Universidad de Oviedo

ESCUELA POLITÉCNICA DE INGENIERÍA DE GIJÓN.

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

ÁREA DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

CLASIFICACIÓN DE BOLAS MEDIANTE UN BRAZO ROBÓTICO UTILIZANDO ROS (ROBOTIC OPERATING SYSTEM) Y TÉCNICAS DE VISIÓN POR COMPUTADOR

D. Lucas Rey Braga
TUTOR: D. Rafael Corsino González de los Reyes

FECHA: Mayo 2021

Resumen

Este trabajo pretende la integración de un brazo robótico educativo mediante la utilización del sistema operativo ROS (Robotic Operating System) en la planta piloto de la Industria 4.0 de la Escuela Politécnica de Ingeniería de Gijón de Universidad de Oviedo, sentando así las bases de un sistema de control para la clasificación de canicas en un contexto de fabricación de *spinners*.

En concreto, se ejemplifica dicha labor de clasificación mediante una demostración con círculos de dos colores. Estos círculos serán localizados en el plano de la mesa, moviendo el robot a los lugares detectados para cada ficha, simulando la recogida y el posterior transporte a la posición de los depósitos. Los dos colores pretenden reducir las características propias de los materiales para la simulación de la tarea.

Dentro de la integración en ROS se diseñarán las piezas en la simulación y se crearán los nodos que compondrán todo el sistema de control del brazo robótico: los nodos encargados de la simulación, de la percepción, del accionamiento y de la manipulación del sistema. Además, se aplicarán técnicas de visión por computador, utilizando la imagen capturada por la cámara, que nos permitirán el reconocimiento y localización de los círculos de colores. Esto dotará al brazo robótico de las herramientas necesarias para el *pick and place* (la acción de manipulación demandada por la planta) dentro de la filosofía propuesta por la Industria 4.0.

El trabajo demuestra las capacidades inherentes de ROS, con la implementación de algunas de sus funciones, las cuales representan solo una pequeña parte de las capacidades futuras que se esperan. Se muestran las virtudes de la modularidad y versatilidad que caracterizan a ROS, permitiendo así realizar la computación en diferentes dispositivos y pudiendo adaptarse a la conectividad buscada para la investigación e implementación de tecnologías IoT (*Internet of Things*) dentro de la planta.

El control se realizará mediante el uso de una placa Arduino que se comunicará con una Raspberry mediante puerto serie, utilizando las herramientas de ROS. También se utilizará un ordenador, que se comunicará con la Raspberry mediante la utilización de una

red local WiFi; en dicha comunicación el ordenador será el encargado de correr el ROS Master. Desde el ordenador, se generarán las trayectorias con la utilización de *Moveit!* partiendo de la detección de los círculos. La información obtenida de esas imágenes se transforma en posiciones angulares y se envían al Arduino, que posicionará los servos comunicándose con el driver mediante I2C. Las trayectorias generadas se podrán visualizar simultáneamente tanto en el robot real como en la simulación.

Finalmente, se mostrarán y analizarán los resultados obtenidos en el desempeño de la labor de clasificación, el funcionamiento de la aplicación de percepción y el estudio de sus límites. Esto se realizará mediante pruebas relacionadas con la comprobación de las distancias medidas en la percepción, la situación de la cámara en diferentes posiciones y la simulación del peso de una garra.

Tabla de Contenido

1. Motivación, objetivos y alcance

1.1.- MOTIVACIÓN

La motivación de este trabajo es la de integrar un brazo robótico de bajo coste en la planta piloto de la Industria 4.0 de la Universidad de Oviedo. Su objetivo dentro de la planta será la clasificación de canicas según su material para la fabricación de *spinners*. Dicha planta piloto, dependiente del Área de Ingeniería de Sistemas y Automática, ha puesto en marcha un proyecto de fabricación de *spinners* que sirve como plataforma para la formación del estudiantado en tecnologías IoT (*Internet of Things*) y de vanguardia, tratando de implantar este nuevo modelo de producción. En este proyecto, se busca la integración en la planta de un brazo robótico, capaz de realizar la labor de clasificación solicitada por la planta. Esta integración se lleva a cabo en ROS y se ejemplifica, debido a la situación provocada por el COVID-19 y las dificultades para acceder a la planta, mediante una demostración de una tarea de *Pick & Place* con círculos de colores; realizada con el robot en casa en vez de con las canicas originales de la planta.

ROS ofrece un gran número de características que se consideran interesantes para distintas utilidades dentro de la Industria 4.0 y que lo hacen muy versátil. Una de las características que ROS ofrece es la posibilidad de reutilización de diversos paquetes, dedicados a una función, en diferentes aplicaciones. Esto implica que los paquetes desarrollados en ROS pueden utilizarse fácilmente para cometidos similares, con pequeñas modificaciones de código, dentro de la estructura de otro sistema robótico. ROS también nos permite el funcionamiento simultáneo en diferentes dispositivos, lo cual lo hace interesante para la coordinación simultánea de diferentes sistemas o componentes robóticos lanzados desde diferentes dispositivos. Todo ello sin contar con la gran cantidad de información y de ayuda que se encuentra disponible en la comunidad de ROS, además de la posible integración y coordinación de gran variedad de hardware robótico dentro de la planta (robots móviles, interacción con otros brazos robóticos, robots de tipo colaborativo y otros).

1.2.- OBJETIVOS

Los objetivos del TFG son los siguientes:

- **Descripción cinemática del robot:** cálculo de las ecuaciones que establecen la relación entre la posición de los actuadores y la posición del efector final.

- **Diseño del robot para su simulación:** utilización de modelos realizados por CAD para la descripción del robot mediante un modelo especial de archivos de formato XML llamado URDF. El robot será simulado en el entorno de Gazebo.

- **Integración del brazo robótico en ROS:** idear un sistema de control utilizando las diversas herramientas que nos ofrece ROS, programándolo y estructurándolo en bloques modulares llamados nodos. Esto implicará la utilización de microcontroladores para el control de los servomotores mediante la utilización de un driver específico y la aplicación de librerías de Arduino.

- **Integración de un sistema de percepción en un robot real:** aplicación de algoritmos de percepción en un sistema robótico real, transformando una imagen en una información geométrica útil para el cometido del brazo robótico.

- **Puesta en funcionamiento del robot mediante una función básica de demostración.**

1.3.- ALCANCE

La automatización de los procesos de fabricación a través de la robótica es uno de los rasgos definitorios de la transformación digital que implica la industria 4.0. Este trabajo pretende la integración de un brazo robótico educativo en la planta piloto de la Industria 4.0 de la Universidad de Oviedo. Se ha utilizado ROS para programar un sistema de control robótico que pueda ser utilizado en la clasificación de canicas en la fabricación de spinners. Debido al COVID-19, la demostración de la tarea se realizó con círculos de colores en una mesa, aplicando técnicas de visión por computador, accionando los motores y situando las articulaciones en función de los cálculos realizados en la cinemática inversa.

Los resultados del trabajo esperan contribuir al avance hacia una labor automatizada de clasificación de canicas, que solicita la planta piloto. Se pretende que, con el sistema creado, se disponga de una base de generación de trayectorias y de control necesaria para integrar un nuevo programa de visión, aplicado al problema real de las canicas y, por lo tanto, adaptado a las condiciones de funcionamiento reales de la planta. También se espera que, a la hora de adaptar el sistema a las condiciones reales, no se tengan que realizar grandes cambios, que el sistema se adapte lo máximo posible y, por lo tanto, el código sea reutilizable para futuras investigaciones.

La estructura del trabajo parte de una revisión del estado del arte atendiendo especialmente a proyectos de tipo open-source y de la Industria 4.0 para, a continuación, describir los principales conceptos y herramientas utilizados dentro de la programación en el sistema robótico desarrollado en ROS. El Capítulo 4 describe la cinemática del brazo robótico junto a la descripción de todo el hardware empleado. A partir del Capítulo 5, se describe el diseño del algoritmo de percepción del robot (explicando los conceptos de visión utilizados) para, en el Capítulo 6, describir el sistema de control robótico diseñado. Se finaliza el trabajo comprobando el funcionamiento de las labores del robot y analizando los resultados obtenidos en diferentes pruebas.

2. Estado del arte

Actualmente, la robótica, al igual que muchas áreas relacionadas con ella, como pueden ser la visión por computador o el *Machine Learning,* están en un momento de gran auge. Esto se debe, en gran parte, a la difusión e impulso de nuevas técnicas robóticas a raíz de la proliferación de plataformas innovadoras dentro de las cuales ROS juega un papel fundamental. ROS se ha constituído como una enorme *framework open-source* en la que tanto expertos como aficionados a la robótica contribuyen a expandir el conocimiento, proporcionando nueva información útil para futuros desarrolladores.

2.1.- PROYECTOS DE RELEVANCIA OPEN-SOURCE

Sin duda, la robótica sigue siendo un nicho en constante crecimiento en cuanto a proyectos de investigación se refiere. Cabe señalar a ROS como el máximo proyecto de código abierto en materia robótica de la actualidad, con una colaboración de miles de desarrolladores y haciendo presencia en el panorama internacional como *framework* sobre la que cualquiera puede desarrollar su proyecto robótico. Existen hoy en día, eficaces grupos de trabajo que, en colaboración con la comunidad de ROS y la participación de pequeñas empresas, logran la creación de proyectos muy interesantes que permiten el acceso a muchísimos más desarrolladores. Se presenta a continuación una selección de proyectos de código abierto que definen muy bien tanto a ROS como al momento de la robótica que vivimos actualmente.

2.1.1.- TURTLEBOT3

Turtlebot, el robot móvil de la empresa Robotis, se ha convertido en uno de los más utilizados por cualquiera que se esté iniciando en el campo de la robótica o que quiera desarrollar su propio robot móvil. Esto se debe a que es un robot totalmente libre y muy bien documentado, que ha ido creciendo con la ayuda de sus creadores y que múltiples usuarios se han sumado a su desarrollo. Gracias a la filosofía *open-source*, Turtlebot es uno de los robots que más se construyen y que son más simulados alrededor del mundo. En muchos

casos, está presente en las casas de desarrolladores de software robótico a nivel profesional; en otros casos, se encuentra en las de muchos aficionados *hobby-makers*.

Su última actualización nos ofrece dos modelos: el modelo Burger y el Waffle. Ambos ofrecen los mismos servicios de navegación, mapeo y localización, con la posibilidad de utilizarlos para servicios domésticos básicos.

Para entender la idea con la que se inventa este robot, tenemos que remontarnos al primer Turtlebot, que fue desarrollado en uno de los primeros lenguajes de programación existentes: Logo. ROS tomó el testigo de este lenguaje para la creación de su clásico tutorial para principiantes Turtlesim, en el que enseña cómo crear un sistema de nodos alrededor de un programa muy parecido al utilizado en Logo. Utilizando la tortuga que se mueve en dicho programa de iniciación, ROS tomó como icono emblema del sistema robótico los seis puntos del caparazón de la tortuga, que se encuentran en el lado izquierdo del logo de ROS.

Los creadores de Turtlebot tenían en su cabeza la idea de crear un robot con el que el principiante pudiera empezar a trabajar, en simulación o en la realidad, con un hardware robótico específico. Esto lo querían realizar de manera parecida a la que se había realizado en Turtlesim, convirtiéndose de esta forma en la plataforma hardware standard de ROS en la actualidad [1].

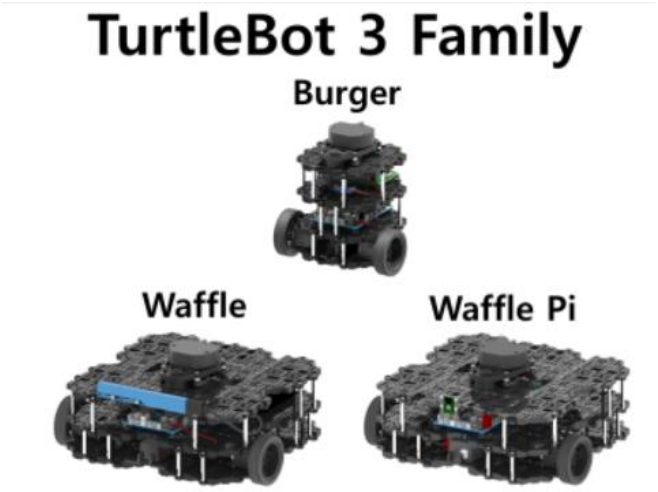


Figura 2.1 - Modelos robots móviles Turtlebot3.

2.1.2.- NATURAL MACHINE MOTION INITIATIVE (NMMI)

Actualmente, según se va desarrollando una robótica más puntera, surge la necesidad de desarrollar sistemas que hagan que los movimientos del robot sean más naturales, más cercanos a los movimientos humanos y animales, con el objetivo de hacer a los robots más hábiles y con movimientos más fluidos.

Dentro de esta línea de investigación, llamada en inglés *Soft Robotics*, surge la plataforma NMMI, que dispone de herramientas para diseñar y construir articulaciones robóticas, así como para el desarrollo de prototipado rápido. Muchas de estas articulaciones van a ser utilizadas en el futuro por robots de ROS y, si sigue desarrollándose, puede que llegue a formar parte del sistema operativo. Esta línea de trabajo está enfocada al desarrollo de un comportamiento robótico que pueda ser compatible con la convivencia con el ser humano, haciendo sus movimientos mucho más seguros. En la actualidad, muchos de los robots industriales e incluso humanoides, presentan potenciales riesgos a la hora de interactuar con el ser humano. Además, el campo de *Soft Robotics* tiene una aplicación directa en cualquiera de las ramas de la robótica actual, desde los robots humanoides hasta los clásicos robots manipuladores en la industria [2].

Un ejemplo del desarrollo de esta tecnología puede ser el robot serpiente, que, gracias a su movimiento en forma de serpenteo, imitando así el de una serpiente, puede pasar por recovecos de difícil accesibilidad con un buen desempeño en su labor. Actualmente, se está utilizando en limpieza de tuberías, aunque cabe pensar que tendrá un papel importantísimo en labores de salvamento y exploración de cavernas o accidentes geográficos, donde el ser humano tiene un difícil acceso [3].

Figura 2.2.- Robot tipo serpiente.

2.1.3.- DESARROLLO DE ROBOTS CUADRÚPEDOS

Los robots cuádrupedos se están popularizando mucho por las cualidades que ofrecen para la interacción con el medio. Son robots de estabilización fácil, con múltiples utilidades posibles y que están alcanzando un nivel de desarrollo importante que los sitúan entre los robots más llamativos del mercado. Esto es debido a su gran versatilidad y estabilidad para la realización de todo tipo de labores.

Existen grandes ejemplos en el mundo open-source de robots cuadrúpedos, entre los que podemos destacar el ANYmal Robot cuadrúpedo, del que podemos encontrar su descripción libre en el repositorio Github de la marca [4], listo para simular en ROS. Las últimas investigaciones realizadas con este robot van enfocadas a la incorporación de ruedas para hacerlo más estable en cualquier tipo de terreno, lo cual está dando resultados muy esperanzadores [5].

Figura 2.3.- Robot cuadrúpedo ANYmal C1.

2.1.4.- AWS ROBOTICS, LA APUESTA DE AMAZON EN LA ROBÓTICA OPEN-SOURCE

Para finalizar este apartado, cabe señalar la entrada de Amazon dentro del sector de forma privada. Amazon ha abierto una interfaz llamada RoboMaker dentro de sus servicios en la nube AWS [6]. Esta aplicación utiliza como base el sistema operativo ROS, aportando una serie de extensiones en la nube que nos permiten una mayor conectividad con robots y su control en tiempo real, así como el uso de todos ellos mediante la utilización de tecnologías IoT (*Internet of Things*) y la nube, ofreciendo además una capa de virtualización en la simulación para comunicarse con los dispositivos utilizados [7]. Amazon, ya desde hace tiempo, se está volviendo un referente de las tecnologías en la nube, mostrando una gran variedad de herramientas en el desarrollo profesional de aplicaciones en empresas tecnológicas. Robomaker AWS ofrece los siguientes servicios en la nube [8]:

- Amazon Kinesis (transmisión de video)
- Amazon Rekognition (análisis de imagen y video)
- Amazon Lex (reconocimiento del habla)
- Amazon Polly (generación del habla)
- Amazon CloudWatch (registro y monitoreo) para desarrolladores que utilizan el Sistema Operativo Robótico (ROS).

Además, esta plataforma permitirá la monitorización de cada equipo para controlar la actualización de todos los dispositivos, así como para posible monitorización de fallos o mal funcionamiento en el caso de la utilización de flotas de robots en grandes almacenes o industrias en tiempo real.

En esta línea, Amazon también ha sacado el AWS Racer, un robot móvil y autónomo para desarrolladores principiantes que empiezan a trabajar con el aprendizaje automático y la robótica.

Figura 2.4.- AWS Racer.

2.2.- INDUSTRIA 4.0

La industria 4.0 supone un cambio de paradigma con respecto a la forma de producción resultado de la revolución industrial del siglo XX. Este cambio de paradigma supone la aplicación de las tecnologías de la información y el Big Data en la producción industrial. Se utilizan técnicas que permiten una mayor conectividad: trabajando con grandes cantidades de información e interactuando con la maquinaria industrial. La utilización de estas técnicas ha permitido la automatización de un mayor número de procesos industriales, así como la aplicación directa de tecnologías de vanguardia. Entre estas tecnologías se pueden encontrar: la inteligencia artificial (IA), las técnicas de visión por computador, mecanismos de comunicación basados en las tecnologías IoT (*Internet of Things*) o el acceso a la nube para el procesamiento de grandes cantidades de información (*Big Data*).

Todas estas tecnologías forman en su conjunto sistemas conocidos como ciber-físicos o *Cyber-Physical Systems* (CPS), capaces de llevar el control de una planta productora y coordinar la producción, conectando entre sí toda la maquinaria y ofreciendo información a tiempo real para el ser humano, lo cual estrecha y facilita la relación hombre-máquina [9]. Entre las tecnologías más habituales que podemos encontrar en la industria 4.0, y que se están utilizando con mayor asiduidad hoy en día, están las siguientes [10]:

- IoT para la industria
- Robótica colaborativa
- Gemelos digitales
- Soluciones de visión inteligente
- 5G y redes WiFi

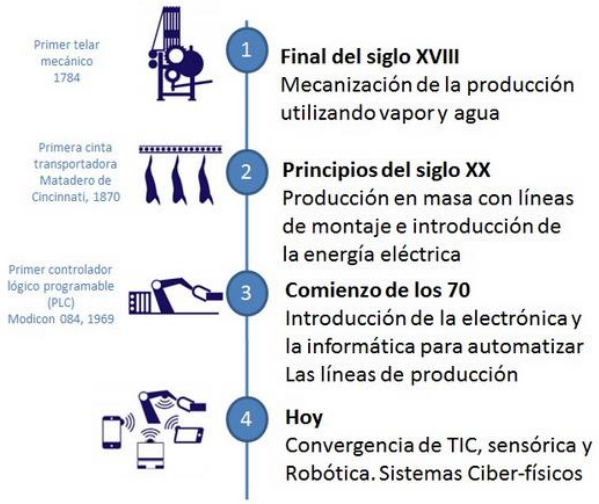


Figura 2.5.- Diagrama comparativo de las diferentes revoluciones industriales [11].

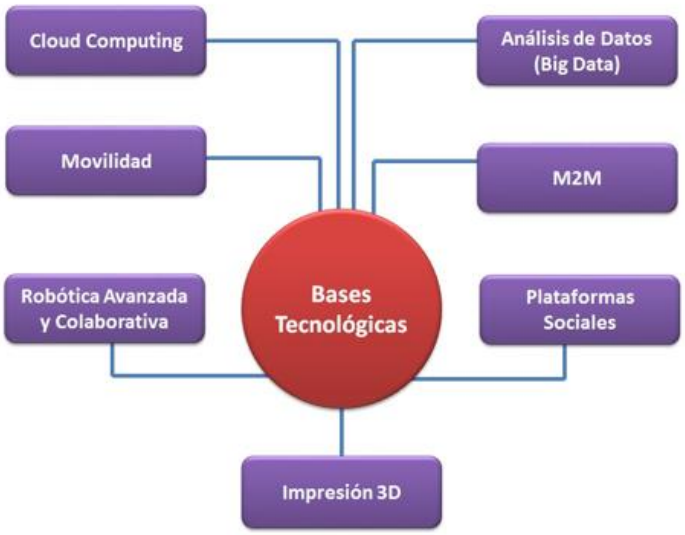


Figura 2.6.- Diagrama de tecnologías que participan en la Industria 4.0 [11].

Muchas de estas tecnologías están siendo utilizadas en la planta para la que va destinado el robot con el que se trabaja en este proyecto. En esta planta, dedicada a la investigación de la industria 4.0, se pretende implantar un sistema de producción utilizando las tecnologías de vanguardia a las que nos referimos. Entre las tecnologías ya implementadas están: la aplicación de conexiones modbus con servidores remotos (IoT), la conexión SCADA server con la utilización de una comunicación via WiFi, el desarrollo de un gemelo digital o la implantación de robots de tipo colaborativo.

Se trabaja para que, en un futuro, todos estos procesos puedan ser controlados en tiempo real desde cualquier parte, con acceso a una conexión a internet, aunque en la actualidad ya trabajan con una buena autonomía. Podemos ver una representación de la planta en la Figura 2.7, en la que se enumeran todas las tecnologías y la maquinaria que están siendo utilizadas.

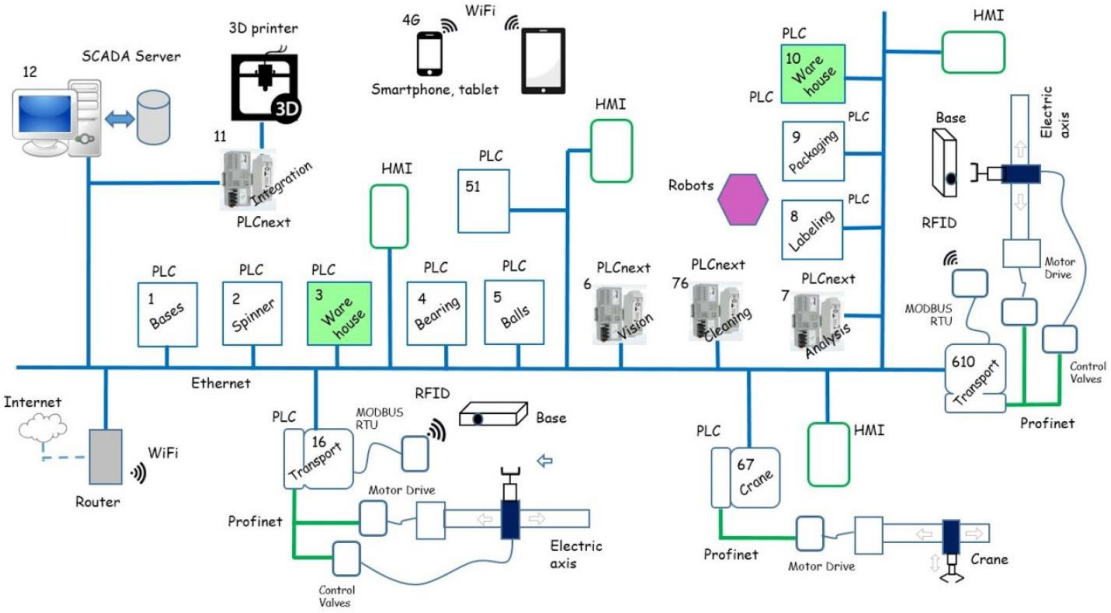


Figura 2.7.- Esquema de la planta de Industria 4.0 de la Universidad de Oviedo.

Se pueden encontrar implementaciones muy interesantes de las tecnologías utilizadas dentro la industria 4.0, aplicables en ROS. ROS recoge y abraza todas estas tecnologías siendo muy útil para la creación de módulos de expansión IoT para sistemas robóticos estándar. Podemos observar un buen ejemplo de ese tipo de integración en el proyecto de la empresa Roboteurs sobre el lanzamiento de nodos en la nube con una comunicación MQTT,

dentro de su repositorio de Github. En este proyecto, se implementa un módulo de integración en forma de nodo de ROS, dentro de una comunicación de tipo MQTT. A diferencia de la comunicación tradicional de ROS, dentro de un mismo equipo o una misma red local, permitiría una comunicación utilizando la nube como servidor, ampliando las posibilidades de transmisión de información a la nube [12].

Este tipo de tecnologías IoT están haciéndose cada vez más demandadas dentro de la robótica y se están abriendo muchas fronteras con la aparición de ROS 2.0, como la actualización ya existente del sistema operativo robótico y que está permitiendo ya desde hace tiempo una mayor adaptabilidad del sistema operativo dentro de la industria [13].

En lo que a este trabajo se refiere, nuestro brazo robótico se encontraría clasificado dentro de la robótica open-source, con el diseño de un algoritmo para la aplicación de visión artificial dirigida a la clasificación y localización de canicas utilizándolas en la producción de *spinners*.

2.3.- PROYECTOS DE BRAZOS ROBÓTICOS

Dentro de la robótica, y acercándonos al objeto de este trabajo, encontramos los brazos robóticos. Los brazos robóticos se pueden describir como conjuntos de eslabones unidos mediante articulaciones, movidos mediante la acción de motores. Mediante el control de la posición de las articulaciones, este tipo de robots logra situar dentro de un espacio de trabajo determinado, un efector final en las coordenadas deseadas. Estas características permiten la manipulación y movimiento de objetos entre posiciones para el desempeño de labores dentro de una función industrial. Entre dichas labores podremos encontrar, por ejemplo, el conocido *pick and place*, una labor de recogida y depósito como la que se realiza en este proyecto.

Históricamente, los brazos robóticos de tipo industrial, estaban pensados para ser máquinas grandes que desarrollaban labores fijas y casi inamovibles, para las que necesitaban ser calibrados específicamente dentro de un mismo lugar de trabajo. Este tipo de robots se sigue utilizando en las industrias y su diseño está enfocado a la producción en

masa, debido a su solidez, efectividad y precisión en labores repetitivas de grandes cadenas de montaje. No obstante, esta solidez y efectividad, se desvanece en el momento en el que algún fallo sucede en la cadena de montaje, que hace frenar toda la producción.

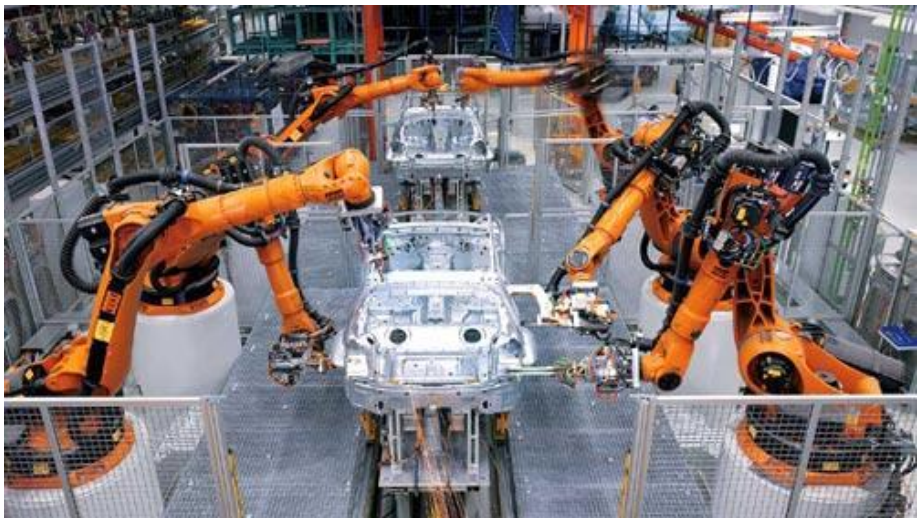


Figura 2.8.- Brazos robóticos industriales en cadena de montaje.

