

8 Planificación de agarre y manipulación

8.1 Cinemática directa e inversa

8.2 Repositorio attach/detach

8.3 Transferencia sim-to-real y validación experimental

Técnicas para reducir la brecha: domain randomization, calibración de cámara y brazo, system identification y HIL. Diseño experimental para comparar simulación y prototipo real, incluyendo métricas y repetibilidad.

9 Evaluación y métricas

Métricas recomendadas: tasa de éxito del pick-and-place, error de posicionamiento, tiempo por ciclo, repetibilidad y robustez ante variaciones de objeto/escenario. Protocolo de pruebas y análisis estadístico básico.

10 Huecos detectados y oportunidades para el TFG

Resumen de cuestiones poco cubiertas en la literatura: modelos dinámicos de alta fidelidad para robots educativos, pipelines integrados Arduino+micro-ROS+MoveIt2, datasets para piezas educativas; propuestas: modelado URDF/SDF del Braccio, pipeline RGB-D para pose, integración y validación sim-to-real.

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
5.2.	Figura 5.1 Representación del robot Braccio Tinkerkit en Gazebo, mostrando dos perspectivas diferentes del manipulador.	17
5.3.	Figura 5.2 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.	18
6.1.	Figura 6.1 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
6.2.2.	Figura 6.2 Diagrama del mando de PS4 con el mapeo de los botones y joysticks a las articulaciones y acciones del robot mencionadas previamente.	20

- 6.3. **Figura 6.3** 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
7. **Figura 7.1** Esquema del flujo de percepción y detección de objetos para el sistema de pick-and-place. 23
- 7.1. **Figura 7.2** 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
- 7.2. **Figura 7.3** 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
- 7.2. **Figura 7.4** 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 7.2. Los cubos temporalmente bloqueados son aquellos destinados a la manipulación en el entorno simulado, estudiados en secciones posteriores. 25
- 7.3. **Figura 7.5** Lectura del terminal de Ubuntu tras la ejecución de *Calculate_homography.py*. Representa la matriz de homografía calculada . 26
- 7.3. **Figura 7.6** 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 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

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
7.3.	Tabla 7.1 Posiciones de los cubos rojos en el entorno simulado y sus centroides capturados por la cámara en píxeles.	26

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%26Robotics&sortBy=>
- [11] Arduino, «Braccio Tinkerkit». [En línea]. Disponible en: https://store.arduino.cc/products/tinkerkit-braccio-robot?srsltid=AfmBOop3_QGF-ekRRLRKSMZ3rJ3aKQ2ERPQLrD1_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 (branch: prueba)». [En línea]. Disponible en: <https://github.com/Ivan-Luque-Valverde/Braccio-Tinkerkit-Arduino/tree/prueba>
- [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>