Durante los últimos años, la investigación alrededor de los brazos robóticos, y de los robots en general, se ha desvinculado de esa producción en masa y ha trascendido hacia el desarrollo de técnicas y métodos que hacen a estos robots más autónomos, flexibles y reconfigurables. Este cambio de enfoque se debe a la aparición de los brazos robóticos en la pequeña y mediana industria, interesada en la producción en menores cantidades, con el propósito de alcanzar precios competitivos en el mercado. Esta evolución también tiene que ver con la aparición de la Industria 4.0 y la integración de tecnologías de vanguardia a los nuevos modelos de robot.

Por lo tanto, se está hablando actualmente de una transición desde la robótica industrial tradicional, con robots tremendamente especializados y que seguirá vigente sobre todo en grandes cadenas de montaje, hacia la implementación de sistemas de producción reconfigurables (Reconfigurable Manufacturing Systems o RMSs). Estos últimos ofrecerán cambios rápidos, tanto en el software como en el hardware, adaptándose a diferentes tipos de procesos. En conjunto con la implementación de sistemas flexibles de producción (Flexible Manufacturing Systems o FMSs), logrará el cambio de producción del lote de productos casi de forma automática [14].

Dentro de este desarrollo, y en las nuevas líneas de investigación asociadas a él, juega un papel muy importante la aparición de brazos robóticos de tipo open-source y educativos, que garantizan la difusión de la robótica y la potenciación de la investigación en la búsqueda de técnicas que avancen hacia la reconfigurabilidad y flexibilidad de estos robots de tipo industrial. Estos robots, aunque no constituyan el hardware utilizado en empresas (puesto que la calidad de los robots industriales es muy superior) han constituido un cambio enorme en el avance hacia sistemas robóticos más inteligentes [14][15].

Dentro de los robots open-source podemos encontrar sistemas de control de brazos robóticos desarrollados en ROS. Sin duda, han sido fuente de inspiración y muy buenas referencias para el diseño de la implementación basada en ROS mostrada en este trabajo. Entre ellos podemos encontrar:

- **uArm** es un robot de código abierto muy similar al que se utiliza en este proyecto. Ofrece una gran cantidad de información en su repositorio acerca de la programación robótica dentro de ROS, así como un modelo URDF del robot con su consiguiente representación de las piezas en archivos de tipo collada *.dae*. También ofrece numerosas aplicaciones de percepción basadas en la segmentación de la imagen para la separación de objetos, además de diversos scripts en Python, en los que se implementan técnicas de *pick and place* y de accionamiento de los motores del brazo [16].

- Otro robot de referencia es un proyecto muy ambicioso cuyo objetivo fue la construcción de un robot móvil con un brazo robótico DIY (*Do It Yourself* o *Hágalo usted mismo*). Combina simulación con la gestión del *feedback* proporcionado por sensores y cámara 3D, ayudando al desarrollo de este trabajo en la comprensión del funcionamiento de un sistema robótico en su conjunto en ROS. El creador ofrece una solución basada en una interfaz hardware C++ que se comunica con el robot, ofreciendo un *feedback* con la información aportada por los sensores [17].

- **Robo ND-Kinematics-Project** es un proyecto de la empresa Udacity basado en el robot industrial KUKA KR210. Combina una buena explicación sobre la aplicación de cinemática inversa mediante scripts de Python, con un planteamiento teórico sobre el cálculo de la cinemática inversa y directa muy didáctico. Dicho proyecto está realizado en ROS y, aunque esté ideado para entender la cinemática, es un buen recurso y ejemplo de modelado y simulación de robots con un sistema básico de control de posición [18].

- **Nyrio One** es un robot 6 DoF (*Degrees of Freedom*) estándar, con un diseño similar al de los robots industriales reales pero utilizado, debido a la cantidad de información vigente en su web, con un enfoque claramente educacional. Ofrece, a su vez, un repositorio Github en el que se encuentra el código en ROS, que permitirá controlar el robot real desde el sistema operativo, aunque existe también la posibilidad de experimentar con el robot en una simulación en Gazebo [19]. Este robot está diseñado para trabajar de forma remota desde el ordenador, funcionando en diferentes dispositivos y distribuyendo la computación, tal y como se realiza en este trabajo. Además, tiene sensores de posición y ofrece opciones de robótica colaborativa, pudiendo interactuar con el ser humano para su calibración, por ejemplo. Por todas estas razones, se vuelve una opción verdaderamente interesante en la Industria 4.0, pudiendo cumplir muchos de sus requisitos de trabajo [20].

3. ROS (Robotic Operating System)

ROS es un sistema operativo robótico de fuente abierta. En él participan múltiples desarrolladores alrededor de todo el mundo y constituye uno de los entornos standard de programación de sistemas robóticos, además de una de las herramientas más interesantes sobre las que desarrollar una investigación robótica actualmente. Además, miles de desarrolladores participan día a día actualizando y mejorando el sistema, tal y como se evidencia observando los repositorios asociados a ROS y ROS 2.0 en Github, así como en los repositorios de las herramientas que forman parte del sistema operativo robótico, como pueden ser *MoveIt!* o ros_control.

3.1.- HISTORIA DE ROS

El inicio de ROS se sitúa en la Universidad de Stanford, donde Keenan Wyrobek y Eric Berger, dos investigadores de la universidad, se dieron cuenta del escaso avance en la consecución de sistemas robóticos más inteligentes, lo que originaba un completo estancamiento en la robótica.

Así en 2006 crean un programa llamado Stanford Personal Robotics Program con el propósito de crear una *framework* común sobre la que poder trabajar junto con otros desarrolladores, contando también con la utilización de herramientas específicas para el desarrollo de sistemas robóticos. Otro de los propósitos era crear una red de contacto entre diversas universidades para lo que se crearon 10 robots idénticos, los *Personal Robots* (PR), que constituirían el primer ejemplo de implementación de un sistema robótico completo en el nuevo programa, así como la prueba sobre un robot real de todas las herramientas que habían sido desarrolladas. Algunas de las herramientas, que surgieron de esos primeros trabajos, fueron las formas de comunicación propias de ROS (la librería ros_comm), Rviz o rqt_tools. Todas estas herramientas constituyen la base del ROS actual.

Posteriormente, en el 2008, Keenan Wyrobek y Eric Berger, buscaron financiación en su proyecto, abriéndolo al mercado para expandirlo y que ROS llegue a más gente. En este periodo el fundador de Willow Garage, Scott Hassan, se interesa por el proyecto e

invierte en él, comenzando la investigación de un nuevo *Personal Robot* (PR2) y naciendo también el sistema robótico operativo ROS como tal. Durante este periodo ROS crece mucho y saca numerosas versiones muy importantes para su desarrollo.

En 2013, Willow Garage deja de existir dejando el legado de ROS a la emergente compañía Open Source Robotics Foundation que lleva sacando diversas distribuciones de ROS bajo ese nombre y bajo el nombre de Open Robotics desde entonces, sacando la última distribución de ROS1 el año pasado con compatibilidad para Windows [15][21].

Desde 2015 se lleva desarrollando a su vez la segunda versión del sistema robótico, que lo que pretende es solucionar las incompatibilidades que tiene el sistema con productos comerciales y que va a acabar desembocando en una mayor visibilidad, así como en un crecimiento enorme de la plataforma en el plano empresarial, en el que se está haciendo cada vez más fuerte [13]. Además, como ya hemos expuesto en el Estado del Arte, muchas empresas ya están empezando a utilizar el sistema robótico y algunas, como es el caso de Amazon [8], están empezando a sacar sus propios complementos con el objetivo de potenciar otras funciones útiles en la robótica, como la virtualización de sistemas robóticos utilizando la nube. También podemos ver cómo otras empresas importantes se están interesando por ROS, como Google o Microsoft [22].

Figura 3.1.- Logo de ROS.

3.2.- CONCEPTOS BÁSICOS DE ROS

3.2.1.- PAQUETES

Los paquetes de ROS son la unidad más simple de un sistema robótico, sería el equivalente a una célula. Esta célula estará formada en su interior por una organización entre la que se encuentran pequeños módulos de código que en ROS son llamados nodos (los cuales explicaremos a lo largo de este capítulo), librerías con diferentes funciones escritas en Python o C++, archivos de configuración JSON, YAML o XML, etc.

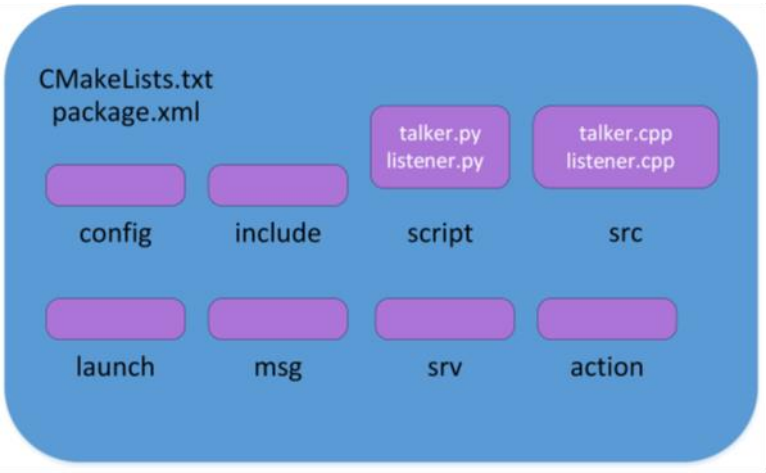


Figura 3.2.- Composición de un paquete completo.

Normalmente, la estructura se mantiene muy parecida a la que se puede ver en la Figura 3.2 y se describe su sistema de ficheros de la siguiente manera [23]:

- Config: esta carpeta contendrá en su interior los archivos de configuración que se escribirán para definir las diferentes características de lanzamiento que tendrán determinados programas, así como diversas preconfiguraciones del sistema. Muchos de estos archivos están escritos en formato de serialización de datos YAML o JSON.

- Include: carpeta destinada a contener las librerías que vamos a utilizar en los nodos para mantenerla separada del resto de scripts. No es estrictamente

necesaria pero su uso se corresponde con una buena práctica de programación.

- Scripts: en esta carpeta tendremos nuestros programas ejecutables (nodos en ROS) escritos en Python.

- Src: tiene la misma función que la carpeta Scripts pero los nodos están escritos en lenguaje C++. En este trabajo no se ha utilizado C++ en ningún caso, por lo que en nuestros paquetes esta carpeta siempre estará vacía o no existirá.

- Launch: en esta carpeta existirán archivos tipo launch . Este tipo de archivos está escrito en XML y permitirá lanzar varios nodos y otros launch files a la vez. Junto con su lanzamiento se lanzará también el ROS MASTER, que iniciará el uso del sistema operativo coordinando su funcionamiento. También permite lanzar nodos con una determinada preconfiguración, la cual puede venir definida en un archivo de la carpeta config. Con la misma funcionalidad se podrá asociar el lanzamiento del nodo a un argumento de forma que ejecute el programa conforme a las particularidades asociadas con su ejecución.

- Msg: contendrá la definición de la estructura de los mensajes que hemos creado para usos determinados dentro de nuestro programa y que no son mensajes estandarizados de ROS, sino creados para una determinada aplicación. En el sistema robótico implementado, se pueden encontrar ejemplos de mensajes diseñados como VectorPos, un mensaje creado para enviar las coordenadas de los objetos encontrados. Este mensaje estará dentro del paquete destinado a la visión.

- Srv: carpeta destinada a la definición de servicios. En nuestro caso esta carpeta no existe debido a que no se han utilizado servicios.

- Action: carpeta que contendrá la definición de las acciones creadas que utiliza el paquete. Pese a no haber diseñado ninguna acción para el desarrollo de este trabajo, se han utilizado acciones de tipo estándar de ROS, lo que hace que esta carpeta no exista en ninguno de los paquetes del trabajo. No obstante, se explicará el funcionamiento de las acciones desempeñadas en el apartado de funcionamiento del sistema, puesto que son esenciales a la hora de entender el funcionamiento del sistema global.

- Package.xml: es el manifiesto, contendrá la forma en la que funciona cada componente dentro del global del paquete.

- CMakeLists.txt: indicará las características de la compilación que se debe llevar a cabo para cada paquete.

Pese a que esta es la forma habitual de estructurar los paquetes, solo se trata de un ejemplo, puesto que en el caso de que alguna de estas carpetas esté vacía se convierte en prescindible. Además, algunos paquetes suelen tener estructuras diferentes por sus características específicas. Es el caso, por ejemplo, de un paquete que contenga la descripción de un robot, en el que se guardan los archivos .dae. Este tipo de archivos será utilizado en la construcción visual del URDF y del SRDF de la simulación, y se guardarán en una carpeta específica *meshes*. La descripción URDF se guardará a su vez en una carpeta llamada urdf, junto con las librerías de macros XML asociadas a dicha descripción.

3.2.2.- NODOS

Los nodos son conjuntos de código ejecutable aislado que, cuando se lanzan, realizan una determinada labor, comunicándose con el resto de nodos mediante el uso de las comunicaciones implementadas en el sistema operativo. ROS Master pondrá en contacto a los nodos realizando dos labores: guardando los roles de cada nodo como un registro en cada tipo de comunicación y haciendo visibles a los nodos partícipes de dicha comunicación entre ellos. Los nodos pueden comunicarse con los otros nodos publicando y subscribiéndose a tópicos, correspondiente con la forma de comunicación más simple (*Publisher-Subscriber*). También lo pueden hacer realizando peticiones de servicios y acciones a otros nodos.

Como resumen se podría decir que cada uno de los nodos partícipes de una comunicación en ROS comunica su rol dentro de la misma al ROS Master. El ROS Master actúa como administrador: hace visibles los nodos al resto de los nodos partícipes en la comunicación y facilita así la transmisión de información entre ellos. El ejemplo más simple, que resume la información que encontramos en la Figura 3.3, es que el nodo *Publisher* y el nodo *Subscriber* informan al ROS Master que van a actuar como tal, es decir, que el nodo de la izquierda va a publicar en el tópico y que el de la derecha va a actuar subscribiéndose a él. ROS Master hará visibles ambos nodos en dicha comunicación, pudiendo ponerse ambos en contacto.

En los subapartados siguientes se especificará más a fondo la labor de cada uno de los elementos de la imagen, así como algunas de sus particularidades.

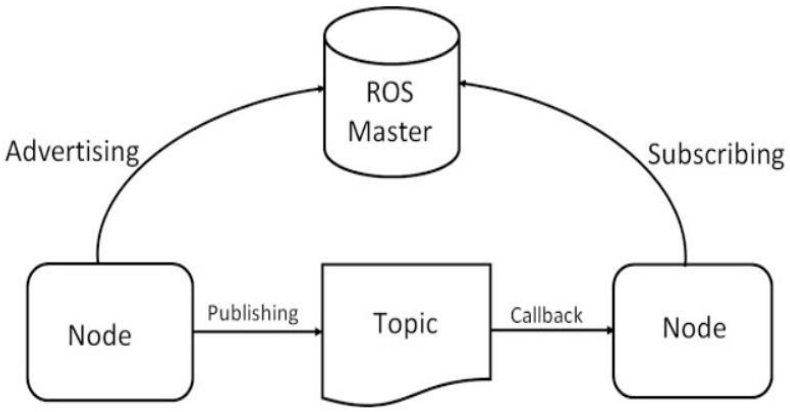


Figura 3.3.- Comunicación básica de ROS de tipo Publicador-Subscriptor

3.2.3.- MENSAJES

Los mensajes son la forma en la que se estructura el intercambio de información entre nodos en una comunicación *Publisher-Subscriber*. Son el equivalente a un tipo de dato en cualquier otro lenguaje de programación o a un objeto si hablamos de programación orientada a objetos. En el caso de ROS, los mensajes están destinados a la transmisión de información. Este mensaje puede estar creado o puede tratarse de un mensaje predefinido de

ROS de la librería *std_msgs*. Los mensajes serán trasladados por medio de tópicos de nodo a nodo.

Un mensaje puede estar creado por diferentes tipos de mensajes predefinidos de ROS, que serán como un tipo de dato en cualquier otro lenguaje de programación. Esto permitirá crear mensajes que puedan tener diferentes utilidades y que se adapten a la funcionalidad del sistema robótico creado; que sean capaces de estructurar adecuadamente la información que se quiere transmitir dentro de su tópico correspondiente. En el paquete desarrollado, se puede encontrar como ejemplo el mensaje Vectorpos, creado para el apartado de visión, en el que se mandará la información sobre los objetos encontrados. La definición de este mensaje se puede observar en la Figura 3.4. y estará dividido en cinco tipos de datos: tres vectores que contendrán las coordenadas X, Y y Z de los circulos localizados y dos variables tipo entero n_rojos y n_verdes, que definirán la cantidad de círculos encontrados de color rojo y verde respectivamente. Las coordenadas de los círculos rojos se guardarán de forma que ocupen las primeras n_rojos posiciones de los tres vectores, siendo la posición 0 de los 3 vectores utilizados el primer círculo rojo encontrado. Posteriormente a estas coordenadas, se guardarán las n_verdes posiciones relativas a los círculos verdes.

```
float32[] x
float32[] y
float32[] z
int32  n_rojos
int32  n_verdes
```

Figura 3.4.- Mensaje de tipo Vectorpos.

3.2.4.- TÓPICOS

Los tópicos son la vía de comunicación principal sobre la que se sostiene la comunicación básica entre nodos. Es una vía de comunicación asociada a un nombre, en la que se difunde y se accede a una determinada información útil para la construcción de nuestro sistema robótico. La información que se transfiere en los tópicos está contenida en mensajes, por lo que estará estructurada en la forma en la que está a su vez definido el tipo de mensaje que se utiliza en ese determinado tópico.

En una comunicación básica, un nodo se encarga de publicar un cierto tipo de información en un tópico (actuará como *Publisher* o Publicador) estructurada en forma de mensaje. El nodo que necesite esa información, para realizar su tarea, podrá acceder a ella subscribiéndose (actuando como *Subscriber* o Suscriptor). De esta forma, accederá a la información del mensaje y llamará a una función *callback* que realizará la labor asociada a la ejecución del nodo.

Al ser un sistema en el que participan muchos nodos, varios nodos pueden compartir la subscripción, es decir, muchos Subscriptores pueden recibir la información proporcionada por un mismo tópico, cada uno con el objetivo de realizar su respectiva labor con dicha información [23].

3.2.5.- SERVICIOS

Los servicios son otra forma de comunicación implementada en la librería básica de comunicación ROS_comm [21] que, como ya hemos visto, nace con la aparición del primer Personal Robot. Por esta razón, al igual que la comunicación tipo *Publisher-Subscriber*, los servicios son una de las formas de comunicación más antiguas del sistema operativo.

Esta forma de comunicación se basa en un protocolo *Server-Client* en el que el servidor ofrece una labor a realizar y el cliente la solicita, esperando una respuesta. Esta implementación es muy útil cuando queremos implementar ciertas labores de tipo solicitud-respuesta, es decir, labores que son necesarias en un determinado momento, únicamente cuando se soliciten. Se distingue de la comunicación Publicador-Subscriptor en que esta se realiza constantemente cada cierto tiempo, mientras que el servicio solo se ejecuta cuando es solicitado por un cliente.

Una vez que el servicio es requerido, este no se podrá parar hasta que termine su labor y devuelva la respuesta, lo cual lo hace ciertamente rígido en algunas implementaciones. La estructura de un servicio en ROS se organiza como la de un mensaje, dividiéndose en dos partes. Por un lado, encontraremos el formato de la petición (*request*) y por otro el de la respuesta (*reply*). Este formato es muy parecido al del mensaje del que ya

hemos hablado y ambos, requerimiento y respuesta, pueden estar conformados por varios *data types*, mensajes predefinidos de ROS.

```
#request constants
int8 FOO=1
int8 BAR=2
#request fields
int8 foobar
another_pkg/AnotherMessage msg
---
#response constants
uint32 SECRET=123456
#response fields
another_pkg/YetAnotherMessage val
CustomMessageDefinedInThisPackage value
uint32 an_integer
```

Figura 3.5.- Estructura de un servicio a modo de ejemplo.

3.2.6.- ACCIONES

Las acciones son una funcionalidad implementada por el paquete de ROS actionlib, que forma parte del metapaquete ros_base. Este paquete, aunque no forma parte del metapaquete original de ros_core, extiende sus funcionalidades y herramientas. Entre esas funcionalidades se puede encontrar la utilización de nodelets, plugins o aplicaciones complementarias, no necesariamente robóticas [24][25].

Las acciones (*actions*) son un derivado de los servicios muy utilizado en el desarrollo de sistemas robóticos en ROS. Una de las particularidades de los servicios es que una vez que se inician, no se pueden parar mientras el sistema operativo siga funcionando. Las acciones sustituyen a los servicios en aquellas aplicaciones en las que la imposibilidad de parar un servicio, a lo largo de su ejecución, suponga un problema. Las tareas de larga duración, en algunos casos, no son aptas para la implementación mediante el uso de servicios ya que no hay manera de frenar el proceso mientras este siga funcionando y no complete su labor. En el caso de que se produzcan errores durante la ejecución, esto puede originar daños en el hardware empleado.

Para solucionar ese problema se utilizan las acciones, basadas en una solicitud-respuesta, pero con una estructura un poco diferente a la de los servicios, esta vez formada por 3 mensajes diferentes.

```
# Define the goal
uint32 dishwasher_id  # Specify which dishwasher we want to use
---
# Define the result
uint32 total_dishes_cleaned
---
# Define a feedback message
float32 percent_complete
```

Figura 3.6.- Estructura de una acción a modo de ejemplo.

En la Figura 3.6. podemos ver el ejemplo de una acción. La primera parte de la acción contendrá un mensaje con el resultado que deseas obtener, el objetivo o *goal*. En la segunda parte de la acción se define un mensaje que se enviará una vez, cuando la acción se haya terminado. En la tercera y última parte, se define el *feedback*, el mensaje que irá informando del estado de la acción mientras esta siga ejecutándose. Este último mensaje será especialmente interesante para el seguimiento de la labor ejecutada y la comprobación de su funcionamiento.

3.2.7.- URDF

ROS dispone de un metapaquete creado para la representación de modelos robóticos llamado urdf. Su nombre proviene del formato unificado de descripción de robots *Unified Robotic Description Format* (URDF). Este es el formato utilizado para la descripción robótica dentro del sistema operativo.

Este metapaquete está diseñado para la creación, visualización y simulación de modelos robóticos, pudiéndose apoyar también en modelos CAD en formato collada. El metapaquete se encarga de convertir la estructura descrita en el formato URDF, escrito en el lenguaje de marcado XML, para adaptar toda esa información a la representación y futura simulación o interacción con Gazebo [26]. Entre la información que se podrá encontrar en una descripción URDF podremos encontrar [27]:

- Relación y disposición de los eslabones y articulaciones.

- Direccionamiento hacia la carpeta donde se encuentran las representaciones de los eslabones en formato collada.

- Matriz de colisión de los eslabones.

- Tipo de grado de libertad o movimiento de las articulaciones.

- Plugin de gazebo para integrarlo en una simulación.

Para entender la disposición de un URDF procedemos a describir a continuación los tags básicos con los que luego se conformará la estructura del robot particular de este proyecto:

- ***Link***: el *tag link* describe un eslabón. El modelado de dicha extremidad incluirá el tamaño, el color y puede incluir también la descripción .dae (*collada*) del mismo para una representación mejor y más visual de todas las partes del robot. En las Figuras 3.7 y 3.8 se puede ver, por un lado, la sintaxis con la que se describirán las extremidades en el formato URDF y, por otro, una representación visual de cómo lo entenderá ROS, diferenciando entre la parte visual del *link*, la parte inercial y las colisiones o límites físicos relacionados con la forma del eslabón. Cuando estos límites físicos impacten entre sí o con otros objetos presentes en el entorno, ROS lo reconocerá como una colisión y dará lugar a un error.

```
<link name="<name of the link>">
<inertial>...........</inertial>
  <visual> ...........</visual>
  <collision>.........</collision>
</link>
```

Figura 3.7.- Sintaxis de la definición de un link en URDF.

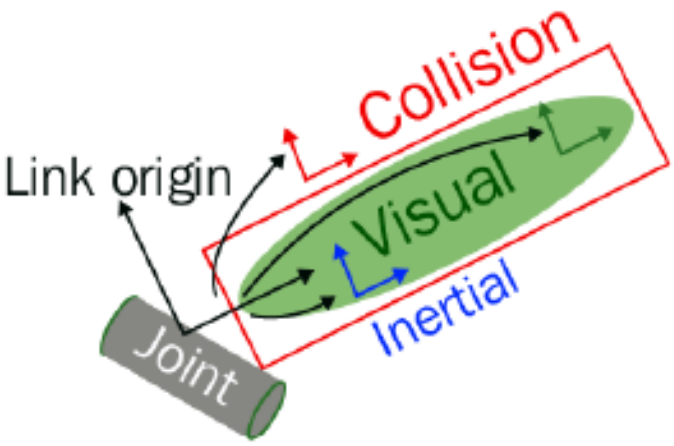


Figura 3.8.- Representación de los argumentos de la sintaxis de un *link*.

- Joint: *tag* que representa las articulaciones del robot. En este *tag* se definirá el tipo de movimiento de la articulación (prismática, continua, fija, …etc.) así como su dinámica y velocidad. Como se sabe, una articulación está comprendida por definición entre dos eslabones, un eslabón origen y un eslabón destino. Estos dos eslabones también se definirán dentro del propio tag.

```
<joint name="<name of the joint>">
  <parent link="link1"/>
  <child link="link2"/>
  <calibration .... />

    <dynamics damping ..../>
    <limit effort .... />
 </joint>
```

Figura 3.9.- Descripción de articulación en URDF.

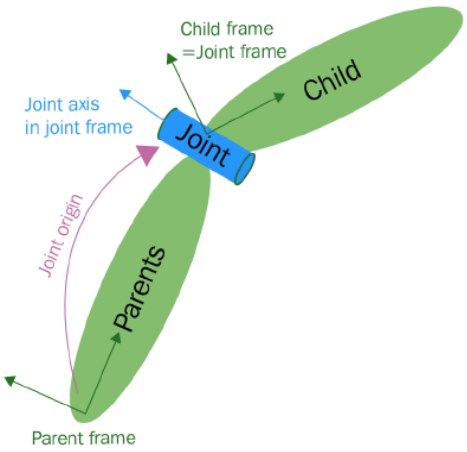


Figura 3.10.- Representación de los argumentos de la sintaxis de una articulación.

- Transmission: este *tag* se encarga de relacionar las articulaciones con sus respectivos actuadores. Define el tipo de transmisión, basado en el tipo de motor que tendrá el robot y sus parámetros. Servirán como argumento para construir la interfaz *hardware* del sistema robótico, para poder así interactuar en ROS con los controladores mediante el uso de tópicos. Se definirá el nombre de la articulación, el nombre de la transmisión, el tipo de transmisión y la interfaz hardware del controlador de ROS asociado a la transmisión.

```
<transmission name="<name of the transmission>">
  <type>.....</type>
    <joint name="<name of the joint of the transmission>">
      <hardwareInterface>.....</hardwareInterface>
    </joint>
  <actuator name="<name of the actuator>">
    <hardwareInterface>.....</hardwareInterface>
    <mechanicalReduction>.....</mechanicalReduction>
  </actuator>
</transmission>
```

Figuras 3.11.- Tag transmission en el modelo URDF utilizado en este trabajo.

- Robot: define el conjunto de articulaciones y eslabones que constituyen, en su conjunto, el robot que se quiere definir. El modelo de robot que se creará en RVIZ no será más que el conjunto de eslabones y articulaciones asociado a un mismo robot. Con dicho *tag* podremos interactuar en muchos casos,

desde diferentes herramientas y programas del sistema operativo, bajo su nombre asociado.

- Gazebo: este tag permite el cómputo de ciertos parámetros en la simulación de Gazebo. Además, se pueden añadir *plugins* de Gazebo para determinadas utilidades de la plataforma de simulación, lo cual permite, por ejemplo, la simulación de los controladores asociados con cada transmisión especificada en el archivo. Esto le permitirá a Gazebo constituir fácilmente la interfaz *hardware* del modelo robot.

3.3.- HERRAMIENTAS Y PAQUETES UTILIZADOS

3.3.1.- RVIZ

Rviz es una herramienta de visualización de información incluida dentro de la instalación básica de ROS. Esta herramienta ofrece una visualización 3D para diferentes aplicaciones. Entre estas aplicaciones se pueden encontrar la visualización de modelos de robot en URDF, la captura de la información recibida por sensores y reproducción de los datos capturados. La representación de la información recibida por los sensores y cámaras 3D se realiza mediante el uso de nubes de puntos en la ventana de visualización de RVIZ. También se podrá añadir la visualización de los ejes de referencia de cada articulación e incluso variar la posición de las articulaciones lanzando el GUI del joint_state_publisher. Esto ayudará a la construcción del modelo robot, así como permitirá la observación de su comportamiento dentro de la herramienta de visualización.

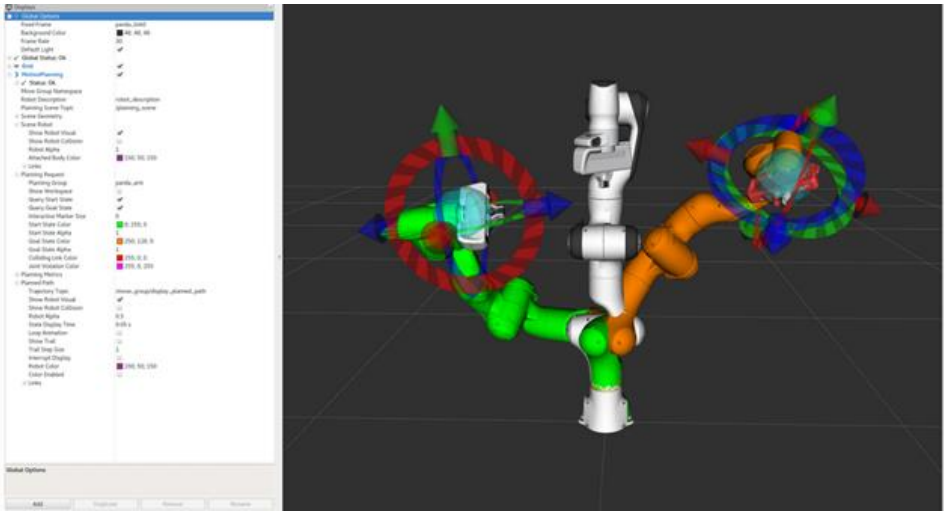


Figura 3.12.- Herramienta de visualización RVIZ.

3.3.2.- MOVEIT!

Moveit! es una plataforma compuesta por un gran número de librerías en ROS cuyo cometido es la planificación de trayectorias, así como el control de hardware simulado y real. Entre las funcionalidades de *Moveit!* encontramos las siguientes [28]:

- Planificación: genera trayectorias para modelos robots de diferentes grados de libertad y para diversos entornos de trabajo.

- Manipulación: contiene herramientas para el control específico de tareas de manipulación, contando con los elementos externos del entorno y pudiendo generar el agarre de objetos.

- Cinemática: integra la cinemática directa del robot con la utilización del URDF y admite la integración de plugins de cinemática inversa para diferentes modelos de robot.

- Control: capacidad de ejecutar trayectorias parametrizadas temporalmente para interfaces reales o simuladas.

- Percepción 3D: integración dentro del sistema robótico de cámaras de profundidad y nubes de puntos que nos servirán para representar y visualizar nuestro entorno con Octomaps.

- Colisiones: consigue generar trayectorias que logran esquivar los obstáculos presentes en el entorno de trabajo, para poder desarrollar su acción sin percances. Los obstáculos se pueden determinar incluyéndolos dentro de la definición del propio robot, si es un entorno de trabajo fijo. También se podrán detectar mediante la utilización de sensores de distancia o cámaras 3D.

Para llevar a cabo todas sus funcionalidades, *Moveit!* cuenta con una arquitectura conformada alrededor del nodo denominado move_group. Este nodo integra la información del robot, la información del entorno y permite al usuario interactuar directamente con el robot. Esto lo hace llamando desde el move_group, mediante la utilización de *plugins*, a acciones y servicios que lleven a cabo las peticiones del usuario.

El move_group utiliza la información del URDF, del SRDF y de distintos tipos de archivos de configuración. Esta información la obtiene de la interfaz *ROS Param server* y le permite integrar el modelo robot y obtener información de sus posiciones articulares en *Moveit!*. En consecuencia, el move_group podrá interactuar con la interfaz *hardware* que permitirá el control del robot real o su simulación a través de sus controladores. Toda la información necesaria para esta interacción se genera con el paquete *Moveit!* de nuestro robot. Una vez configurado adecuadamente, se podrá interactuar con *Moveit!* creando nodos que incluirán la librería MoveitCommander de Python o C++. También admite una interacción desde la misma GUI del RViz dedicada al MotionPlanning, como se puede observar en la Figura 3.12.

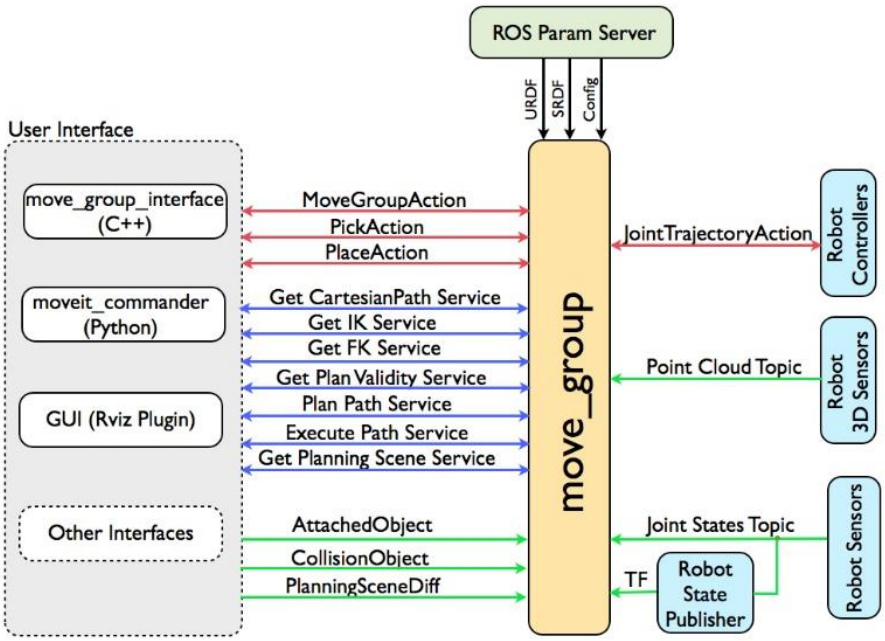


Figura 3.13.- Esquema explicativo de las herramientas de alto nivel del move_group [28].

En la Figura 3.13 se puede ver cómo el move_group funciona como un nodo integrador de las funcionalidades de Moveit!, conectándolas e incorporándolas como plugins o complementos, que pueden estar desarrollados o mejorados por el programador. Esta integración ayuda a conformar un sistema robótico compacto: que contenga acceso al cálculo de la cinemática directa e inversa, que utilice distintos planificadores de trayectoria y que incorpore múltiples funcionalidades nuevas que faciliten el trabajo robótico.

La herramienta utilizada para generar el paquete Moveit! del robot (para la configuración del move_group) es el Moveit! Setup Assistant. Esta herramienta se utiliza para generar el SRDF, todos los archivos de configuración, archivos tipo *launch* y algunos *scripts*. Todos ellos se generarán partiendo de una descripción URDF del robot.

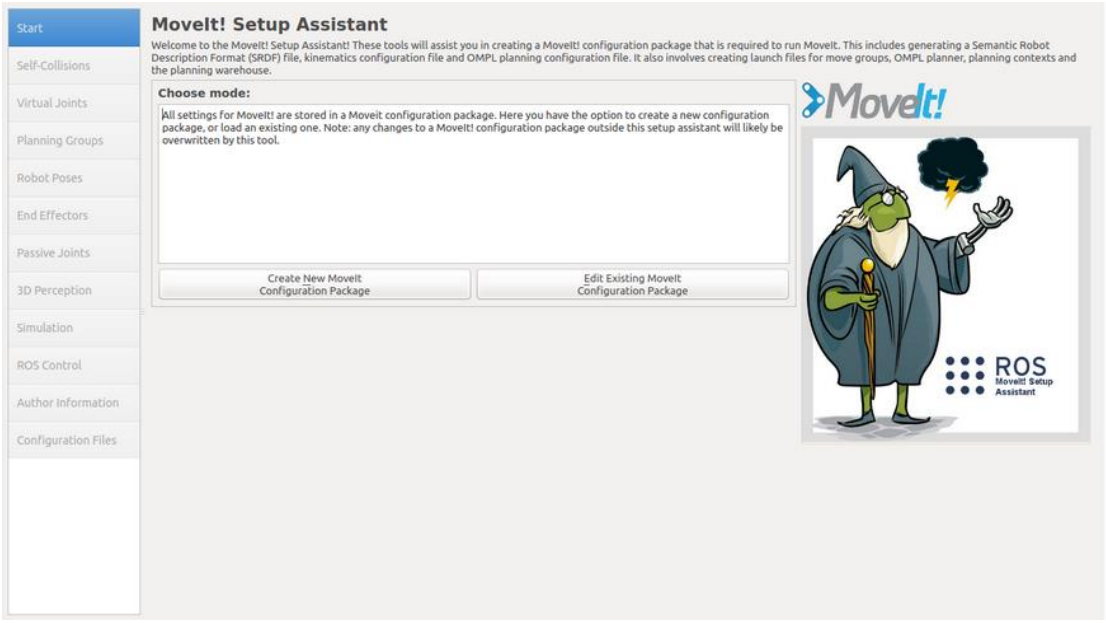


Figura 3.14.- *Moveit! Setup assistant*.

En la Figura 3.14 se puede ver la herramienta *Moveit! Setup Assistant*, así como todos los pasos que se deberán seguir para obtener el paquete y junto con los archivos que permitirán trabajar con el modelo robot. A lo largo del proceso de creación del paquete, se localizarán aquellas colisiones del URDF que deben ser desactivadas por ser links adyacentes. También se añadirán grupos de planificación, los cuales nos servirán a la hora de controlar partes específicas del robot. Estos grupos de planificación se suelen dividir comúnmente en dos partes: el grupo encargado del posicionamiento y orientación del robot; y el grupo correspondiente a la garra o efector final. *Moveit! Setup A*ssistant nos permite además la configuración de determinadas poses predefinidas, que pueden ayudar a la hora de desarrollar distintas aplicaciones para facilitar su llamada desde las APIs de *Moveit!*. Se puede añadir también un efector final, escogiendo el grupo correspondiente con nuestra garra. Por último, se pueden seleccionar aquellas articulaciones que se consideren pasivas en nuestro modelo, es decir, que no contengan ningún tipo de actuador.

3.3.3.- ROS GUI

ROS ofrece herramientas para visualizar gráficos, representaciones o interactuar con el sistema, escribiendo en tópicos y ayudando a depurar el sistema robótico. En la Figura 3.15 se puede observar la herramienta de ROS GUI llamada rqt_graph, que permite la visualización de las comunicaciones entre nodos que estén funcionando dentro del sistema robótico.

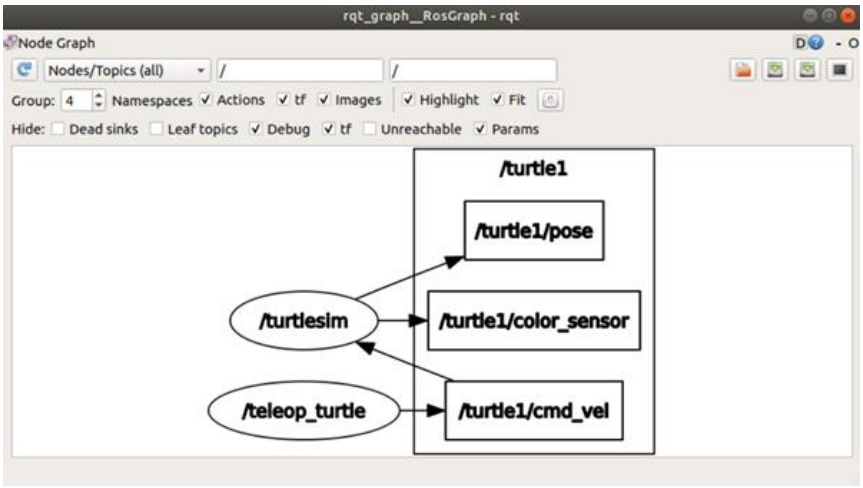


Figura 3.15.- Diagrama ejemplo rqt_graph.

3.3.4.- GAZEBO Y ROS_CONTROL

Gazebo es una herramienta de simulación independiente de ROS, capaz de trabajar con la dinámica, inercia y representación del robot. Es apta para simular los modelos de robot en condiciones reales, antes de probarlos con el hardware real, lo cual es el complemento perfecto para testear algoritmos o condiciones reales de funcionamiento, antes de ser implementados. Gazebo da la posibilidad de tener hardware robótico sin falta de tener que desembolsar dinero en un hardware real, por lo que fue, sin duda, una pieza importante en el desarrollo robótico desde que entró a formar parte de ROS.

Además, con la utilización de los paquetes ros_control, el standard de ROS para la creación de interfaces software para trabajar con controladores, se tiene la opción de trabajar con la simulación de los motores del robot en Gazebo. Dentro del sistema operativo, se pueden integrar los controladores desde el propio URDF mediante el uso de transmisiones.

En la Figura 3.16 se puede ver el entorno de simulación de Gazebo vacío con el modelo del robot Sainsmart.

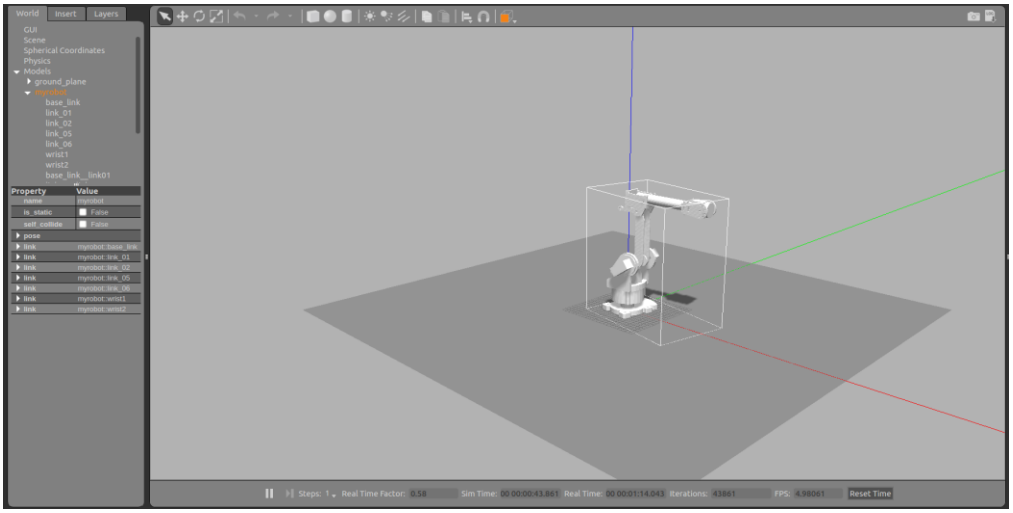


Figura 3.16.- Entorno de simulación de Gazebo vacío con el modelo Sainsmart.

3.3.5.- ROSSERIAL

Rosserial es un paquete de ROS dedicado a la integración de Arduino (así como otros dispositivos de puerto serie) dentro del sistema robótico. Este paquete actúa como un nodo en el que, en su interior, encontramos el dispositivo conectado por el puerto serie con su respectivo programa en él. De esta forma, el dispositivo conectado a dicho puerto serie actúa como una especie de pseudonodo recubierto por la capa del nodo rosserial, pudiendo interactuar como un nodo más de nuestro sistema junto con el resto de nodos [29]. En la Figura 3.17 se muestra un esquema de cómo se comunica el nodo de rosserial con ROS, actuando el Arduino como Subscriber y utilizando el tópico /command como medio de comunicación entre ambos.

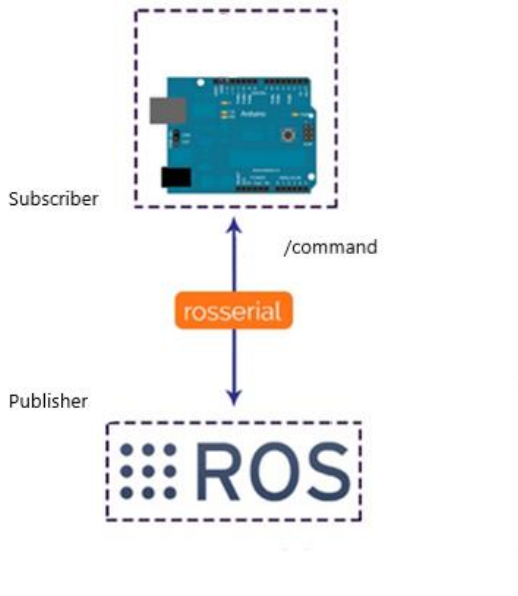


Figura 3.17.- Funcionamiento de rosserial en ROS.

3.3.6.- RASPICAM_NODE

Al igual que en el caso de rosserial, raspicam_node es un nodo que permite integrar de forma simple una Raspicam en cualquier sistema robótico de ROS, pudiendo publicar las imágenes del entorno del robot en un tópico. Los nodos del sistema robótico podrán acceder así a dichas imágenes y tratarlas como un mensaje de tipo Image.

Raspicam_node, además de funcionar como un nodo, tiene diversas funcionalidades que también pueden ser útiles para el procesamiento de imágenes. Por un lado, permitirá procesar el mensaje de salida (la imagen) con la herramienta rqt_configure, con la que se podrán ajustar las propiedades de la imagen (el brillo, el contraste, …etc.) para adecuarlas a las condiciones que se consideren idóneas. También ofrecerá diversas opciones para la visualización de la imagen, incluyendo los *frames per second (fps)*, que se podrán regular para obtener la visualización más fluida o mejor adaptada para el caso particular de aplicación. Además, se podrán escoger el número de píxeles del formato con el que trabajaremos en la imagen. Por último, raspicam_node ofrece una opción de calibración automatizada para la detección de patrones y la búsqueda de las distorsiones de la imagen obtenida. Dicha información será guardada y utilizada automáticamente por el raspicam_node para eliminar cualquier tipo de distorsión. Cada vez que se inicie la cámara, cargará dicha información del archivo de configuración *.yaml*, guardado en la carpeta camera_info del paquete [30].

4. Planta de trabajo

La planta de trabajo para la que se desarrolla la labor va a contar con la presencia de un brazo de tipo paletizador educacional de la marca Sainsmart. Este brazo tiene seis grados de libertad y es la pieza principal sobre la que se sostiene todo el desarrollo software llevado a cabo durante el proyecto. Este robot lo podemos ver en la Figura 4.1. Pero también se necesitará el uso de diversos dispositivos de control que se encarguen de correr toda la arquitectura software que se ha creado en ROS y, por tanto, de llevar a cabo el control del robot encargándose también del apartado hardware del brazo robótico. En la planta se dispone de una Raspberry, un Arduino Mega y una placa driver de motores PCA9685 de la marca Adafruit, con la que accionaremos los motores del robot para situarlo en la posición final. Durante los siguientes subapartados se expondrá con más especificidad el funcionamiento de cada una de las partes mencionadas.

Figura 4.1.- Foto del robot Sainsmart 6DoF Desktop [31].

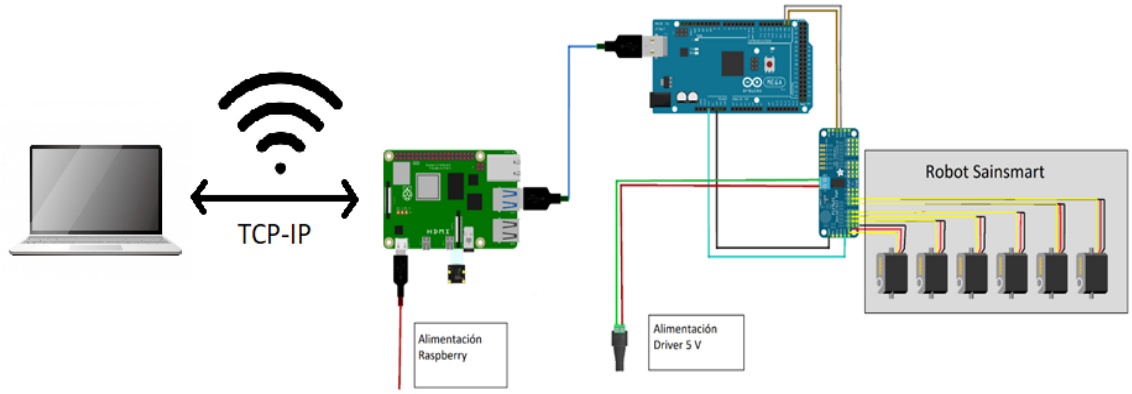


Figura 4.2.- Esquema global de las conexiones y los componentes de la planta de demostración.

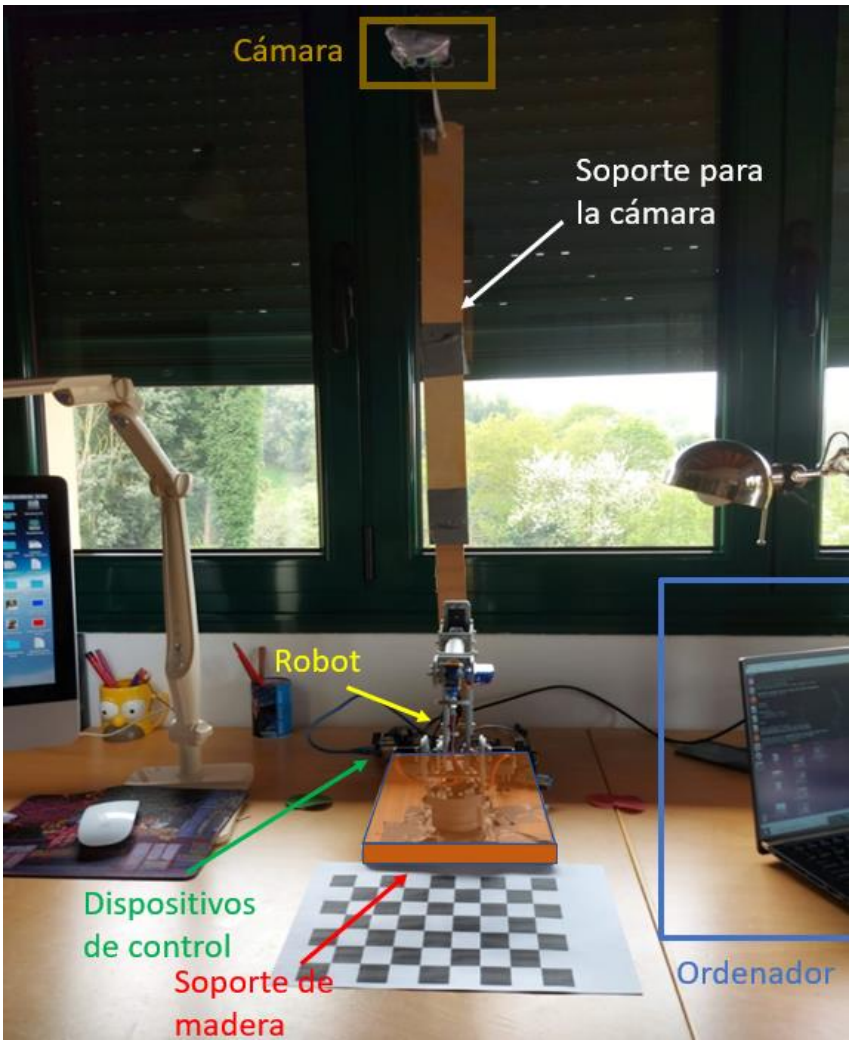


Figura 4.3.- Fotografía de la planta de trabajo en la que se ha llevado a cabo la demostración.

4.1.- BRAZO ROBÓTICO

El brazo robótico del que se dispone en la planta es el robot 6-Axis Desktop Robotic Arm 6 DoF (Degrees of Freedom) de la marca Sainsmart. Este robot de tipo educativo está construido en material PVC y está formado por 6 servomotores [31]:

- 4 servos MG996 para las cuatro primeras articulaciones (base, hombro, codo y antebrazo).

- 2 servos SG90 para los dos últimos giros correspondientes a los dos giros asociados a la muñeca.

- En este proyecto no se tiene a disposición un efector final o garra por lo que solo se hablará sobre la posición del brazo.

Una particularidad de este robot, que tiene cierta importancia mencionar, es que el brazo contiene una cinemática cerrada típica de los robots paletizadores en forma de mecanismo paralelogramo. Este tipo de configuraciones buscan la horizontalidad del movimiento, manteniendo la orientación paralela al suelo mientras el motor dos, que encontramos en la base, no se mueva. Esta utilidad es especialmente interesante para el desplazamiento de palets y hay que contar con ella para el cálculo.

4.1.1.- PARÁMETROS DENAVIT-HARTENBERG DEL ROBOT

Los parámetros DH sirven para describir la posición de las articulaciones de una forma matricial y simple siguiendo un convenio de descripción, el cual va a ser clave para el cálculo de la cinemática directa. Estos parámetros se han calculado siguiendo la convención modificada del algoritmo. Aunque el robot de este trabajo contenga un cuadrilátero cerrado, no lo consideraremos para la descripción del robot con los parámetros DH. Esto se debe a que se hallarán los ángulos conforme a una descripción sin este tipo de mecanismo y luego se detallará la relación de dichos ángulos para el cálculo de los ángulos en el robot Sainsmart. Esta convención constará, por lo tanto, de los siguientes criterios [32][33][34]:

1. Las transformaciones correspondientes a dichos parámetros siempre se realizarán con respecto al eje precedente y posterior. Se deben situar los ejes de forma que sigan el convenio que veremos en los pasos siguientes.

2. El eje Z deberá siempre estar contenido en la dirección del eje de rotación de la articulación pertinente.

3. El eje X, deberá situarse perpendicular al eje Z e intentando buscar, siempre que sea posible, la dirección del eslabón que continúa a la articulación.

4. Asignación del eje Y para que forme un triedro a derechas con los ejes X y Z.

5. Se hallan las distancias y ángulos que definen la tabla DH basándonos en las siguientes reglas, que vienen representadas de forma gráfica en la Figura 4.4:

   - ai = distancia desde Zi hasta Zi+1 medida a lo largo del eje Xi.
   - αi = ángulo desde Zi hasta Zi+1 medida alrededor del eje Xi.
   - Di = distancia desde Xi-1 hasta Xi medida a lo largo del eje Zi.
   - Θi = ángulo desde Xi-1 hasta Xi medida alrededor del eje Zi.

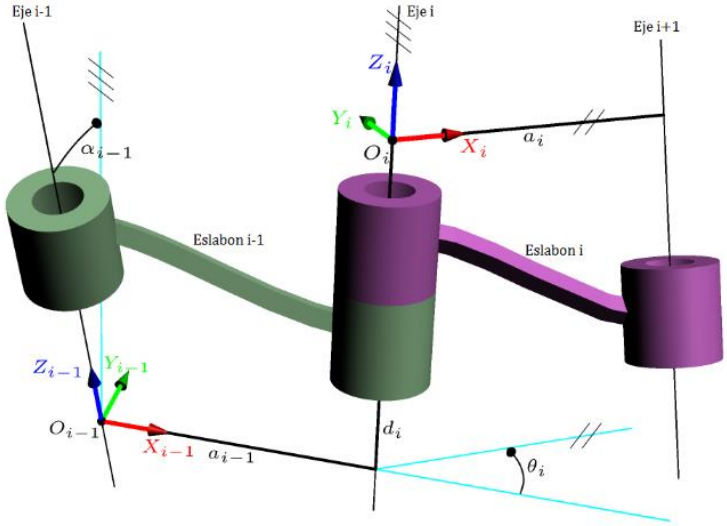


Figura 4.4.- Representación del sistema de coordenadas Denavit-Hartenberg modificado [35].

Los parámetros DH hallados se pueden ver plasmados en la Tabla 4.1 y señalados en la representación esquemática del robot de la Figura 4.5:

i	Alpha(º)	a(mm)	D(mm)	Tetha(º)
1	0	0	0	Tetha1
2	-90	a1	0	Tetha2
3	0	a2	0	Tetha3
4	-90	a3	d4	Tetha4
5	90	0	0	Tetha5
6	-90	0	0	Tetha6

Tabla 4.1.- Parámetros Denavit-Hartenberg del robot.

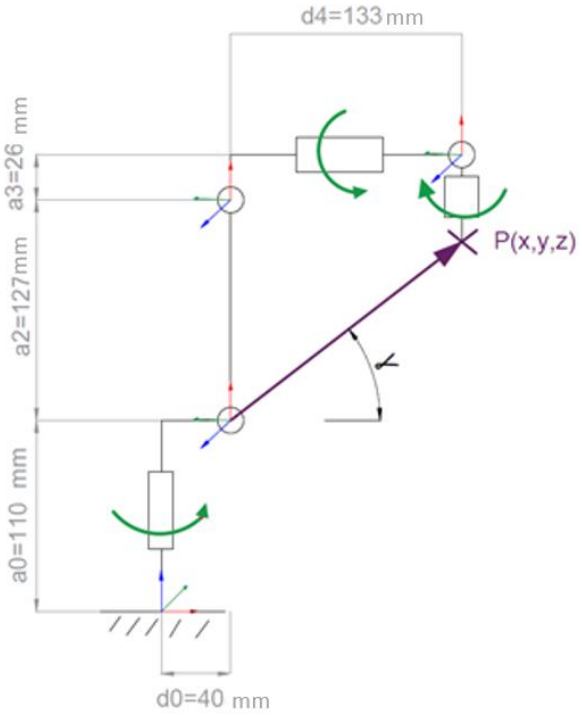


Figura 4.5.- Representación de los parámetros en la simplificación del robot Sainsmart.

4.1.2.- CINEMÁTICA DIRECTA

La cinemática directa del robot es la parte de la cinemática que estudia la posición del efector final, tanto en orientación como en posición, partiendo de los ángulos de rotación de las articulaciones. Esta cinemática puede definirse haciendo uso de las matrices homogéneas. Estas matrices definen la transformación de traslación y rotación entre cada eje de la estructura del brazo con respecto al anterior y, multiplicando todas ellas, se obtiene la matriz que transforma posición y orientación de la base al efector final [28]. Estas matrices de transformación se pueden calcular sustituyendo los parámetros DH en la matriz de la Ecuación 4.1 para cada una i:

$$ {}^{i-1}A_i = \begin{pmatrix} \cos(\theta i) & -\sin(\theta i)\cos(\alpha i) & \sin(\theta i)\sin(\alpha i) & ai\cos(\theta i) \\ \sin(\theta i) & \cos(\theta i)\cos(\alpha i) & -\cos(\theta i)\sin(\alpha i) & ai\sin(\theta i) \\ 0 & \sin(\alpha i) & \cos(\alpha i) & di \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.1) $$

Las matrices homogéneas calculadas se encuentran detalladas en la ecuación 4.2 y fueron calculadas mediante la utilización de la Ecuación 4.1:

$$T_{01} = \begin{pmatrix} \cos(\theta_1) & -\sin(\theta_1) & 0 & 0 \\ \sin(\theta_1) & \cos(\theta_1) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}; \quad (4.2)$$

$$T_{12} = \begin{pmatrix} \cos\left(\theta_2 - \frac{\pi}{2}\right) & -\sin\left(\theta_2 - \frac{\pi}{2}\right) & 0 & 40 \\ 0 & 0 & 1 & 0 \\ -\sin(\theta_2 - \frac{\pi}{2}) & -\cos\left(\theta_2 - \frac{\pi}{2}\right) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

$$T_{23} = \begin{pmatrix} \cos(\theta_3) & -sin(\theta_3) & 0 & 127 \\ \sin(\theta_3) & \cos(\theta_3) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

$$T_{34} = \begin{pmatrix} \cos(\theta_4) & -\sin(\theta_4) & 0 & 26 \\ 0 & 0 & 1 & 133 \\ -\sin(\theta_4) & -\cos(\theta_4) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

$$T_{45} = \begin{pmatrix} \cos(\theta_5) & -\sin(\theta_5) & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \sin(\theta_5) & \cos(\theta_5) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix};$$

$$T_{56} = \begin{pmatrix} \cos(\theta_6) & -\sin(\theta_6) & 0 & 0 \\ 0 & 0 & -1 & 0 \\ \sin(\theta_6) & \cos(\theta_6) & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Se añaden, para que el cálculo de la posición sea definitivo, las matrices de transformación asociadas con la base y con el efector final. No se ha añadido en el cálculo de la cinemática directa ningún efector final, ya que el robot empleado no cuenta con él. Las matrices asociadas al efector final y a la base se dejarán indicadas en las ecuaciones 4.3, 4.4 y 4.5 [32].

$$T_B = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 110 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.3)$$

$$T_{TF} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.4)$$

$$T_{PTF} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & f \\ 0 & 0 & 1 & e \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (4.5)$$

La matriz de transformación total es aquella en la que, sustituyendo los ángulos de nuestras articulaciones, se obtiene la rotación y traslación del efector final multiplicando todas las matrices descritas. Esta matriz de transformación total se calcula con la Ecuación 4.6. En este proyecto se prescinde del uso de las magnitudes correspondientes con el efector final, las cuales se pueden incluir en la transformada de forma fácil una vez que se conozcan.

$$H_T = T_B \times T_{01} \times T_{12} \times T_{23} \times T_{34} \times T_{45} \times T_{56} \times T_{TF} \times T_{PTF} \quad (4.6)$$

4.1.3.- CINEMÁTICA INVERSA

La cinemática inversa es la parte de la cinemática que, partiendo de la posición y orientación deseada para el efector final, calcula la posición angular de cada articulación. En el caso de este trabajo, se va a trabajar con los 6 grados de libertad del robot empleado. Esto quiere decir que el resultado cuenta con una solución de 6 ángulos: con los tres primeros ángulos se logra situar el brazo en la posición deseada para la recogida, mientras que con el resto se logra orientar el efector final para que tenga la orientación adecuada. Para ello, se divide el cálculo en dos partes: el cálculo de la posición y el de la orientación [32][36].

4.1.3.1.- Posiciones extraídas de la matriz homogénea de transformación

Para el cálculo de θ1, θ2 y θ3, los ángulos responsables de la posición, se usan los elementos de posición de la matriz homogénea de transformación, extraídos de los tres primeros elementos de la última columna de la matriz. La matriz homogénea de transformación fue calculada en la ecuación 4.6. Los resultados de las posiciones XYZ se pueden ver en las ecuaciones 4.7, 4.8 y 4.9 respectivamente:

$$P_x = cos(\theta_1) \times (133 \times cos(\theta_2 + \theta_3) + 26 \times sin(\theta_2 + \theta_3) + 127 \times sin(\theta_2) + 40) \quad (4.7)$$

$$P_y = sin(\theta_1) \times (133 \times cos(\theta_2 + \theta_3) + 26 \times sin(\theta_2 + \theta_3) + 127 \times sin(\theta_2) + 40) \quad (4.8)$$

$$P_z = 127 \times \cos(\theta_2) + \sqrt{18365} \times \cos(\theta_2 + \theta_3 + \operatorname{atan}(133/26)) + 110 \quad (4.9)$$

Para el cálculo de estos tres primeros ángulos se realiza un estudio del plano de movimiento de cada articulación, obteniendo las relaciones geométricas que relacionan la posición buscada con el ángulo de las tres articulaciones.

4.1.3.2.- Cálculo de los ángulos de posicionamiento

En el caso del ángulo θ_1, se analiza el plano XOY. Es el único ángulo independiente de los demás, puesto que solo se necesita girar el brazo hasta que todas las extremidades estén en el plano vertical que contiene al eje origen del sistema de coordenadas y al punto correspondiente con la posición objetivo. De esta forma se obtienen las ecuaciones 4.10 y 4.11, cuya combinación culmina en el cálculo de θ_1 utilizando la ecuación 4.12, resultado de la aplicación del teorema de Pitágoras. Esta relación geométrica se puede ver, expresada de forma gráfica, en la Figura 4.6.

$$\cos\theta_1 = Px \tag{4.10}$$

$$\sin\theta_1 = Py \tag{4.11}$$

$$\theta_1 = \operatorname{atan}\left(\frac{Py}{Px}\right) \tag{4.12}$$

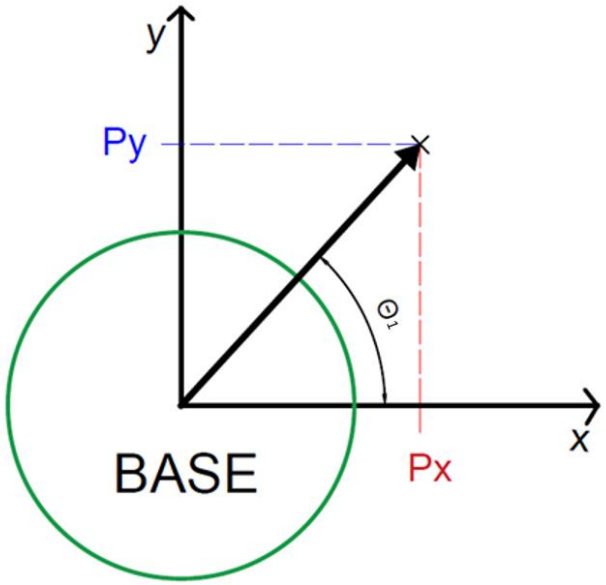


Figura 4.6.- Representación esquemática de la obtención del ángulo θ_1.

Para calcular θ_3 habrá que tener en cuenta dos cuestiones. Por un lado, se halla la relación de ángulos en la suposición de que el brazo parta de estar completamente estirado, lo cual nos servirá posteriormente para deducir nuestro ángulo desde la posición en la que nuestro robot inicia el movimiento. Por otro lado, habrá que tener en cuenta que la longitud respecto a la articulación del hombro no será la longitud d4 (H_{ex} en la Figura 4.7), sino que será una longitud que se denominará como He (Hombro efectivo) con dos componentes (H_{ex} y H_{ey}). Esta simplificación del eslabón se puede observar en la Figura 4.7 y supone que tendremos que obtener el valor de la diferencia Ω, para el cálculo de θ_3.

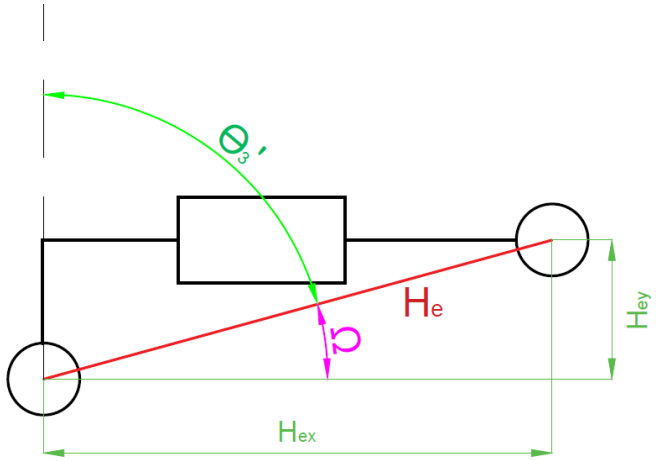


Figura 4.7.- Cálculo del hombro efectivo He.

Posteriormente, se adaptarán los resultados geométricos obtenidos a la configuración inicial del robot, a sus posiciones iniciales. El ángulo θ_3 buscado se puede encontrar representado en la Figura 4.8, que muestra la relación que tiene este último con el ángulo θ_3' de la Figura 4.7.

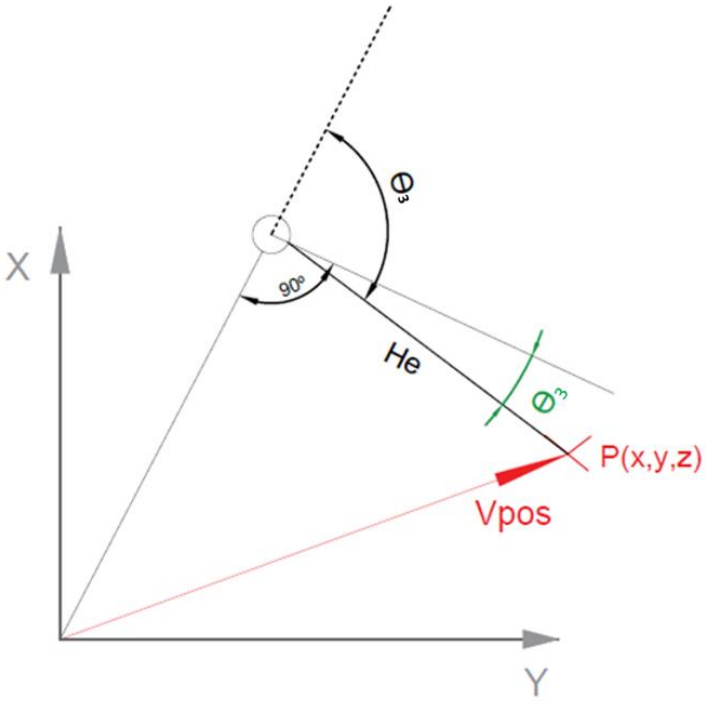


Figura 4.8.- Cálculo de θ_3.

Como se puede observar en la Figura 4.7, el eslabón He se ha simplificado como la hipotenusa de sus dos componentes H_{ex} y H_{ey}, que son los componentes vertical y horizontal del eslabón. Esto provoca que, al simplificarlo como la línea He (que va desde la articulación del codo hacia el final del eslabón) surja la necesidad de girar una diferencia Ω extra. Esta diferencia se corresponderá con el arco tangente de los componentes H_{ex} y H_{ey}. Θ_3' será el ángulo que se debería mover la tercera articulación para posicionarse, sin contar con el efecto geométrico causado por la simplificación de H_e, debido a la forma particular del eslabón. El cálculo final de Θ_3 dependerá de esos dos ángulos, a la hora de que no haya errores que alteren el posicionamiento del brazo. El procedimiento para calcular los ángulos de posicionamiento es puramente geométrico y las fórmulas resultantes son las que podemos encontrar desde la 4.13 hasta la 4.18.

$$Hex = d4 \tag{4.13}$$

$$Hey = a3 \tag{4.14}$$

$$He = \sqrt{Hex^2 + Hey^2} \tag{4.15}$$

En la Ecuación 4.15 se describe la simplificación de la longitud de nuestro eslabón conforme a sus dos componentes Hex y Hey definidos en las Ecuaciones 4.13 y 4.14.

$$m = ||Vpos|| = \sqrt{Px^2 + Py^2 + Pz^2} \tag{4.16}$$

$$\cos(\theta_3') = \frac{-a2^2 - He^2 + m^2}{2 \times a2 \times He} \tag{4.17}$$

$$\theta_3 = \frac{pi}{2} - atan\left(\frac{\sqrt{1 - \cos(\theta_3')^2}}{\cos(\theta_3')}\right) + atan(\frac{a3}{d4}) \tag{4.18}$$

En la Ecuación 4.16 se halla el módulo del vector distancia entre el eje de rotación del eslabón 2 y la posición objetivo. En la Ecuación 4.17, calculado previamente He y m, se utiliza el Teorema del Coseno para extraer θ_3'. Posteriormente, tal y como se puede observar

en la Ecuación 4.18, θ_3 es despejada con respecto a la posición inicial del brazo, utilizando los elementos calculados y apoyándose en la geometría descrita en las Figuras 4.7 y 4.8.

Una vez calculado θ_3 se procederá al cálculo de θ_2. La Figura 4.9 muestra las relaciones geométricas existentes en el cálculo de θ_2. Las Ecuaciones que formarán parte del cálculo de θ_2 irán desde la 4.19 hasta la 4.23.

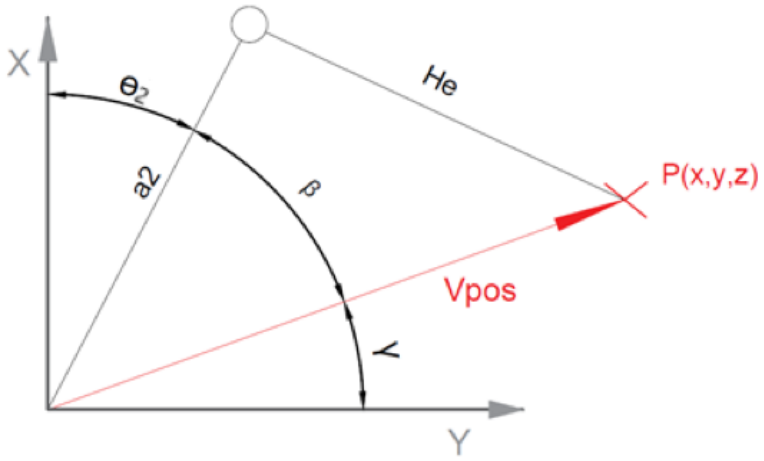


Figura 4.9.- Cálculo de θ_2.

$$r = \sqrt{Px^2 + Py^2} \tag{4.19}$$

$$\gamma = \text{atan}\left(\frac{Pz}{r}\right) \tag{4.20}$$

La Ecuación 4.20 muestra el cálculo de γ como el ángulo que forma el vector posición con el plano horizontal. Para su cálculo, se necesita el uso del módulo de la proyección del vector V_{pos} (r) en el plano horizontal, el cual se ha calculado en la Ecuación 4.19.

$$cos\beta = \frac{m^2 + a2^2 - He^2}{2 \times m \times a2} \tag{4.21}$$

$$\beta = \text{acos}\,(cos\beta) \tag{4.22}$$

$$\theta_2 = \frac{pi}{2} - \gamma - \beta \tag{4.23}$$

La ecuación 4.21 calcula el coseno de β mediante la utilización del Teorema del Coseno, haciendo uso también de He y m, calculados previamente para θ3. Una vez conocida β y γ se puede calcular θ2, apoyándose dicho cálculo en la relación geométrica representada en la Figura 4.8, resultando en las Ecuaciones 4.22 y 4.23.

4.1.3.3.- Cálculo de los ángulos de orientación

Para la orientación se puede realizar un cálculo matricial analítico basándonos en los datos ya conocidos [32]. Este cálculo se basará en la relación mostrada en las Ecuaciones 4.24, 4.25 y 4.26.

$${T_{03}}^{-1} \times T_{06} = T_{36} \tag{4.24}$$

$${T_{04}}^{-1} \times T_{06} = T_{46} \tag{4.25}$$

$${T_{05}}^{-1} \times T_{06} = T_{56} \tag{4.26}$$

Con estas tres ecuaciones de matrices se puede hallar la solución cinemática del problema, escogiendo los elementos de la ecuación que más nos convengan [33]. Como se puede observar, todos los datos de estas ecuaciones son conocidos, exceptuando θ_4, θ_5 y θ_6. Son conocidos porque θ_1, θ_2 y θ_3 son los ángulos encargados de colocar el brazo en la posición deseada, calculados anteriormente. Además, al conocer la orientación buscada se tomará como "dato". La matriz de rotación buscada y que utilizaremos como dato será la que podemos ver en la Ecuación 4.27:

$$R = \begin{pmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix} \tag{4.27}$$

La matriz de rotación representada en la Ecuación 4.27 es la parte de rotación de nuestra matriz homogénea, calculada en la Ecuación 4.6. Esto querrá decir que, conociendo la traslación de T_{06} (posición deseada) y la matriz de rotación (orientación buscada), conocemos T_{06}. Es decir, podemos utilizar las Ecuaciones 4.24, 4.25 y 4.26 para despejar los ángulos θ_4, θ_5 y θ_6. Se puede ver la composición de una matriz homogénea dividida en sus respectivas partes en la Ecuación 4.28. En dicha Ecuación se puede ver la parte de rotación (R_{xy}) y la parte de traslación (t_{xy}) de las que se ha hablado.

$$T_{xy} = \begin{bmatrix} R_{xy} & t_{xy} \\ \emptyset & 1 \end{bmatrix} \tag{4.28}$$

Al tener una ecuación de matrices, para el cálculo de θ_4 se elegirán las ecuaciones más convenientes en cuanto a la facilidad de su resolución. Se encuentran dos elementos en la matriz T_{36} que pueden valer para calcular la posición angular de nuestra cuarta articulación. Estos son los elementos (1,2) y (2,2) que se denominarán en las Ecuaciones 4.29 y 4.30 como E_{12} y E_{22} respectivamente:

$$E_{12} = -\cos(\theta_4) * \sin(\theta_5) \tag{4.29}$$

$$E_{22} = -\sin(\theta_4) * \sin(\theta_4) \tag{4.30}$$

Tomando estos dos elementos y despejándolos en la Ecuación 4.24 se obtendrán el coseno y el seno de 4 que, a partir de ahora, se denominarán como c4 y s4.

$${R_{03}}^{-1} = \begin{pmatrix} \frac{cos(\theta_1+\theta_2+\theta_3)}{2} + \frac{\cos(\theta_2-\theta_1+\theta_3)}{2} & \frac{sin(\theta_1+\theta_2+\theta_3)}{2} - \frac{\sin(\theta_2-\theta_1+\theta_3)}{2} & \sin(\theta_2+\theta_3) \\ -\frac{sin(\theta_1+\theta_2+\theta_3)}{2} - \frac{\sin(\theta_2-\theta_1+\theta_3)}{2} & \frac{cos(\theta_1+\theta_2+\theta_3)}{2} - \frac{\cos(\theta_2-\theta_1+\theta_3)}{2} & \cos(\theta_2+\theta_3) \\ \cos(\theta_2)*\sin(\theta_3)+\sin(\theta_2)*\cos(\theta_3) & \cos(\theta_2)*\cos(\theta_3)-\sin(\theta_2)*\sin(\theta_3) & 0 \end{pmatrix} \tag{4.31}$$

$$c4 = -\frac{{R_{03}}^{-1}(1,1) \times R_{12} + {R_{03}}^{-1}(1,2) \times R_{22} + {R_{03}}^{-1}(1,3) \times R_{32}}{\sin(tetha5)} \tag{4.32}$$

$$s4 = -\frac{{R_{03}}^{-1}(2,1) \times R_{12} + {R_{03}}^{-1}(2,2) \times R_{22} + {R_{03}}^{-1}(2,3) \times R_{32}}{\sin(tetha5)} \tag{4.33}$$

$$\theta_4 = \text{atan}\left(\frac{s4}{c4}\right) \tag{4.34}$$

En la ecuación 4.31 se muestran los componentes T_{03}^{-1} de las tres primeras filas y columnas, correspondientes con la parte de la rotación de la inversa de T_{03} (R_{03}^{-1}) y que permiten resolver θ_4. En las ecuaciones 4.32 y 4.33 se despeja el coseno y el seno del ángulo θ_4 para los elementos E_{12} y E_{22} en la ecuación de matrices 4.24. Por último, se despeja θ_4 en la ecuación 4.34 utilizando el seno y el coseno calculados.

Siguiendo la misma lógica para los siguientes ángulos, se procederá al cálculo de θ_5 y θ_6. Para θ_6 se utilizan los elementos (3,1) y (3,3) de la matriz T_{36} que se llaman, en este caso, E_{31} y E_{33}. Desde la Ecuación 4.35 hasta la 4.39 se mostrarán los cálculos de θ_6.

$$E_{31} = -\cos(\theta_6) * \sin(\theta_5) \tag{4.35}$$

$$E_{33} = \sin(\theta_6) * \sin(\theta_5) \tag{4.36}$$

$$c6 = -\frac{{R_{03}}^{-1}(3,1) \times R_{11} + {R_{03}}^{-1}(3,2) \times R_{21} + {R_{03}}^{-1}(3,3) \times R_{31}}{\sin(\theta_5)} \tag{4.37}$$

$$s6 = \frac{{R_{03}}^{-1}(3,1) \times R_{13} + {R_{03}}^{-1}(3,2) \times R_{23} + {R_{03}}^{-1}(3,3) \times R_{33}}{\sin(\theta_5)} \tag{4.38}$$

$$\theta_6 = \text{atan}\left(\frac{s6}{c6}\right) \tag{4.39}$$

Por último y para terminar se calcula θ_5. Para θ_5 se escoge el elemento (3,2). Los cálculos para hallar θ_5 se mostrarán desde la Ecuación 4.40 hasta la 4.43:

$$E_{32} = -\cos(tetha5) \tag{4.40}$$

$$c5 = -{R_{03}}^{-1}(3,1) \times R_{12} - {R_{03}}^{-1}(3,2) \times R_{22} - {R_{03}}^{-1}(3,3) \times R_{32} \tag{4.41}$$

$$s5 = \sqrt{1 - c5^2} \tag{4.42}$$

$$\theta_5 = \text{atan}\left(\frac{s5}{c5}\right) \tag{4.43}$$

La cinemática inversa del robot ha quedado pues descrita, calculando los ángulos que sitúan al robot en la posición deseada (θ_1 θ_2 y θ_3) y aquellos que orientan el efector final (θ_4 θ_5 y θ_6).

4.1.4.- TRABAJO DEL ROBOT REAL

Se ha hablado en este proyecto de la condición de paletizador del brazo robótico empleado. Esta condición implica que los dos motores se localizan en el mismo eje que la segunda articulación del brazo, en la base. Dos eslabones extra son utilizados para formar un mecanismo en forma de palalelogramo que permite el control de la tercera articulación. Se puede ver una representación de este mecanismo en la Figura 4.10, así como la presencia del mecanismo en el robot real en la Figura 4.11.

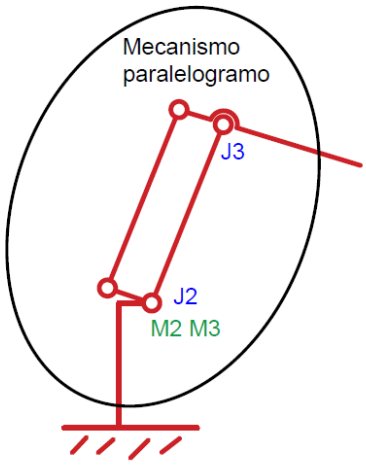


Figura 4.10.- Representación gráfica del mecanismo en forma de palalelogramo característico de los robots paletizadores.

Figura 4.11.- Visualización del mecanismo trabajando en el robot real [31].

Se puede observar cómo el eje de la articulación 2, que se ha denominado en la Figura 4.12 como J2, es compartido por los motores M2 y M3. También se puede ver la estructura cinemática cerrada descrita, que permite al motor M3 tomar el control de la articulación J3. Esta configuración robótica provocará un funcionamiento diferente al que se produce en brazos robóticos sin este tipo de mecanismo.

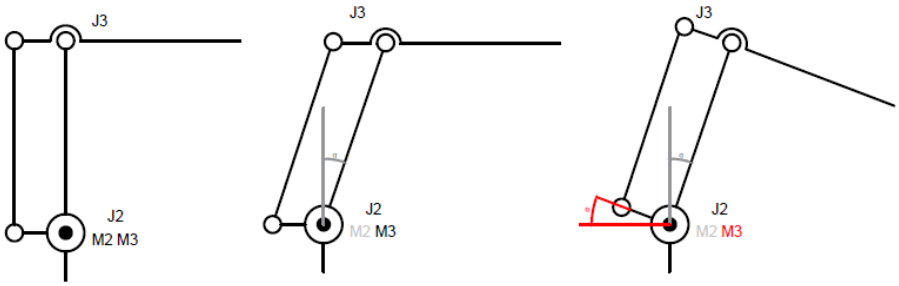


Figura 4.12.- Representación del movimiento del brazo debido a la presencia del paralelogramo.

En la Figura 4.12 se ve dicho funcionamiento en el que se muestra cómo, cuando el motor M2 desplaza el eslabón 2 un ángulo α, el eslabón 3 mantiene su orientación. Esto se debe a que el eslabón paralelo al eslabón 3, al no producirse ningún giro en el motor M3, mantiene su orientación. La acción del mecanismo provoca que ambos mantengan siempre una misma orientación. La articulación J3, se comporta en la práctica compensando el

movimiento de J2. En la segunda imagen de la Figura 4.12, J3 sin la cinemática cerrada estaría en una posición -α, por lo que se observa con claridad el efecto compensatorio generado por el mecanismo paralelogramo en búsqueda de la invariabilidad de la orientación; todo esto sin el accionamiento de ningún motor. También se puede apreciar cómo, en la tercera imagen de la Figura 4.12, se puede compensar dicho movimiento, girando ambos motores el mismo ángulo y actuando como un brazo robótico que careciera del mecanismo del que hablamos.

En la práctica, esto se traduce en que la carga del movimiento de los ángulos calculados (θ_2 y θ_3) recaerá íntegramente sobre el motor conectado al eslabón que cierra el mecanismo paralelogramo, es decir, en M3. Esto lo diferencia de un brazo robótico de 6 grados de libertad convencional ya que, en este último, la responsabilidad del movimiento de cada ángulo calculado recaería únicamente en el motor asociado a su movimiento.

Una vez explicado el mecanismo se procede a la descripción de los límites físicos de las articulaciones del robot, que restringen su movimiento limitando la zona hábil de trabajo. Estos límites se pueden observar en la Tabla 4.2.

Articulación	Ángulo	Rango de movimiento	
Base	Θ_1	-90º	+90º
Hombro	Θ_2	-60º	+70º
Codo	Θ_3	-10º	+145º
Antebrazo	Θ_4	-90º	+90º
Muñeca	Θ_5	-30º	+60º
Muñeca 2	Θ_6	-90º	+90º

Tabla 4.2.- Límites físicos de las articulaciones del robot.

Una vez conocidos los rangos de movimiento de cada una de las articulaciones, contando a su vez con el mecanismo paralelogramo y la cinemática directa del robot, se procederá a obtener el espacio de trabajo de nuestro robot paletizador. Esto se realizará siguiendo las siguientes fases:

1. Se cambiará el ángulo Θ_1 desde su ángulo mínimo hasta su máximo.

2. Para cada ángulo Θ_1 se cambiará Θ_2 en todo su rango.

3. El mecanismo particular del paletizador limitará el movimiento del eslabón 3. Debido al eslabón paralelo controlado por el motor 3, la orientación del eslabón 3 permanecerá constante. Los puntos alcanzables estarán limitados por la posición de Θ_3, además de que existirá un ángulo mínimo y máximo entre la diferencia de ambos ángulos Θ_2 y Θ_3. Esto será provocado por la colisión en ciertos puntos de los componentes del mecanismo.

Siguiendo estos pasos, se ha realizado el dibujo de la nube de puntos del plano XOZ que se puede observar en la Figura 4.13. Junto con la nube de puntos en verde, se ha dibujado un plano paralelo al suelo de trabajo. Este plano permite conocer la zona de trabajo del robot en Z=0 suponiendo un tamaño del efector final de 6 cm. Esta zona de trabajo se puede encontrar dibujada vista desde arriba en la Figura 4.14.

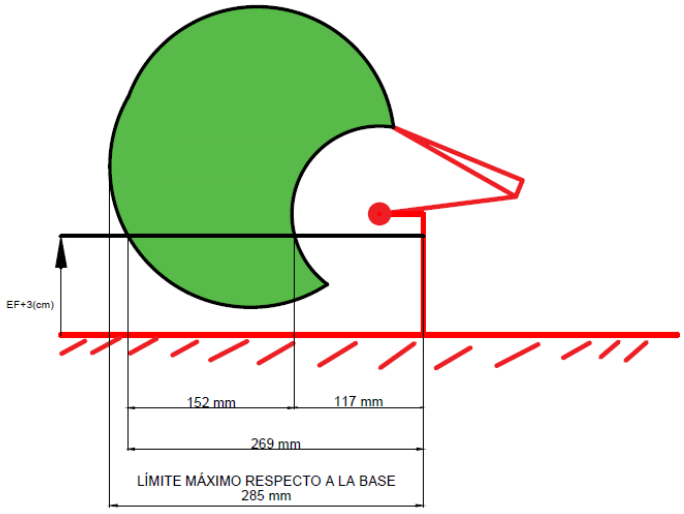


Figura 4.13.- Zona de trabajo idónea del robot vista de perfil.

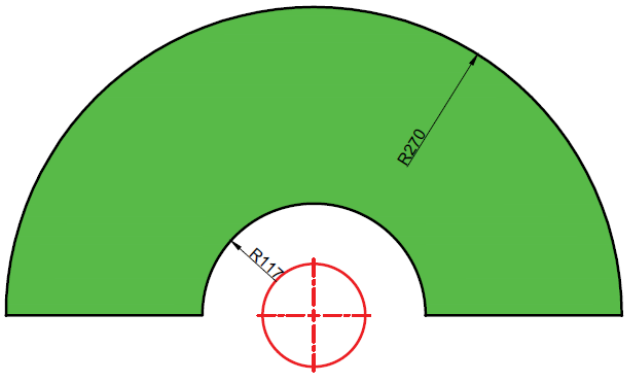


Figura 4.14.- Zona de trabajo del robot en Z=0 suponiendo presencia de efector final (en mm).

4.1.5.- APLICACIÓN DE LA CINEMÁTICA PARA LA LABOR DE PICK AND PLACE

A lo largo del trabajo, con la intención de demostrar el funcionamiento del robot en la práctica (cinemática inversa y calibración de los ángulos del motor) se ha desarrollado una aplicación de demostración con círculos de colores. Por ello, para mejorar y optimizar la función de recogida realizada, se detallan a continuación las particularidades de la acción a desempeñar:

- El objeto serán canicas, las cuales al ser esféricas no tendrán ningún tipo de orientación predefinida. Al ser así, estando localizadas en un mismo plano horizontal y sin obstáculos, los giros de las articulaciones 4 y 6 pueden mantenerse intactos en la posición inicial 0, solo teniendo que situar la posición de la articulación 5 encargada de mantener la garra perpendicular a la superficie de trabajo.

- Las canicas se pueden mover. Esto hace que sea fundamental realizar un algoritmo de *pick & place* que, conociendo la posición de las canicas y utilizando la parte de visión (tal y como se verá más adelante) pueda llegar a recoger la canica en cualquier momento, una vez esta se encuentre parada.

- Los depósitos de las canicas estan localizados a los lados izquierdo y derecho del robot, diferenciando el lado de depósito según el material del que estén construidas. El depósito rojo se encuentra localizado en el punto (0,20) cm, mientras que el verde se encuentra localizado en el punto (0,-20) cm.

Teniendo en cuenta todas estas características la conclusión es que el robot solo dependerá del ángulo θ5 para situar la orientación, resultando en θ4 y θ6 nulas (en la posición inicial del robot) con el objetivo de simplificar el cálculo de la cinemática inversa. El robot pasará a ojos del procedimiento a tener 4 grados de libertad.

$$\theta 5 = \frac{pi}{2} - \theta 2 - \theta 3 \tag{4.44}$$

En la Ecuación 4.44 se puede observar la simplificación de la obtención del ángulo θ5 para la labor de este proyecto. En el caso de que se necesitara colocar el efector final en una orientación más compleja, se acudiría a las fórmulas destinadas a la orientación que ya han sido descritas con anterioridad.

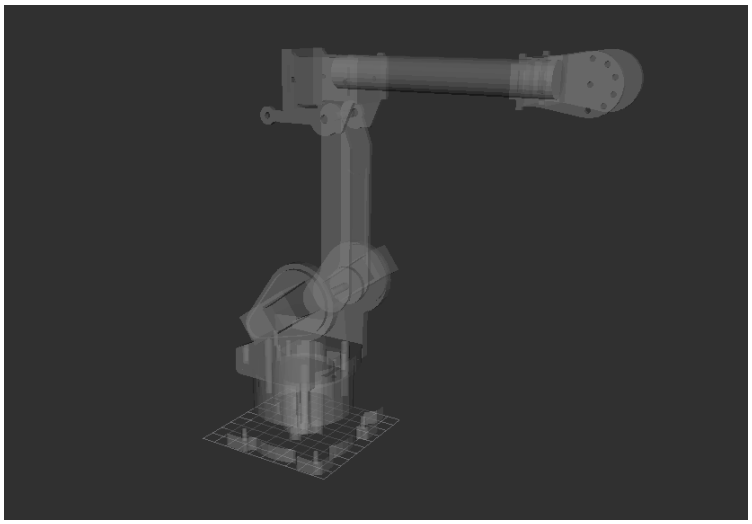


Figura 4.15.- Representación de nuestro robot en URDF visualizado en RViz.

A continuación, se describe la secuencia del tradicional algoritmo de *pick & place*:

1. El robot estará localizado en una posición inicial en la cual va a esperar a que haya alguna información para la recogida de canicas.

2. Recibe la información de la localización de los objetos dispuestos en la mesa para ser recogidos y se lanza para la recogida del primero, localizándose unos centímetros por encima. En este punto esperará un cierto tiempo y abrirá la garra (en el caso del robot utilizado no posee garra por lo que esta parte se obviará).

3. Desde esa posición descenderá para la recogida de la canica. Cuando esté en la posición cerrará el efector final. Se volverá a elevar hasta la posición del apartado dos y se desplazará hasta unos centímetros por encima del depósito correspondiente.

4. Descenderá y abrirá la garra, depositando la canica en el depósito y retornando a la posición inicial.

Como se detalló con anterioridad esta técnica permite trabajar en el plano de forma perpendicular a la mesa para labores simples de *pick and place* como la detallada, evitando en lo máximo posible el movimiento o la colisión con el resto de los objetos situados en la mesa. Esta estructura será utilizada con unas pequeñas modificaciones en la secuencia llevada a cabo en el presente trabajo.

4.2.- RASPBERRY PI

El modelo del que se dispone en la planta es la Raspberry pi 4B, la versión de 4GB de RAM. Este dispositivo contará con la instalación de linux y ROS Kinetic. En concreto, cuenta con la instalación en la tarjeta SD de 64 GB de una imagen ISO que lleva una distribución de un sistema operativo linux con ROS Kinetic ya integrado. Esta imagen se conoce con el nombre de Ubiquity Robotics y fue desarrollada por la marca de robótica Ubiquity. Permite disfrutar de ROS en la Raspberry Pi sin falta de instalar muchos de los paquetes [37].

En la Raspberry Pi se corren los nodos relacionados con la visión, al estar directamente conectada con la Raspicam, de la que se obtiene la imagen del entorno de trabajo. También se encarga de los nodos relacionados con el accionamiento de los motores

y del cálculo de la cinemática inversa. Se han escogido estos procesos para la Raspberry debido a que son actividades más cercanas al robot, mientras que el ordenador carga con la parte computacional más grande, correspondiente a *Moveit!*.

La distribución final se ha pensado para que todas las conexiones dependientes de cables vayan a la Raspberry y el ordenador pueda estar localizado fuera de la mesa de trabajo, conectado al mismo Wifi y siendo ciertamente independiente de la planta de trabajo.

4.3.- ARDUINO MEGA

El Arduino actúa con su propio programa mediante el uso de la librería para Arduino ros_lib y del nodo Rosserial, tal y como se ha explicado en las herramientas utilizadas por ROS en el Capítulo 3. Podrá actuar como un nodo más de nuestro sistema robótico estando conectado al puerto USB de la Raspberry. El Arduino se comunica con la placa driver PCA9685 mediante la utilización de una conexión de tipo I2C, tal y como se puede ver en el esquema de la Figura 4.2, con motivo de controlar la posición de los motores.

4.4.- DRIVER PCA9685 ADAFRUIT

El driver del que se dispone es un dispositivo controlable por I2C y que se alimenta con una entrada de 5V con la que podrá generar la señal PWM para alimentar hasta a 16 motores. Esto es de gran utilidad cuando se quiere realizar el control de varios servomotores a la vez, como en el caso del robot que se está utilizando en este trabajo.

Trabajar con este dispositivo es muy cómodo, debido al gran número de librerías disponibles para su trabajo en internet. Se ha escogido el trabajo coordinado con la placa Arduino Mega debido a lo intuitivo que resulta trabajar con la librería mencionada en el Capitulo 3. Se conectará por tanto al I2C del Arduino, con las 6 primeras salidas (de la 0 a la 5) conectadas a los motores del robot, como se puede apreciar en la Figura 4.16.

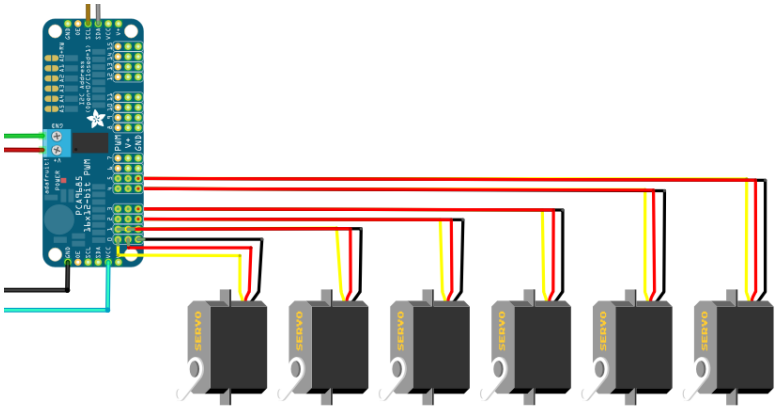


Figura 4.16.- Conexiones del driver con los motores del robot.

4.5.- CÁMARA

Contamos con una cámara Raspicam v2.1 que se conecta con la Raspberry por un bus CSI-2, específicamente diseñado para la conexión de este tipo de cámaras. La raspicam la utilizaremos dentro del raspicam_node y funcionará como un nodo más, tal y como se ha explicado en el funcionamiento de la herramienta en el Capítulo 3. En el siguiente Capítulo, destinado al sistema de localización de objetos, se ofrece una visión mucho más amplia y detallada de cómo funciona este nodo en la práctica.

5. Sistema de localización de objetos

El sistema de localización de objetos proporciona la posición en el plano del objeto que hay que recoger. A la hora de realizar una demostración visual de cómo funciona el robot se han utilizado círculos impresos de colores en lugar de canicas, dada la imposibilidad de trabajar en el laboratorio de la EPI debido a la situación provocada por el COVID19. Estos círculos de colores simulan las canicas esféricas, siendo reducidas sus características particulares a un mero color. En un futuro, estas canicas deberán ser detectadas en un programa de visión más complejo, pero igualmente integrable dentro del presente sistema de control.

La parte de visión del sistema es la encargada de recoger la imagen de la mesa de trabajo y localizar nuestros objetos en el plano de trabajo (nuestro Z=0 será la mesa). Para poder extraer la posición de los círculos de colores desde la imagen proporcionada por la cámara, se utilizó el siguiente algoritmo de detección:

1. La imagen se convierte al modelo de color HSV.

2. Se realiza una umbralización o thresholding con los parámetros del color de los círculos a localizar, poniendo a 1 todos aquellos píxeles que estén dentro del rango de color precisado.

3. Se detectan los contornos en la imagen binaria obtenida en el paso 2.

4. Se dibujan los recuadros o BoundingBoxes que encuadran a nuestros círculos, obteniendo así sus propiedades de forma, tamaño y localización dentro de la imagen.

5. Se realiza una transformación homográfica que transforma las coordenadas en píxeles de la imagen en las coordenadas en cm del plano en la mesa, hallando así la localización de los círculos.

6. Se aprovecha la información proporcionada por estas BoundingBoxes, hallando los centros de cada círculo dentro de la imagen.

Como requisito indispensable para que el proceso de detección obtuviera buenos resultados, se lleva a cabo una corrección de las distorsiones de la imagen capturada por la cámara. Además, se debe ajustar el rango de color de los círculos para la umbralización, de forma que se obtengan buenos resultados de esta. Por último, antes de cada demostración, se calcula la matriz Homográfica 3x3 que relaciona los píxeles con el plano de la mesa de trabajo.

5.1.- CALIBRACIÓN DE LA CÁMARA

La lente utilizada tiene una distancia focal pequeña. Esto produce que, al tener un ángulo de visión grande, se produzca una distorsión de tipo barril en la imagen capturada. Esta distorsión no nos permitirá ver la imagen como un plano bidimensional, sino que alterará las características y dimensiones de los objetos que encontramos en ella. El tipo de distorsión de la cámara solo permite que los objetos del centro de la imagen se vean cercanos a la proporción real con respecto al resto de la imagen. El resto de la imagen irá distorsionándose más en cuanto más nos alejemos del centro [38]. Esta distorsión tipo barril se puede entender mejor en la comparación de la Figura 5.1, en la que se representa lo que supone con respecto a una imagen no distorsionada.

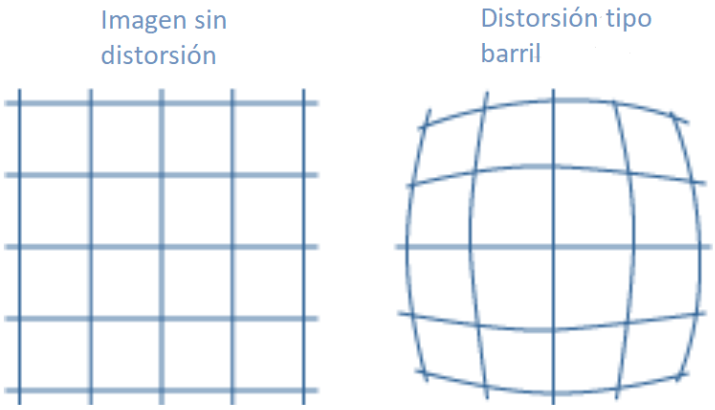


Figura 5.1.- Distorsión tipo barrilete.

Es muy importante para el proyecto eliminar estas distorsiones ya que, al trabajar con distancias y transformaciones de planos, se necesita que todos los objetos y distancias se visualicen con sus proporciones reales en todas las partes de la imagen. En otro caso, dichas distorsiones provocarían como resultado un error en el cálculo de las distancias; siendo los errores más grandes en cuanto más se aleje uno del centro de la imagen). Como resultado, esto originaría peores resultados en la obtención de las distancias. Esta matriz de distorsión se muestra en la Ecuación 5.1.

$$K = \begin{pmatrix} fx & s & x0 \\ 0 & fy & y0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.1}$$

En la Ecuación 5.1 se describen varios parámetros que conforman la matriz de distorsión, definidos de la siguiente manera [39]:

- fx y fy distancia focal.

- s o skew modela la inclinación del plano imagen respecto al eje óptico como una pérdida de perpendicularidad de los ejes en píxeles.

- x0 e y0 representan la posición del punto principal en la imagen.

Para la calibración de la cámara contamos con el uso de una herramienta dentro del nodo raspicam_node. Se llevó a cabo un proceso de calibración automatizado con la ayuda de un tablero de ajedrez impreso colocado en un tablón. Cambiando las posiciones y orientaciones del tablero con respecto a la cámara, el raspicam_node va adquiriendo los diferentes datos de orientación y posición. Posteriormente, se utilizan dichos datos para la aplicación del algoritmo de Zhang, que calcula las distorsiones ópticas y la calibración de la cámara. Estos datos se registrarán por defecto en un archivo de configuración .yaml y se aplicarán automáticamente en la corrección automática de la imagen, cada vez que utilicemos la cámara en ROS. En la Figura 5.3 se puede ver la detección del tablero durante el proceso de calibrado.

Para comenzar la calibración, se lanza el nodo raspicam_node, con el argumento enable_raw puesto a True. Esto permite al nodo publicar la imagen en bruto de forma que, desde ROS (mediante la herramienta rqt_image_view) se pueda visualizar la imagen. Además, esto le permite a ROS procesarla como imagen, pudiendo aplicar nuestros algoritmos de percepción. Por otro lado, se lanza un nodo desde ROS que se encarga de llevar a cabo la calibración previamente mencionada. Para ello, se lanza camera_callibration tomando como argumento: el programa que va a llevar a cabo la calibración (cameracalibrator.py), el tópico de la imagen de donde se van a tomar los datos (/raspicam_node/image), el número de las esquinas interiores de nuestro tablero (en este caso 8x6) y el tipo de patrón a detectar con su medida en m (cuadrados de 0.025). En la Figura 5.2 se muestra el output del programa de calibración mientras está en funcionamiento.

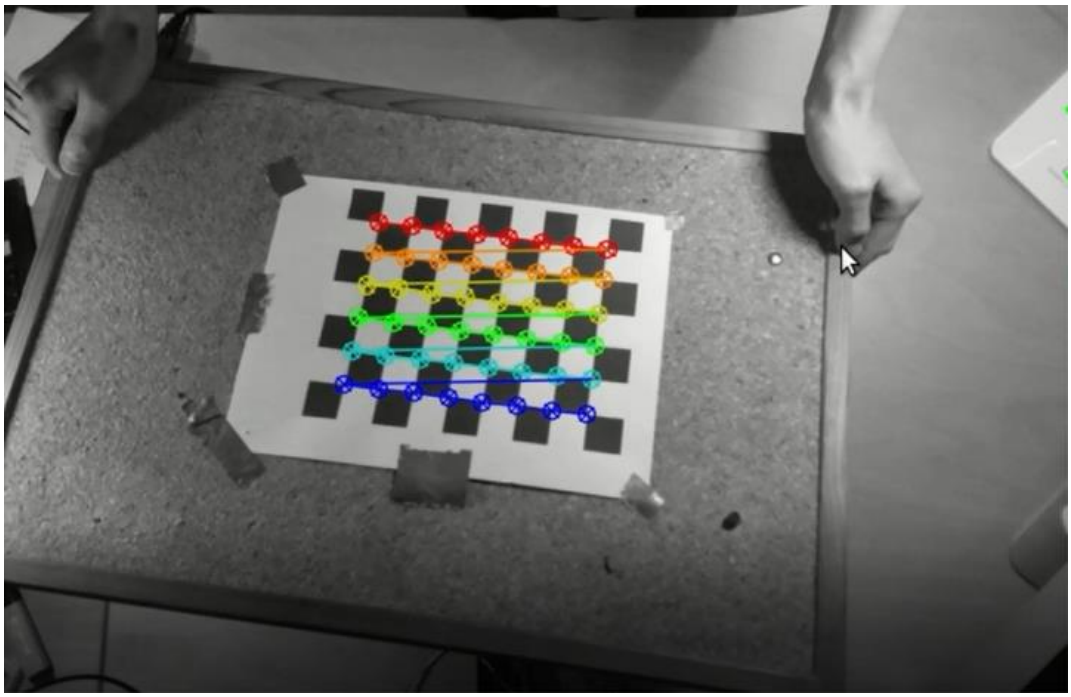


Figura 5.2.- Foto demostración de la detección de las esquinas del tablero de ajedrez durante la calibración de la cámara.

5.2.- HOMOGRAFÍA

Con motivo de hallar la relación entre los píxeles que se encuentran en la foto con las distancias en la mesa real, una vez eliminadas las distorsiones de la cámara, se buscó un

método que relacionase ambos planos. Los píxeles de la imagen se miden por convenio desde la esquina superior izquierda y los mm, para hallar la homografía, se miden desde la esquina inferior derecha de nuestro tablero de ajedrez. El eje X del plano de la mesa P2 se corresponde con el eje vertical, que sube hacia la parte superior de la imagen. El eje Y es el que apunta hacia la izquierda de la imagen. Se puede ver un modelo que esquematiza la proyección entre el plano de la imagen de la cámara (P1) y el plano de la mesa de trabajo (P2) en la representación de la Figura 5.4.

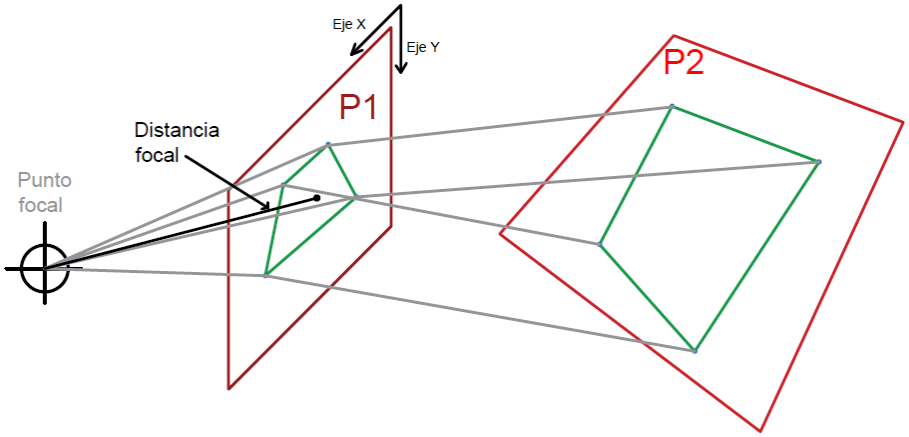


Figura 5.3.- Proceso general de proyección entre el plano de la cámara P1 y un plano P2 de la escena.

Se consideró hallar una relación de distancias en X y en Y mediante una transformada afín que nos proporcionara una relación de tipo lineal entre los centímetros X e Y de la mesa y los píxeles X e Y de la imagen [38]. Pese a ser una opción que funcionaba aproximadamente bien cuando la cámara estaba lo suficientemente paralela al plano de trabajo, la imposibilidad de mantener dicha posición, requisito necesario para este tipo de transformada, la convirtieron en una opción inviable.

Finalmente se optó por una transformación Homográfica, una conversión entre dos planos en la que se halla una correspondencia expresada en forma de matriz 3x3, la cual permite un cambio entre planos de una manera sencilla. La homografía modela un proceso general de proyección entre dos planos.

$$\tilde{P}_2 = H \times \tilde{P}_1 \tag{5.2}$$

$$\tilde{P}_1 = H^{-1} \times \tilde{P}_2 \quad (5.3)$$

En las Ecuaciones 5.2 y 5.3 encontramos las fórmulas que nos permiten proyectar los puntos para obtener sus coordenadas representadas en el otro plano.

En el desarrollo de este trabajo la transformación homográfica se utiliza para la medición de distancias y corrección de la posible distorsión de perspectiva. Utilizamos el mismo tablero de ajedrez de la calibración, situado en nuestra mesa (Z=0), para obtener sus esquinas interiores 8x6 y asociarlas con sus medidas correspondientes en el tablero real (con respecto al eje de referencia tomado para la transformada). Las esquinas internas del tablero, las cuales se han detectado con OpenCV Python, constituyen una cantidad de puntos suficiente para describir el plano de la mesa en el que se quieren detectar nuestros círculos. Esto es importante, puesto que cuantos más puntos se utilicen del plano, mejor será el resultado.

La detección del plano de la mesa se realizó con un *script* que se incluye dentro del Anexo 1 y que muestra, como salida, las esquinas localizadas en la propia imagen. Este programa imprime por terminal, a su vez, la matriz homográfica. Esta matriz se integrará, cada vez que sea calculada, en el programa de visión utilizado. Se cambia la matriz, en el programa de visión, cada vez que haya cambiado la orientación o la posición de la cámara.

En el trabajo se utiliza el script “Homografia.py”, que realiza el cálculo de la matriz homográfica que vamos a emplear. Para realizar el cálculo, primero detecta las esquinas interiores del tablero que, al ser una hoja de papel, está localizado en el plano de la mesa. Posteriormente, las asocia con las medidas con respecto a la esquina inferior izquierda del tablero en cm y asocia cada esquina a dichas medidas, calculando la transformación homográfica entre ambos planos. El resultado de la utilización de este programa se puede observar en la Figura 5.4. A la derecha de la Figura, sale la imagen con el dibujo de las esquinas detectadas. En el terminal, situado a la izquierda de la Figura, se imprime la matriz Homográfica de transformación.

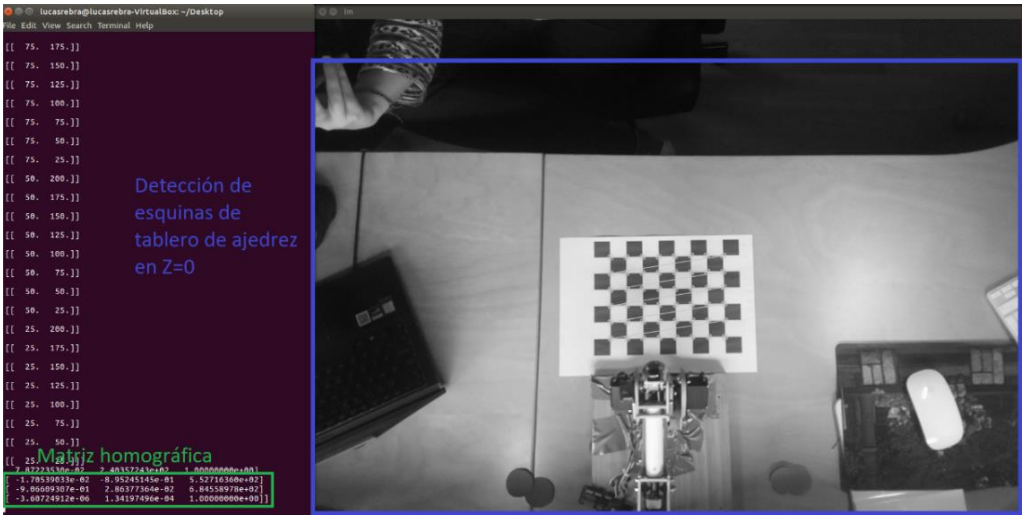


Figura 5.4.- Resultados del script que calcula la matriz Homográfica utilizando imagen actualizada.

Mediante esta Homografía, se obtiene la transformación de la imagen (en pixeles) en las coordenadas (en cm) del plano de la mesa, pudiendo así localizar los objetos. Como estamos trabajando en coordenadas homogéneas, se transforman las coordenadas u y v (correspondientes con los píxeles X e Y en la imagen tomada por la Raspicam) en coordenadas homogéneas. Esto resulta en un vector vertical 3x1 en el que los primeros dos elementos se corresponden con las coordenadas ***u*** y **v** y el tercer elemento sea un 1, tal y como se puede ver en la Ecuación 5.4.

$$Pos_{imagen}(pixels) = (u, v) \;\rightarrow\; Homogéneas = \begin{pmatrix} u \\ v \\ 1 \end{pmatrix} \tag{5.4}$$

El vector transformado a coordenadas homogéneas se multiplica por la matriz Homográfica para corregir la perspectiva, obteniendo los puntos de nuestra imagen en coordenadas homogéneas. Esta multiplicación da un componente Z que no será igual a 1 en la salida. Esto se puede ver en la Ecuación 5.5 correspondiente con el cálculo de la transformación homográfica de un punto de la imagen. Para obtener las distancias en cartesianas, se tendrán que normalizar las coordenadas X, Y y Z del vector resultado de la transformación, tal y como se puede ver en la Ecuación 5.6. Esto se realiza con el objetivo de deshacer el paso, transformando las coordenadas de homogéneas a cartesianas. Las coordenadas X e Y obtenidas después de esta normalización son las coordenadas en cm con

respecto al punto de referencia escogido (el eje inferior derecho del tablero). Dentro del nodo de percepción, se transforman esas coordenadas y se referencian al eje sobre el que se ha construido la descripción DH del robot. En la Figura 5.5 están dibujados ambos ejes de coordenadas con su respectiva relación de distancias.

$$\begin{bmatrix} \lambda x \\ \lambda y \\ \lambda \end{bmatrix} = H \times \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{5.5}$$

$$\bar{X}(cm) = \mathcal{P}_z(p) = \begin{bmatrix} \lambda x/\lambda \\ \lambda y/\lambda \\ \lambda/\lambda \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \tag{5.6}$$

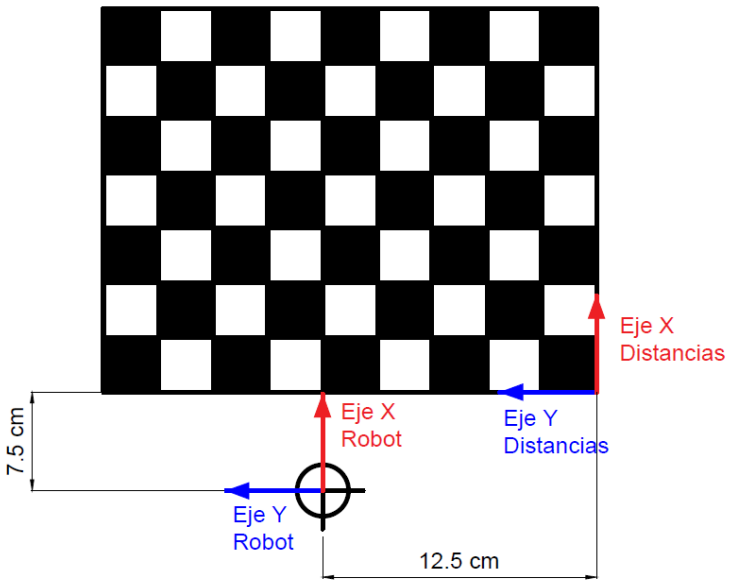


Figura 5.5.- Relación de distancias entre el eje de referencia tomado para el cálculo de distancias y el eje de coordenadas escogido para el primer eslabón de su descripción DH.

5.3.- RECONOCIMIENTO DE OBJETOS

5.3.1.- UMBRALIZADO O THRESHOLDING

Para el reconocimiento de los círculos se trabaja con la imagen en el espacio de color HSV. HSV (Hue, Saturation, Value) define un modelo de una transformación no lineal de coordenadas polares basado en el espacio de color RGB (Red, Green, Blue). Este modelo es utilizado para la realización del umbralizado debido a que el modelo RGB, al medir la cantidad de cada uno de los tres colores primarios en un determinado píxel, es verdaderamente difícil de umbralizar, dando lugar a una mayor cantidad de ruido. Esto es debido a que gran parte de los colores que encontramos en la realidad están compuestos por todos los colores primarios. El modelo HSV permite hacer un *thresholding* o umbralizado, según sus 3 parámetros, con una precisión mayor que si se hiciera con RGB.

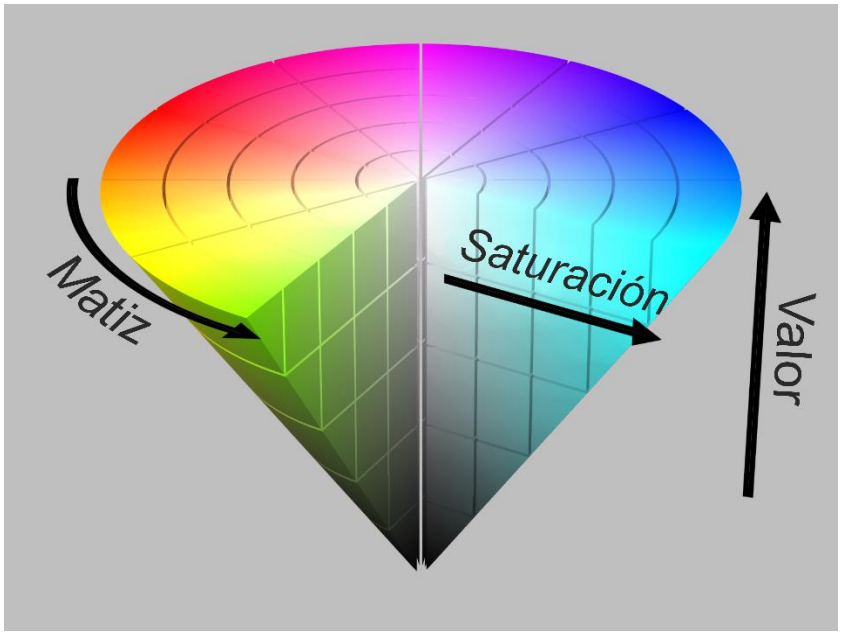


Figura 5.6.- Representación del espacio HSV.

En la Figura 5.6 se puede ver la representación del modelo HSV en el que H y S dan el tono y la saturación del color en un sistema de coordenadas polares en el que H será el ángulo y S la distancia al origen. V representará la altura en el eje blanco-negro[38].

El umbralizado, basándose en los valores HSV mínimos y máximos de cada parámetro (que irán de 0 a 255), detectará aquellos píxeles que se encuentren dentro de los mismos; que cumplan el criterio. Posteriormente, se convertirá la imagen origen en una binaria, en la que los píxeles que cumplan la condición se pongan a 1 y el resto a 0 [38].

Los valores HSV de los umbrales se determinan experimentalmente. Para ello se ha desarrollado un programa que aplica un *thresholding* a una imagen. Este programa permite regular los parámetros Hmin, Hmax, Smin, Smax, Vmin y Vmáx, pudiendo visualizar la imagen resultado como salida. Los parámetros pueden ser introducidos mediante una pequeña interfaz de usuario con una barra, que se moverá de 0 a 255, para cada uno de los seis parámetros. La imagen de salida, una vez ajustados los rangos del umbralizado, debería ser similar a la que podemos visualizar en la Figura 5.7.

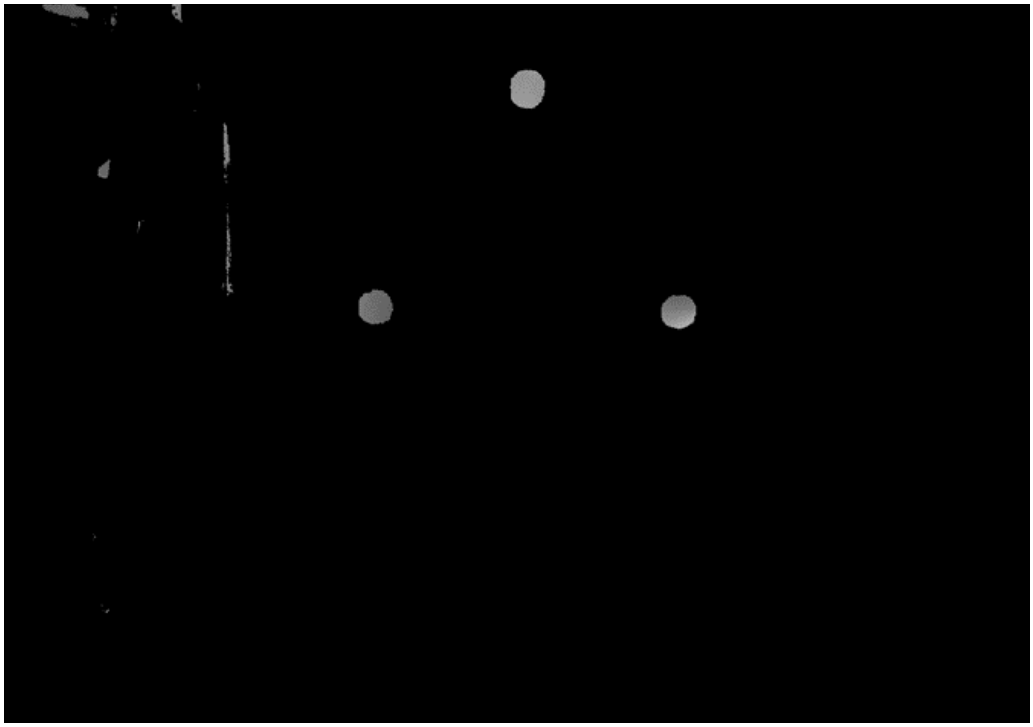


Figura 5.7.- Resultado del thresholding filtrando los círculos del color buscado.

5.3.2.- DETECCIÓN DE CONTORNOS

Una vez localizados nuestros objetos, se deben obtener los contornos, es decir, los límites de nuestras áreas de color blanco (con el resto de la imagen binaria); los puntos en los que nuestra imagen binaria pasa de 1 a 0. Para realizar esta detección, OpenCV utiliza el algoritmo de *Suzuki and Abe*, basado en el seguimiento de contornos de imágenes binarias [40].

Después de obtener los contornos de nuestros objetos, se utiliza la imagen resultante para obtener las *BoundingBoxes*. Estos rectángulos nos ofrecen las magnitudes de nuestro objeto, definiendo así sus propiedades y nos permiten localizar el centro de nuestro círculo. Por último, se determina si el área encontrada es suficiente para considerar que la forma es el objeto buscado, es decir, eliminando todos aquellos colores similares que se consideren ruido en el programa.

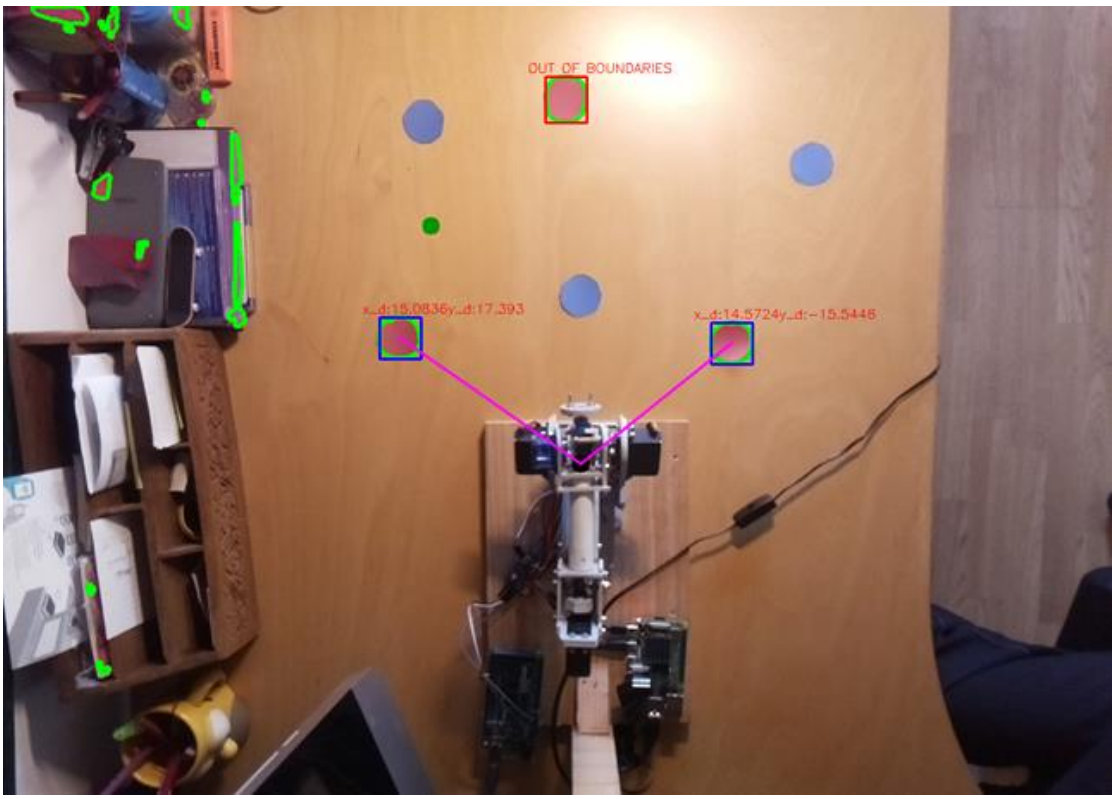


Figura 5.8.- Imagen de salida del nodo de percepción con los contornos, bounding boxes y distancias dibujadas.

En la Figura 5.8. se ve representada una imagen de salida del programa implementado en ROS, aplicado particularmente a la detección de círculos rojos. Los contornos de los círculos saldrán dibujados en verde. También aparecerán las *Bounding Boxes* rectangulares de los objetos dibujadas en azul. El centro de nuestros objetos estará unido mediante una línea con el eje de referencia del robot, lo que hace que la detección sea más visual.

6. Integración y control del brazo robótico en ROS

El propósito de este trabajo es integrar el robot Sainsmart 6dof, para que funcione dentro del sistema operativo robótico ROS. A lo largo de este capítulo, se expone la estructura que conforma nuestro sistema global de control del robot. El robot real, y sus actuadores, funcionarán haciendo uso de las posiciones angulares generadas por el planificador de trayectorias *Moveit!*. Gazebo utilizará también esas mismas coordenadas para realizar la simulación del robot. Además, la interfaz *hardware* encargada de la simulación de Gazebo devolverá *feedback* simulado, permitiendo el control del robot real; *Moveit!* necesita feedback para la planificación de trayectorias y los motores del robot no tienen sensores de posición. Con todo ello, se implementó una demostración que utiliza la percepción de círculos de colores y desarrolla una labor de *pick & place* utilizando un nodo basado en la librería Python de ROS MoveitCommander. Esta librería hace uso de las aplicaciones integradas en el move_group de Moveit! para la generación de trayectorias, situando el brazo robótico en las posiciones de la secuencia *pick & place.* Se puede ver una descripción global del sistema en la Figura 6.1, con todos los nodos que conforman el sistema robótico desarrollado en este trabajo.

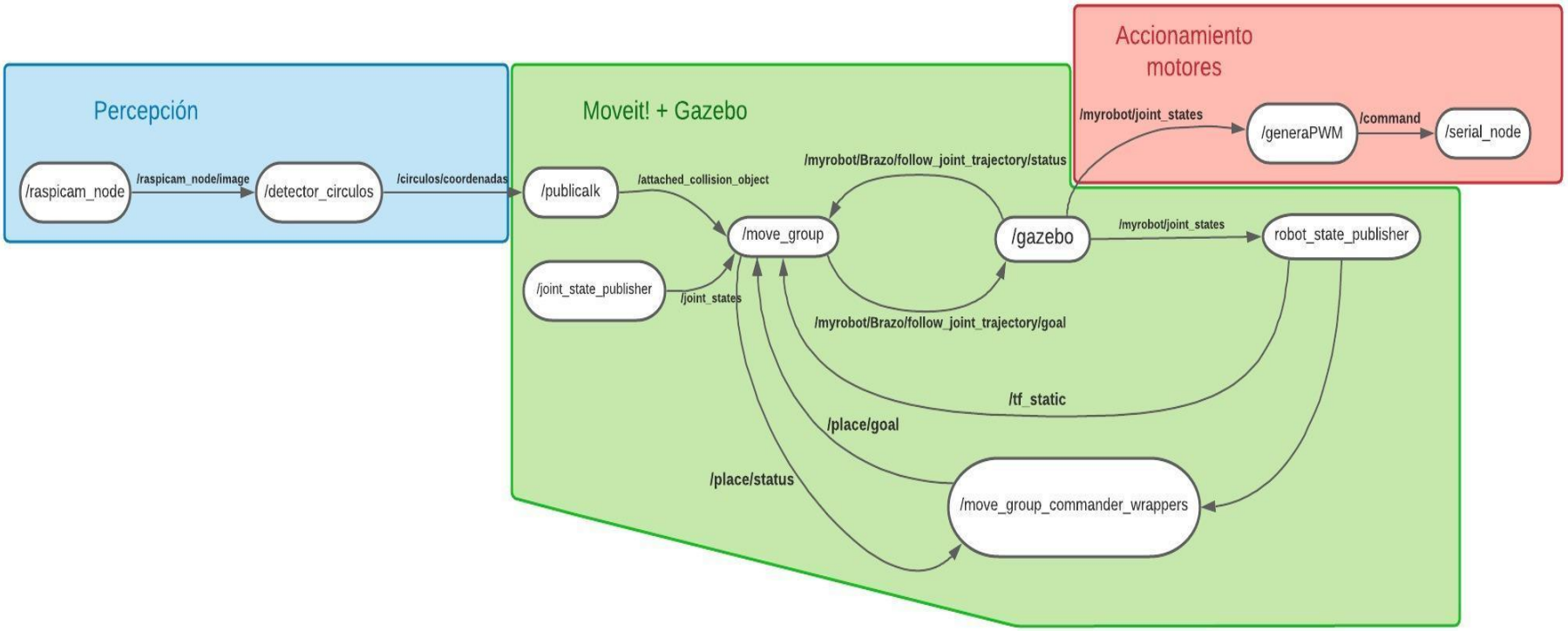


Figura 6.1.- Esquema del funcionamiento global del sistema robótico.

Como se puede observar en la Figura 6.1, se ha dividido el sistema robótico en tres partes: subsistema encargado de la Percepción, subsistema encargado de la planificación de las trayectorias y subsistema encargado del accionamiento de los motores del robot.

El subsistema de percepción cuenta con la presencia de dos nodos. El primero es el nodo raspicam_node, que integrará la cámara y publica la imagen en el tópico /raspicam_node/image. Esta imagen tendrá las distorsiones corregidas utilizando los datos obtenidos en la calibración. El nodo detector_circulos se subscribe para la localización de los círculos en la imagen utilizando el algoritmo explicado en el Capítulo 5, publicando la posición de los círculos clasificados según su color estructurados en el mensaje de tipo Vectorpos.

En el subsistema encargado de la planificación de trayectorias tendremos, por un lado, el nodo publicaIk que se encarga de generar, subscribiéndose al tópico de las posiciones de los círculos, la secuencia de *pick & place* de nuestra demostración. Lo hace comunicándose con el move_group mediante el uso de la librería *MoveitCommander* y con la ayuda de un *wrapper* que integra esas funcionalidades en Python. El move_group utiliza una acción de tipo *JointTrajectoryAction* para interactuar con la interfaz de simulación de Gazebo, que hace de intermediario entre la generación de trayectorias y la simulación, así como también se comunica con el hardware del robot real. La interfaz robótica publica en el tópico /myrobot/joint_states las posiciones de cada una de las articulaciones, resultado de la planificación. Este mismo tópico es el encargado, a su vez, de enviar el *feedback* para la planificación de trayectorias con *Moveit!*.

El sistema encargado del accionamiento de los motores consta de dos nodos. El nodo generaPWM genera los comandos asociados con la posición de cada motor, subscribiéndose al tópico /myrobot/joint_states y publicando el vector en /command con los valores calculados. El mensaje command de tipo Int32MultiArray es el que utilizará el programa del Arduino para el control de los motores controlando el driver por I2C.

6.1.- SIMULACIÓN DEL ROBOT EN GAZEBO

Para llevar a cabo la simulación del robot se han diseñado primero las piezas, utilizando la herramienta AutoCAD. Posteriormente, con el formato URDF definido, se ha creado la disposición de nuestro robot, la cual ha ayudado a visualizar sus movimientos en simulación y comprender la forma de funcionar del robot dentro de ROS.

Algunas de las piezas, al formar parte de la cinemática cerrada, han sido obviadas en la simulación, siendo sustituidos sus efectos de forma programática en el nodo generaPWM. Esto se debe a que el URDF no permite la representación de cadenas cinemáticas cerradas como la que tiene el robot utilizado. Se planifica, por lo tanto, para un robot que no cuenta con el mecanismo cerrado. Posteriormente, el nodo generaPWM, se encargará de corregir dichas posiciones según la relación de ángulos entre el robot utilizado y un robot sin mecanismos paralelogramos. Esta relación se detalla en la cinemática del robot (Ver Apartado 3.1).

6.1.1.- PIEZAS DIBUJADAS EN AUTOCAD

Se han diseñado todas las piezas que conforman el modelo del robot en AutoCAD 3D. Desde la Figura 6.2 hasta la Figura 6.9 se pueden ver los diseños de las piezas junto con el resultado de la construcción URDF de nuestro robot (Figura 6.9). Como se puede apreciar en dicha Figura 6.9, la descripción URDF no tiene representados los *links* de las Figuras 6.5 y 6.6 por la imposibilidad de representación de cadenas cinemáticas cerradas.

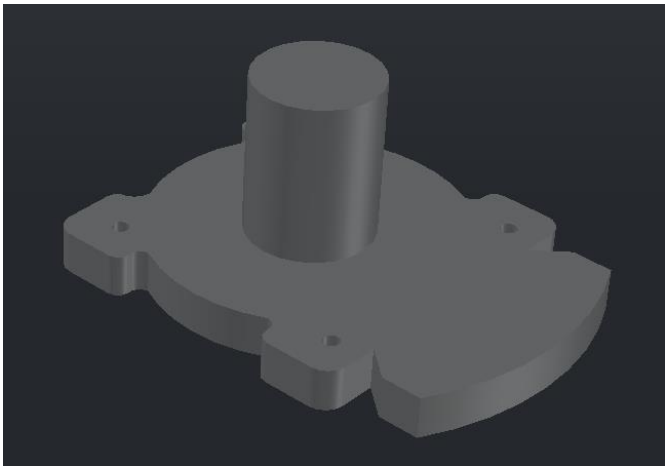


Figura 6.2.- Base del robot.

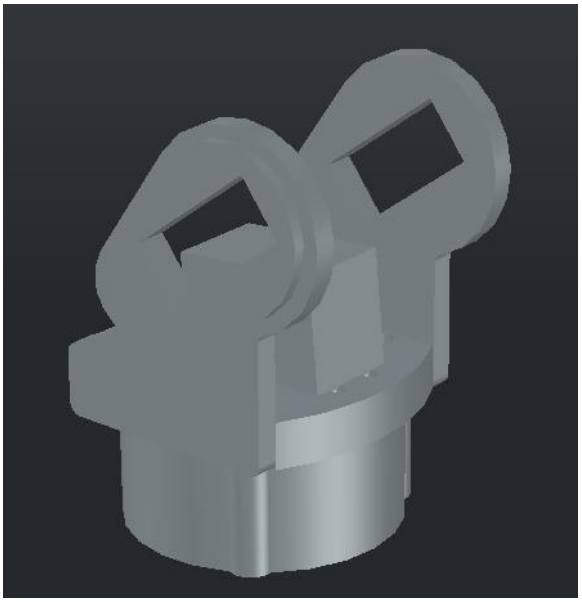


Figura 6.3.- Pieza 1 del robot.

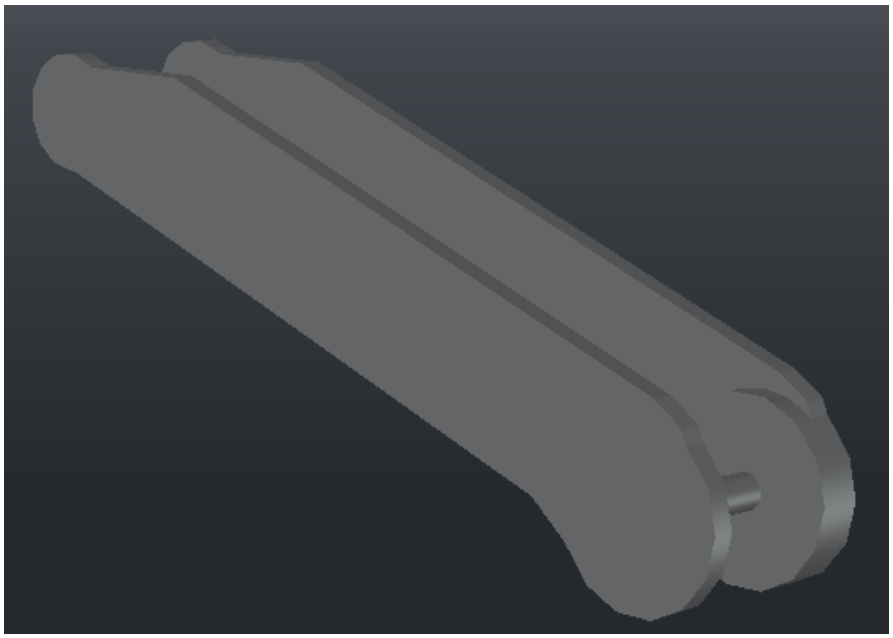


Figura 6.4.- Pieza 2 del robot.

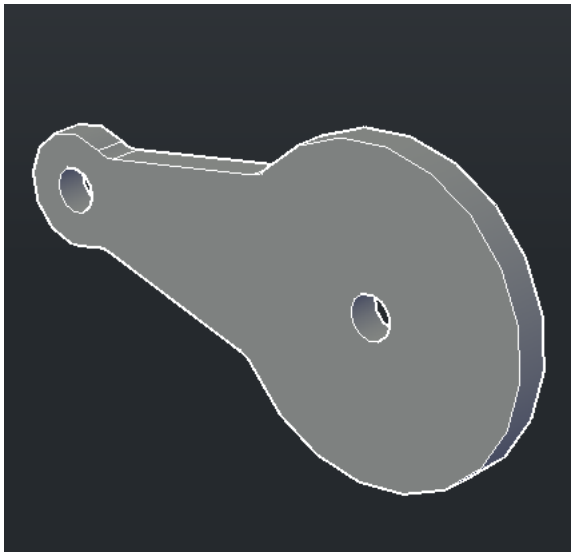


Figura 6.5.- Pieza 3, perteneciente al bucle cinemático cerrado.

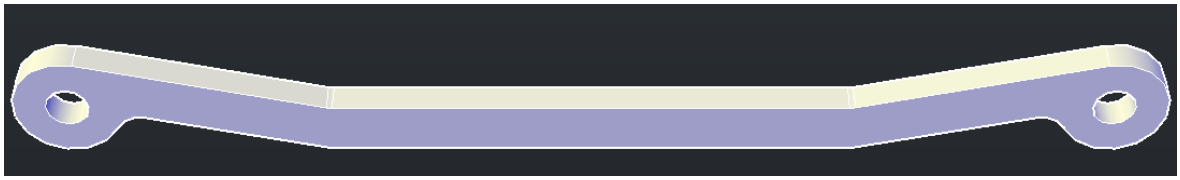


Figura 6.6.- Pieza 4, perteneciente al bucle cinemático cerrado.

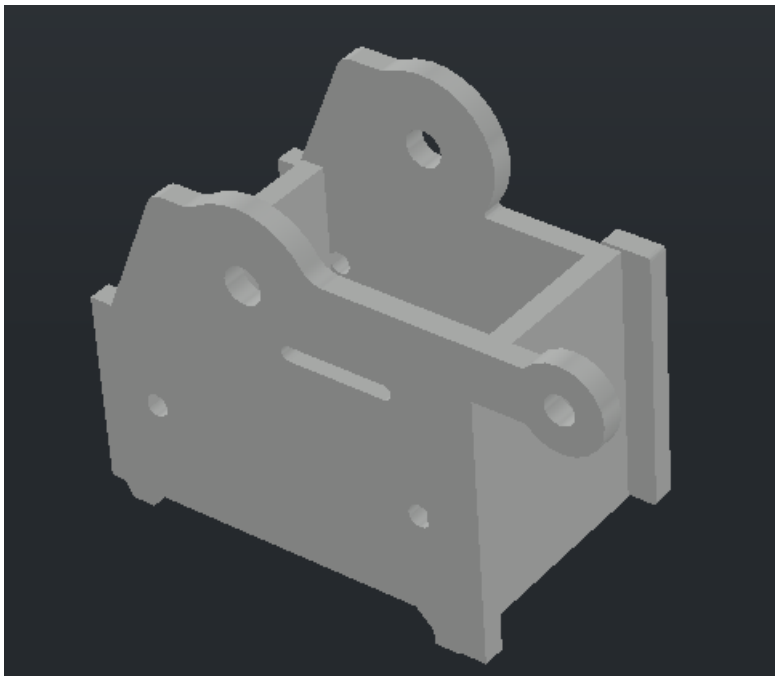


Figura 6.7.- Pieza 5 del robot.

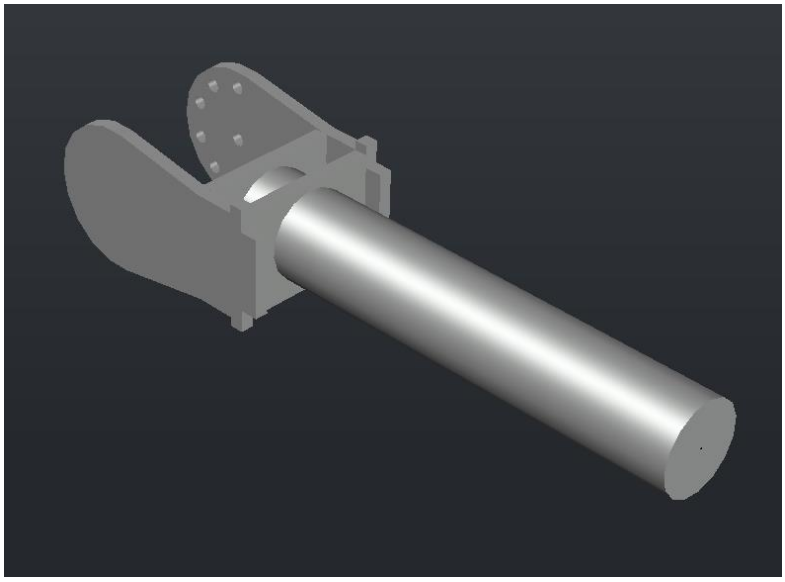


Figura 6.8.- Pieza 6 del robot.

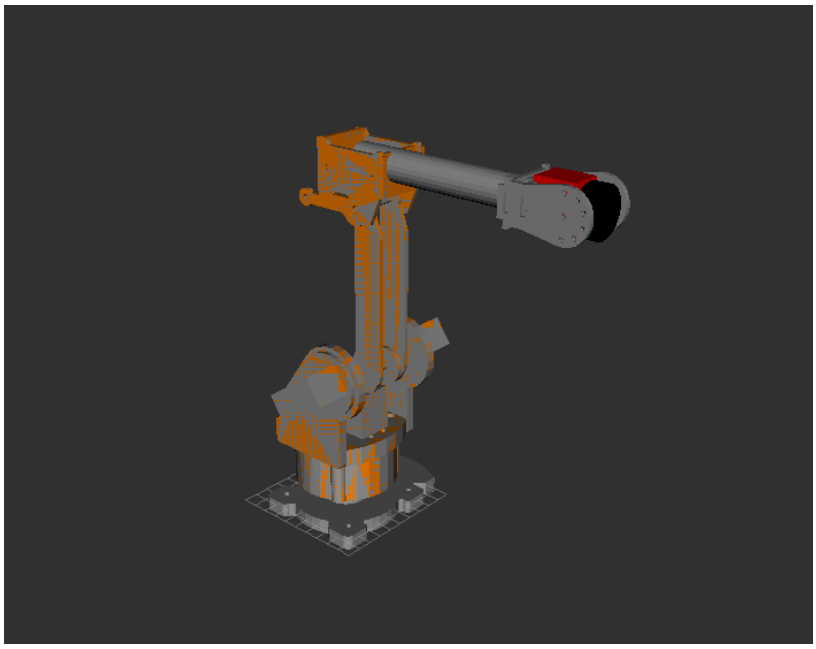


Figura 6.9.- Representación del robot Sainsmart en RVIZ.

6.1.2.- DESCRIPCIÓN DE LOS NODOS DE SIMULACIÓN

Los nodos encargados de la simulación y la planificación de trayectorias, al estar basados en Moveit! y en Gazebo, tienen una carga computacional grande que la Raspberry no es capaz de ejecutar por falta de memoria. Por esa razón, estos nodos se lanzan desde el ordenador, que es el encargado de correr el ROS Master. Para poder poner esto en práctica se modifica el *bashrc* de la Raspberry para que el host sea su propia IP pero trabaje con el

ROS Master en la dirección IP del ordenador. De esta manera se podrán correr nodos en un mismo ROS Master, siempre y cuando compartan red local.

Todos los nodos encargados de la simulación y planificación de trayectorias se lanzan a la vez haciendo uso de un archivo *launch*. Mediante la acción del move_group se crean las trayectorias y se comunica con los controladores de Gazebo para el posicionamiento de los actuadores simulados [23]. La simulación funciona tal y como lo hace el robot real con la interfaz hardware de Gazebo. El archivo *sm_bringup_moveit.launch* es el encargado de lanzar todos estos nodos , así como todos aquellos archivos tipo *launch* encargados de la simulación y generación de trayectorias. Se describen a continuación los archivos *launch* lanzados y aquellas modificaciones y argumentos utilizados en cada caso:

- Se lanza el archivo myrobot_world.launch. Este archivo es el encargado de configurar el entorno de simulación, el cual constará del mundo predefinido de Gazebo empty_world. También lanzará el nodo spawn_robot, que utiliza el modelo URDF como argumento para representar el modelo robótico en Gazebo.

- Se lanza el archivo sm_gazebo_states.launch. Este archivo fue generado por el Moveit setup assistant. Por un lado, carga la información en el param server de los estados de las articulaciones del archivo de configuración joints.yaml y, por otro, carga los controladores de posición en cada una de las articulaciones. El robot_state_publisher se configura para que publique la información de los estados de las articulaciones en el tópico /myrobot/joint_states.

- Se lanza el archivo sm_trajectory_controller.launch. Este archivo utiliza la configuración de los controladores de trayectoria para iniciarlos en el controller manager. Así se puede interactuar con la simulación generando trayectorias. El archivo de configuración utilizado es *trajectory_control.yaml*. Este archivo se escribió de forma que los controladores se encuentran bajo el nombre de nuestro robot *myrobot* y de nuestro grupo de control *Brazo*. De esta forma definimos el conjunto de

controladores de posición (en el lanzamiento de *gazebo_states.launch)* encargados de la generación de trayectorias para su utilización por parte de *Moveit!*.

- Se puede incluir el joint_state_publisher para modificar el estado de las articulaciones en RVIZ teniendo el GUI de joint_state_publisher activado. En nuestro caso, el valor de GUI lo tenemos desactivado, su valor es False.

En las Figuras 6.10 y 6.11 podemos ver los resultados de lanzar nuestra simulación con Moveit! y la generación de trayectoria basada en la acción follow_joint_trayectory, destinada a interactuar con la interfaz de Gazebo. Esto permite generar trayectorias en nuestra simulación, las cuales se pueden probar en la propia herramienta MotionPlanning de RViz, determinando el estado inicial y final del movimiento que se quiere planificar. Esta generación de trayectoria en RViz se puede observar en el video adjuntado en el Anexo I de este trabajo "PruebaSimulación.mp4".

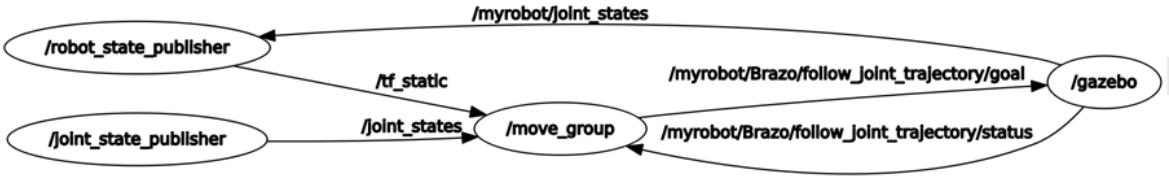


Figura 6.10.- Rqt_graph de la comunicación entre los nodos lanzados para la simulación.

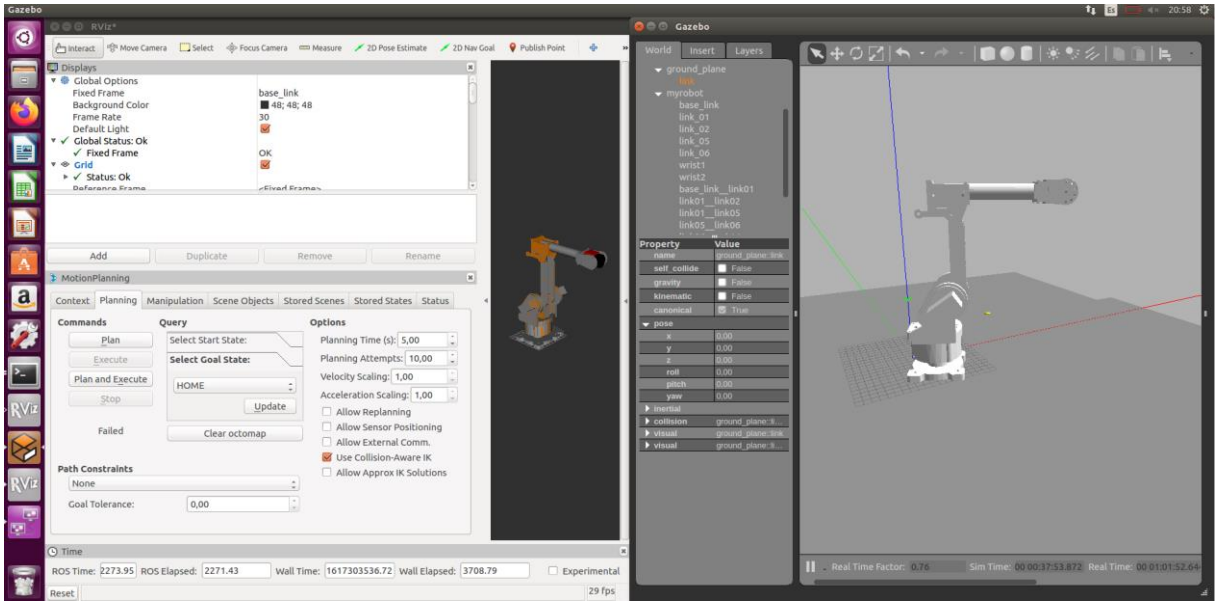


Figura 6.11.- Resultado del lanzamiento de los nodos para la simulación del brazo robótico.

6.2.- COMUNICACIÓN CON EL ROBOT REAL

Los nodos encargados de realizar el accionamiento de los motores son generaPWM y el nodo serial_node de rosserial. Ambos nodos están comunicados mediante el tópico /command, que contendrá los valores asociados al posicionamiento de los motores. Estos valores, publicados por el nodo generaPWM, son recogidos por el programa del Arduino. Este programa los interpreta internamente (haciendo uso de la librería del driver) como el PWM que debe aplicar, utilizando el driver (por medio de una conexión I2C) y situando los motores del robot. El nodo generaPWM está comunicado directamente con la interfaz hardware de gazebo_ros_control, que le ofrece la información de los estados de las articulaciones, publicándola en el tópico /myrobot/joint_states. Esta relación de nodos se puede observar en la Figura 6.12.

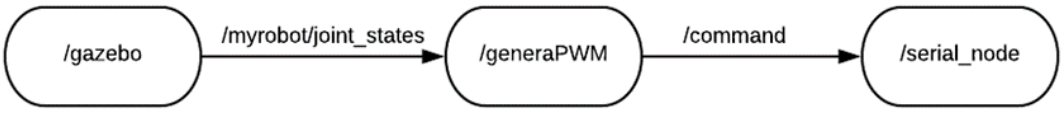


Figura 6.12.- Esquema de la comunicación de los nodos accionadores.

A la hora de posicionar nuestro robot en la realidad, debemos controlar nuestro driver para que genere el PWM adecuado para el posicionamiento. Se ha implementado el nodo generaPWM que, al lanzarlo, actúa de intermediario, convirtiendo las posiciones de trayectoria generadas (leídas del tópico /myrobot/joint_states) en el comando que utiliza la librería del driver para situar los motores. Esto se realiza teniendo en cuenta las limitaciones físicas de la cinemática cerrada del robot, aplicándolas para que desarrolle el movimiento ideado por la acción JointTrajectoryAction y posicionando los motores en la posición calculada para cada motor.

Para la programación del Arduino se utilizó el propio Arduino IDE. Dentro del programa del Arduino se utilizó la librería *Adafruit_PWMServoDriver*, destinada a la placa driver utilizada en este proyecto. Esta librería está desarrollada por Adafruit y se puede encontrar entre las librerías oficiales de Arduino [41]. Para lanzar Arduino como un nodo más del sistema robótico se lanza el nodo serial_node de Python utilizando, como argumento, el puerto al que está conectado el dispositivo (*/dev/ttyACMX*). El nodo rosserial de Arduino recoge los comandos ya ajustados para fijar la posición deseada de los motores. Estos comandos se envían en forma de números enteros positivos, que se recogen como elementos del mensaje *command* de tipo vector Int32MultiArray. La posición 0 de este vector es el valor asociado al motor 1 de nuestro robot, la posición 1 es el valor asociado a nuestro motor 2 y así sucesivamente. Estas posiciones se corresponden con los conectores del 0 al 5 del driver PCA9685, conectados cada uno a su respectivo servomotor.

6.2.1.- CALIBRACIÓN DE LOS MOTORES

Como no se conocía el comportamiento del robot, ni cómo se debía calcular el mensaje publicado por generaPWM, se realizó una calibración, que está aplicada a la conversión realizada por el nodo. En dicha calibración se han hallado los límites mínimos, máximos y cero, así como sus respectivos valores en el mensaje /command. Esto nos permite calcular la regresión lineal para cada motor en función de sus límites, que el nodo generaPWM utiliza para la creación del mensaje /command.

Para poder conocer estos límites, así como para poder controlar la posición del motor, tenemos que ajustar los comandos a la posición asociada a cada uno. Para ello se realizó una calibración en base a los valores introducidos y las mediciones de los ángulos en el robot

real. Este proceso se llevó a cabo publicando por terminal, con el uso del comando *rostopic pub* en el tópico *command*, observando así la posición que tomaba el robot con cada valor publicado. Luego se procedió a la búsqueda del valor del comando máximo, del mínimo y del asociado al ángulo cero de nuestro robot. Todos los ángulos se tomaron con respecto a la posición inicial de nuestro robot en el URDF, que es la misma que la configuración asociada a nuestra descripción DH. A continuación, se detallan esos límites junto con la fórmula utilizada para el cálculo.

Articulación	Valor comando PWM mínimo	Valor comando PWM máximo	Valor comando PWM cero
1	100	500	280
2	500	100	300
3	100	500	160
4	100	500	280
5	100	490	430
6	100	500	250

Tabla 6.1.- Tabla de PWMs asociados a cada articulación resultado de la calibración.

Articulación	Ángulo mínimo (º)	Ángulo máximo (º)	Ángulo cero (º)
1	-90	90	0
2	-81	72	0
3	-5	150	0
4	-90	90	0
5	-120	30	0
6	-90	90	0

Tabla 6.2.- Ángulos de las articulaciones asociados a los valores de los comandos PWM mínimo, máximo y cero.

$$PWMi = (ANGULO_i - ANGULO_{CEROi}) \times \frac{(MAXPWM_i - CEROPWM_i)}{(ANGULO_{MAXi} - ANGULO_{CEROi})} + CEROPWM_i \qquad (6.1)$$

La Ecuación 6.1 nos muestra la forma en la que se calculan las rectas de regresión que calibran los motores en base al valor asociado a cada posición, publicado en el tópico /command.

6.3.- NODOS ENCARGADOS DE LA PERCEPCIÓN

En el Capítulo 5, dedicado a la Percepción para la detección de los círculos, se ha explicado el algoritmo que se aplica para detectar los centros de los círculos en el plano de la mesa, calculando así su posición.

El raspicam_node se configura para que publique la imagen en formato 1280x960 y con 30 fps (frames per second) en el tópico /raspicam_node/image. Para que la imagen se publique adecuadamente se debe poner el valor enable_raw a True. El nodo automáticamente aplica la corrección de las distorsiones halladas con la calibración (Ver Capítulo 5). Después se lanzará el nodo detector_circulos, que utiliza como entrada las imágenes publicadas por el raspicam_node y les aplica la percepción, publicando en el tópico /circulos/coordenadas las posiciones de los objetos detectados, estructurados en un mensaje de tipo Vectorpos. Por otro lado, se publica una imagen en /circulos/imagen_deteccion, que mostrará la salida con los contornos de los círculos, BoundingBoxes y posiciones de los círculos detectados, dibujados en las imágenes a las que está suscrito en cada momento. Esto permite comprobar que la detección se está produciendo de forma correcta. La salida de /circulos/imagen_detección es la asociada con la salida de nuestro programa de percepción, el cual ya se ha visto en las Figuras ejemplo del Capítulo 5. La relación entre los nodos está representada en la Figura 6.13.

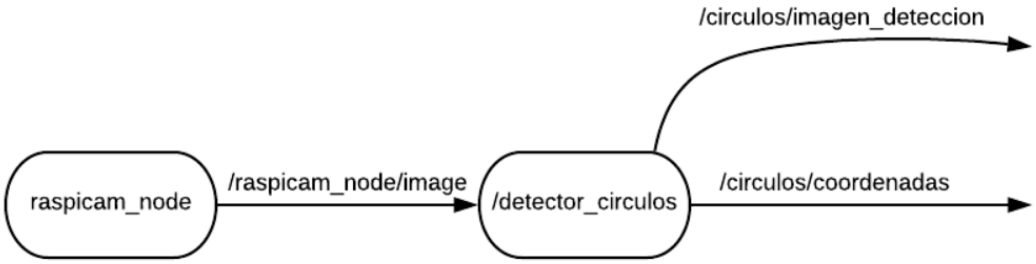


Figura 6.13.- Representación de los nodos y tópicos asociados a cada nodo de percepción.

La detección se realizará con una frecuencia de 50 Hz. En el caso de que no se detecte ningún círculo, el valor de todos los vectores asociados con las coordenadas de los círculos (los vectores X Y y Z del mensaje Vectorpos) estarán vacíos. Las variables enteras n_rojos y n_verdes del mismo mensaje se mantendrán a 0. De esta forma, el nodo que se subscriba

al tópico /circulos/coordenadas conocerá la cantidad de cículos rojos y verdes que hay en la mesa antes de acceder a las coordenadas. Las coordenadas de los círculos están ordenadas de la siguiente forma: las primeras n_rojos posiciones de los tres vectores, se corresponden a las coordenadas XYZ de los círculos rojos; las siguientes n_verdes posiciones, se corresponden con las coordenadas de los verdes.

6.4.- DEMOSTRACIÓN DEL SISTEMA ROBÓTICO

Se ha implementado en Python una clase que calcula la cinemática inversa del robot. Esta cinemática es necesaria si se quiere implementar la cinemática particular de nuestro robot de una forma teórica y didáctica. La cinemática será utilizada por el nodo publicaIk, que utiliza la librería de Moveit! MoveitCommander para comunicarse directamente con el move_group, enviando las posiciones articulares calculadas, generando y ejecutando acciones de trayectoria. Todo ello partiendo de las posiciones de los círculos publicadas en el tópico /circulos/coordenadas. El robot, y su simulación, irán recorriendo dichas posiciones y ejecutando una labor de *pick & place,* como se puede visualizar en el video "demostración.mp4" (aportado en el Anexo I de este trabajo).

El nodo publicaIk hace dos funciones claves en el funcionamiento de nuestra simulación. Por un lado, se comunica con el nodo de percepción que, como ya hemos expuesto en el Capítulo 5, es el encargado de localizar la posición de los círculos de colores, publicando dicha posición en el tópico /círculos/coordenadas. PublicaIk calcula la cinemática inversa y envía las posiciones de las articulaciones al move_group para el posicionamiento correcto de los controladores.

La demostración utilizada consta de una sencilla secuencia. Esta secuencia empezará desde una posición inicial, la cual está pensada para que el brazo no oculte los posibles círculos presentes en la mesa. En este punto lee la imagen, que será el mensaje en ese determinado momento del tópico /raspicam_node/image. Con esa imagen se realiza un *callback* que implementa la secuencia *pick & place*. Esta secuencia comenzará por la detección de los círculos en la imagen, continuará convirtiendo dichas posiciones en las posiciones de las articulaciones en cada pose y finaliza generando las trayectorias para

alcanzar todos los puntos del *pick & place*. La secuencia de demostración contará con las siguientes fases

1. Posición inicial.

2. Posición del objeto.

3. Punto intermedio: se ha seleccionado la posición inicial.

4. Posición del depósito correspondiente al objeto recogido: a la izquierda del brazo el depósito rojo y a la derecha el verde.

Una vez que se lleven todos los círculos a sus depósitos, el brazo se quedará en espera hasta que vuelvan a aparecer nuevos círculos dentro del rango de trabajo del robot.

7. Resultados y discusión

El resultado principal del trabajo es el diseño de la arquitectura y programación del sistema robótico. Debido a las circunstancias provocadas por la pandemia del COVID19 se ha situado al Robot Sainsmart, utilizado en la planta de la Industria 4.0 de la EPI, ante la tarea de clasificar círculos de colores, tarea similar al *pick & place* que realizará en la misma. Se espera que el diseño desarrollado pueda servir de base para futuras investigaciones e implementaciones del sistema robótico dentro de la planta. El desarrollo completo de la programación implementada para la simulación de este trabajo se encuentra en el Anexo I, junto con un video del robot realizando su tarea.

Para obtener una visión más amplia de los resultados, se ha sometido al sistema robótico a diferentes pruebas, con el objetivo de estudiar más detenidamente su funcionamiento. Esto se ha realizado para conocer las limitaciones y particularidades de nuestro diseño. Este análisis también nos ayudará a sostener argumentalmente los desarrollos que se pueden llevar a cabo en futuras implementaciones del proyecto.

7.1.- RESULTADOS DE LA PERCEPCIÓN

En el caso de la percepción se han llevado a cabo dos pruebas básicas que persiguen comprobar la calidad y versatilidad de nuestro programa.

La primera prueba está relacionada con la verificación de las distancias medidas por el nodo de percepción, así como el cálculo de sus errores asociados. Esta prueba demuestra que, desde la posición que hemos utilizado para el experimento, las distancias nos dan resultados bastante aproximados, nunca superando los 8 mm de error absoluto. En la Tabla 7.1 se representan las distancias tomadas para la comprobación de las medidas. Para realizar las pruebas, se ha utilizado el mismo tablero de ajedrez con el que se realizó la calibración. En primer lugar, se posicionó un objeto en la posición X=0, variando su Y de 5 cm en 5 cm hasta 20 cm. Posteriormente, se dejó el eje Y fijo en 12.5 y se varió X de 5 en 5 cm hasta los 15 cm. Una vez terminada esta comprobación, se seleccionaron 10 medidas aleatorias, que

no coincidieran con ninguno de los puntos del tablero empleados para la transformación. Los resultados obtenidos se pueden ver en la Tabla 7.1 junto con los errores asociados.

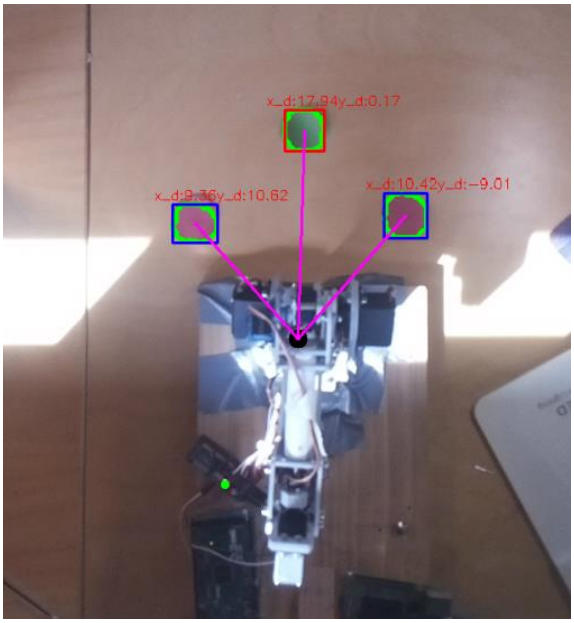


Figura 7.1.- Salida del nodo de visión en el tópico /circulos/imagen_deteccion.

Xreal	Yreal	Xobtenida	Yobtenida	Error X	Error Y	Error
0	0	0,12	0,3	0,12	0,3	0,32310989
0	5	0,2	4,4	0,2	-0,6	0,63245553
0	10	0,26	9,64	0,26	-0,36	0,44407207
0	15	0,46	14,45	0,46	-0,55	0,71700767
0	20	0,32	19,68	0,32	-0,32	0,45254834
0	12,5	0,08	12,3	0,08	-0,2	0,21540659
5	12,5	5,12	12,1	0,12	-0,4	0,41761226
10	12,5	9,95	12,06	-0,05	-0,44	0,4428318
15	12,5	14,72	12,3	-0,28	-0,2	0,34409301
24,5	27	23,92	27,08	-0,58	0,08	0,58549125
24,5	12,5	24,32	12	-0,18	-0,5	0,53141321
24,5	0	24,1	-0,12	-0,4	-0,12	0,41761226
24	-5	23,74	-5,23	-0,26	-0,23	0,3471311
18	0	17,94	0,17	-0,06	0,17	0,18027756
10	-9	10,42	-9,01	0,42	-0,01	0,42011903
9	10	9,36	10,62	0,36	0,62	0,71693793
13	14	13,31	14,22	0,31	0,22	0,38013156
17	12	17,11	12,29	0,11	0,29	0,31016125
20	5	20,08	5,12	0,08	0,12	0,14422205

Tabla 7.1.- Tabla de errores siguiendo el método de verificación.

En la Figura 7.2 podemos ver la distribución normal de los errores obtenidos. En ella se puede observar que los errores que aparecen con mayor frecuencia son los que rondan los 3-4 mm. En la Tabla 7.2 se puede apreciar la media de error es de 4,22 mm y una desviación típica de 0,16.

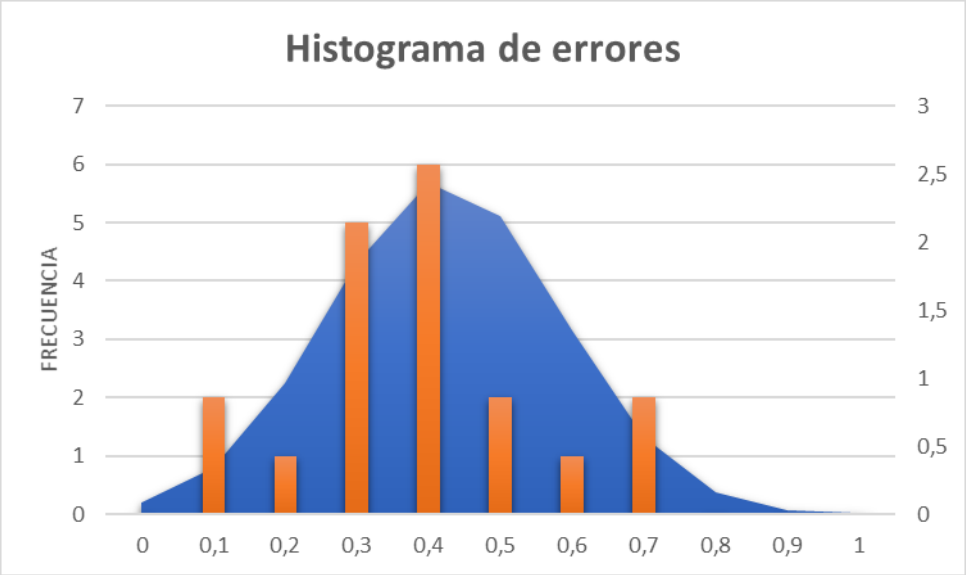


Figura 7.2.- Histograma de errores de la localización de los círculos.

Media	0,42224391
Desviación típica	0,16208562

Tabla 7.2.- Media y desviación típica de los errores obtenidos.

Se ha encontrado como referencia un artículo con una aproximación de la medición de distancias realizadas con homografía. Las medidas que encontramos en él fueron realizadas con una perspectiva más pronunciada y una distancia mayor que en nuestro experimento. En este artículo se obtiene un error medio de aproximadamente 10 mm y un error máximo de aproximadamente 60 mm [42]. En consecuencia, se puede concluir que, pese a que esta detección no es la ideal si se busca una precisión alta, sí que estos errores se pueden considerar aceptables, teniendo en cuenta el sistema de detección empleado. Además, se deberá de tener este tipo de error en cuenta a la hora de seleccionar la garra, adaptando así la manipulación al error y la superficie de las canicas; esto último en el caso de que se quiera continuar utilizando el algoritmo del presente trabajo.

Por otro lado, en una segunda comprobación, se ha experimentado con diferentes posiciones de la cámara, calculando la homografía asociada a dichas colocaciones y observando los resultados de la detección. Estas disposiciones las podemos observar en las Figuras 7.3 y 7.4.

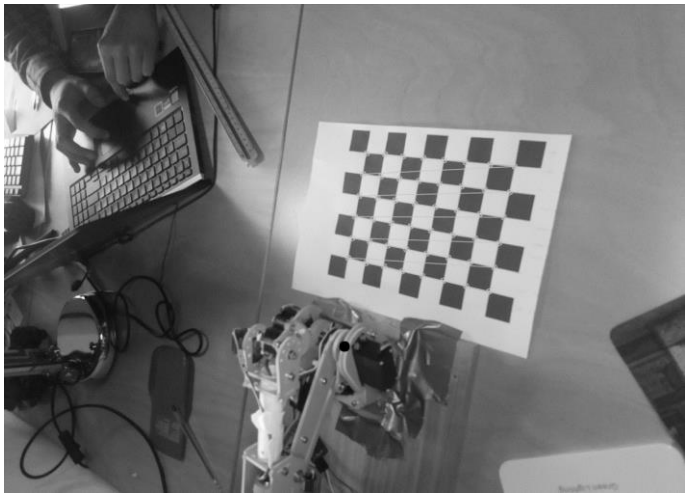


Figura 7.3.- Disposición alternativa 1.

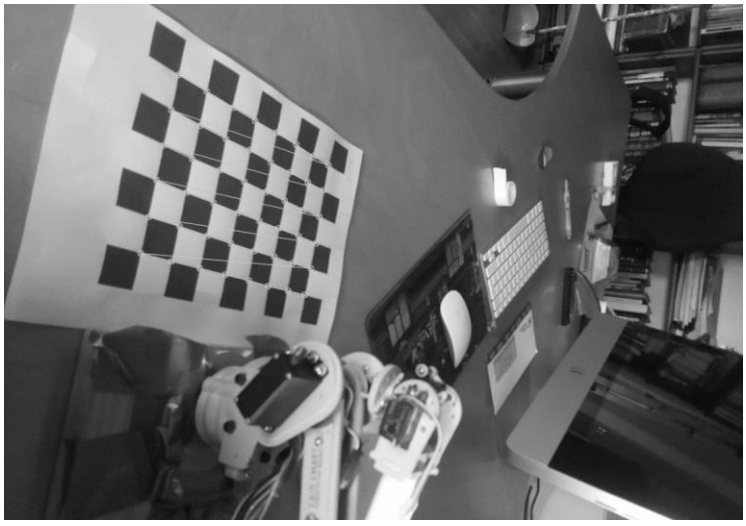


Figura 7.4.- Disposición alternativa 2.

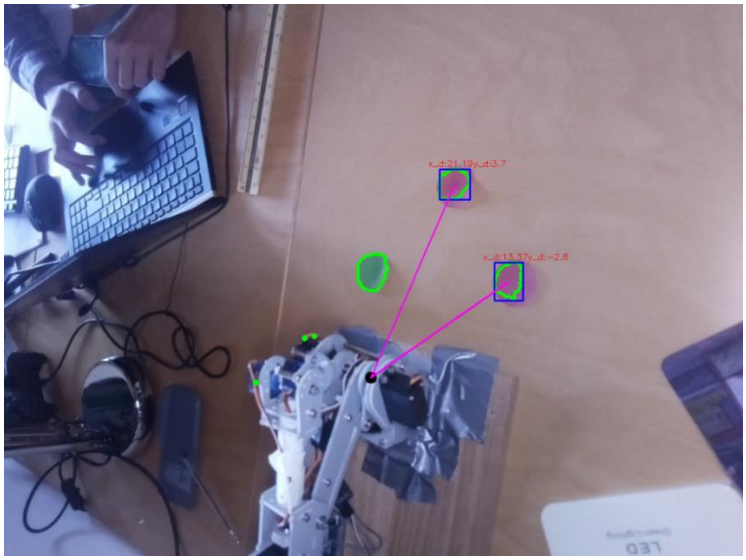


Figura 7.5.- Resultado posición alternativa para nuestro programa de detección.

En ambas disposiciones (alternativa 1 y 2) se puede apreciar el mismo efecto, observable en la Figura 7.5. Existen reflejos que no ayudan a la detección de los círculos, detectando solo una parte de los objetos correctamente y desplazando la posición central de la *BoundingBox*. Esto hace que la detección sea más imprecisa.

Al depender únicamente de los colores de los círculos, nuestro sistema de detección es frágil en cuanto a que dependemos de un nivel de luz estable. Nos favorece, por lo tanto, la inexistencia de reflejos en los círculos. Los reflejos suelen producir color blanco. Este color blanco genera una saturación del color. Esto produce que la saturación del modelo HSV tienda a 0 y que H no sea representativo a la hora de identificar el color. En el caso de que existan reflejos, por lo tanto, se va a localizar de manera incorrecta el centro del contorno, ya que solo se corresponderá con una parte de este, tal y como se puede ver en la Figura 7.5. Por todo ello, la posición de la cámara utilizada a lo largo del trabajo es la más apropiada, puesto que mantiene el color de los círculos (en la imagen obtenida) lo más homogéneo posible, en comparación con las posiciones de las Figuras 7.3 y 7.4. De la misma manera, se deben evitar aquellas zonas de la mesa en las que haya mucha luz localizada, las cuales originan el mismo efecto indeseable.

Se puede observar también cómo, aquellos círculos que se encuentran fuera del rango de distancias en las que trabaja el robot, se enmarcan y se declaran como *OUT OF*

BOUNDARIES. Es necesario que esto funcione apropiadamente, para que no se envíen al pick & place las posiciones inalcanzables.

Por un lado, esto permitirá que el robot no calcule ángulos imposibles, lo cual bloquearía nuestro sistema de control, al detectarlo RVIZ como límite fuera de rango de la articulación en el URDF. Por otro lado, las canicas deben detectarse fuera de rango cuando estén situadas en los depósitos. La detección de los objetos no alcanzables la hace de forma correcta y se pueden observar las dos situaciones descritas en las Figuras 7.6 y 7.7.

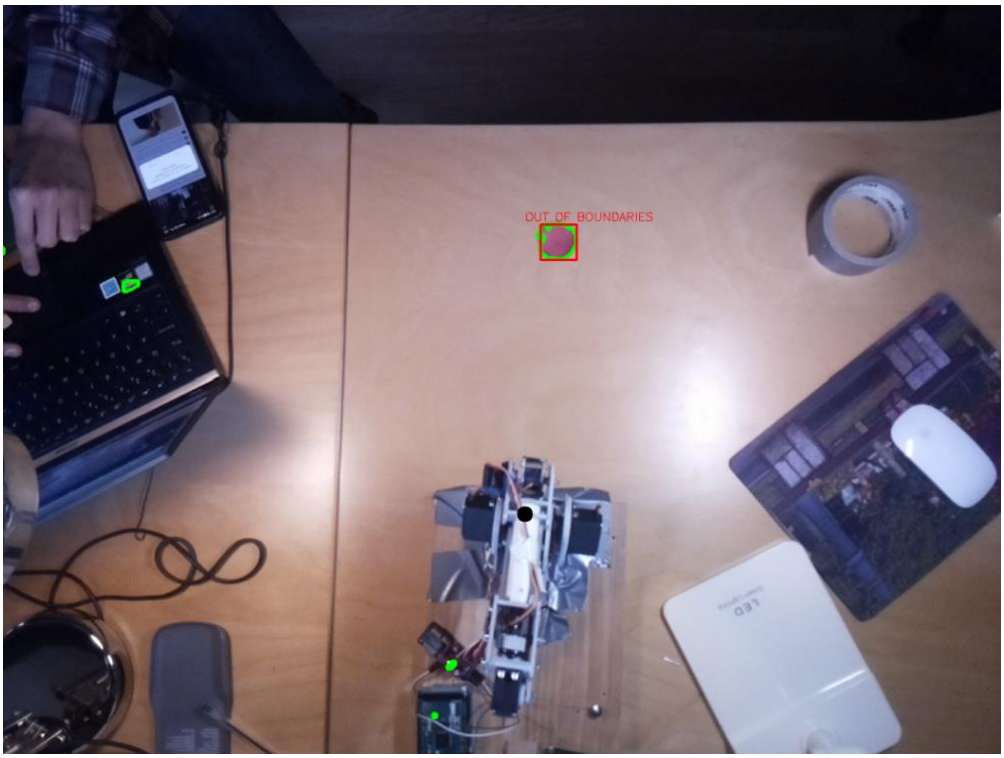


Figura 7.6.- Círculo fuera de rango por distancia.

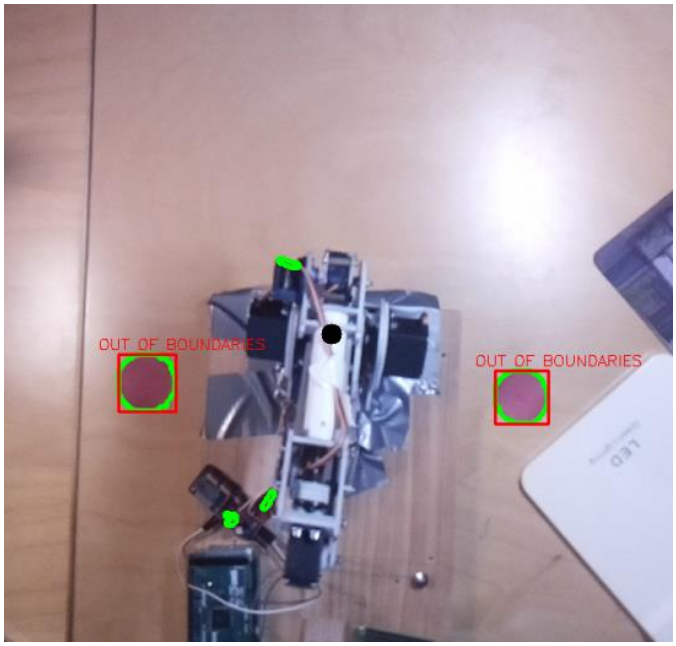


Figura 7.7.- Círculos fuera de rango por localización en depósitos.

Por último, cabe señalar que se ha utilizado el modo del raspicam_node de captura de video a 30fps. Este modo parece adecuado en cuanto a que las bolas, antes de ser recogidas, han de estar paradas. Esto hace que no se necesite una excesiva suavidad en las transiciones del video para un buen resultado de nuestra aplicación.

7.2.- POSICIONAMIENTO DE LOS MOTORES

El posicionamiento de los motores se adecúa en cuanto a nuestra calibración se refiere. No obstante, el salto de un PWM a otro no se produce de una forma suave en determinadas ocasiones. Es decir, algunos movimientos se producen a golpes. Esto se debe a que los servomotores del robot no permiten hacer un control de velocidad, solo se puede controlar la posición. Como consecuencia, no pueden sincronizarse los movimientos de varias articulaciones distintas a la perfección.

También existe un cierto error en el posicionamiento del brazo, el cual se asocia con la calibración realizada y los errores presentes en la percepción. En cuanto a la calibración de la posición de los motores, lograr el control exacto mediante un PWM en servos de posición como los utilizados, teniendo en cuenta los *offsets* y situando con eficacia sus

posiciones, es claramente difícil. Además, debido a la ausencia de sensores de posición angulares en los motores, no podemos comprobar el correcto posicionamiento más que de forma visual o midiendo con herramientas de baja precisión. A pesar de todo esto, los resultados obtenidos se consideran lo suficientemente buenos y coherentes con respecto a las posiciones enviadas.

También se ha sometido al sistema, con el objetivo de visualizar los resultados con la presencia de una garra, a una carga simulada por un saquito en el que hemos ido variando su peso. Esto se ha realizado con el objetivo de simular el peso del efector final, conociendo así el rango de pesos de la garra con el que el robot utilizado podría trabajar. Esta carga ha sido variada buscando el peso a partir del cual la estructura del robot le cueste desplazarse. El resultado de estas pruebas mostró que, utilizando pesos entre 50 y 200 gramos, el robot funciona correctamente. Cerca de los 300 gramos las articulaciones comienzan a tener problemas para soportar el peso. A partir de 500 gramos en adelante, el robot no es capaz de levantar la carga, por lo que no se recomienda funcionar con pesos de 300 gramos en adelante, ya que podríamos estropear alguno de los servos. Estas consideraciones se deberán tener en cuenta a la hora de escoger el efector final del robot.

Se puede apreciar en el video “Simulación.mp4” encontrado en el Anexo I cómo, en la simulación, las poses adoptadas por el robot son correctas, además de los ángulos calculados. Esta comprobación se ha realizado con los resultados adoptados en la cinemática inversa mediante la aplicación de la cinemática directa en scripts de Matlab (Ver Anexo II) sobre los ángulos adoptados por la simulación en Gazebo (tomando los resultados publicados en el mensaje /myrobot/joint_states). Los resultados obtenidos al calcular la cinemática directa sobre dichas posiciones angulares daban la posición buscada por el robot, deshaciendo previamente la vinculación de los ángulos $\theta 2$ y $\theta 3$ producida por la presencia de la estructura cerrada del robot paletizador.

Por último, se ha probado a realizar el control con un Arduino UNO. No obstante, por el pequeño tamaño de su memoria, el Arduino no permitía el control del robot, únicamente el de las articulaciones de forma aislada, lo cual no nos sirve. El Arduino MEGA, un modelo con más memoria, le permite al sistema robótico diseñado funcionar adecuadamente.

7.3.- RESULTADOS DE LA DEMOSTRACIÓN

La demostración se ha llevado a cabo y se muestra en el video "demostración.mp4" que se encuentra en el Anexo I. Dentro de nuestra demostración se formula un ejemplo de *pick & place* simulado, debido a que no existe la presencia de una garra como tal que realice la manipulación. Esto supuso que el movimiento lo hace un humano que interactúa directamente con el robot, moviendo los círculos en función de sus indicaciones. Las indicaciones las realiza de forma correcta situando cada círculo según su color en su conveniente depósito sin ningún tipo de fallo (primero conduce los rojos hacia la izquierda y, posteriormente, los verdes a la derecha). Una vez terminado el proceso se sitúa en una posición de espera, la cual está pensada para que no tape el plano de trabajo, esperando a que un nuevo círculo se sitúe dentro del rango para retomar su acción sin problemas.

Algunas limitaciones son evidentes. Puede que una canica que tenga algo de movimiento no sea correctamente recogida en el momento en el que el brazo dé una primera vuelta en su secuencia. Esto se debe al hecho de que variará su posición con respecto a la imagen que utilizará la función callback de publicaIk, tomada en el primer momento antes de la acción.

No obstante, esto no debería suponer un problema en el caso de las canicas, debido a que, de la forma en la que se ha diseñado el programa, en caso de que una canica no fuese recogida en un primer intento (y permaneciese dentro de los límites de recogida del robot), en el siguiente *callback* volvería a localizarla, recogiendo todas aquellas canicas que hayan permanecido en el área alcanzable por el robot. Por otro lado, también relacionado con la demostración, el robot utilizado es un robot de reducidas dimensiones, cuyo rango de recogida es pequeño. Esto lo limita mucho, puesto que las canicas podrían salirse del área de trabajo con facilidad en el caso de no haber un recinto bien delimitado.

7.4.- COMUNICACIÓN RASPBERRY-ORDENADOR

Debido a la gran carga computacional que supone la utilización del move_group, la Raspberry de la que se dispone en la planta no nos permitía la utilización de la interfaz junto con el resto del programa, dando un error de memoria que no se ha podido solucionar

directamente. Por ello se abordó dicho problema con una distribución computacional entre el ordenador y la Raspberry, de forma que ambos estén conectados a una misma red local.

A la hora de tener el sistema funcionando, se considera que esta implementación ofrece bastantes ventajas pese a no ser tan compacto; al estar conectado a la red inalámbrica, no necesita estar ocupando sitio en la mesa de trabajo del robot, pudiendo interferir con él u ocupar espacio de forma indeseada. El ordenador solo corre con la parte computacional grande de nuestro sistema robótico, las conexiones físicas recaerán sobre la Raspberry.

8. Conclusiones y trabajo futuro

8.1.- CONCLUSIONES

El sistema operativo robótico ROS nos ofrece una interesante herramienta a la hora de implementar el control de un brazo robótico. Robotic Operating System, debido a su modularidad y versatilidad, nos permite no solo trabajar con diferentes nodos dentro de nuestro sistema robótico, sino que también nos permitiría comunicarnos con otras implementaciones de sistemas robóticos simultáneamente e integrar sensores para un control más complejo. Esto hace que se amolde a la perfección al funcionamiento y filosofía demandado por la Industria 4.0.

Nuestro sistema robótico funciona, con eficacia, dentro de las limitaciones relatadas en los resultados, constituyendo una buena arquitectura base para la utilización de casi cualquier brazo robótico. Además, la construcción de nuevos proyectos sobre el expuesto en este trabajo sería relativamente fácil: implementando nuevos programas de pick and place, modificando el programa de visión para integrar nuevas funcionalidades u originando *plugins* específicos para la cinemática inversa.

La cinemática responde correctamente basándonos en los tests realizados de forma manual en Matlab, así como en la prueba del nodo de cinemática en funcionamiento, el cual da buenos resultados. Estos resultados también se pueden observar de forma visual tanto en RVIZ, como en simulación y en el robot real.

El accionamiento es correcto obteniendo resultados de posición aproximados a los buscados y al tipo de calibración llevada a cabo, a pesar de las limitaciones y la ausencia de *feedback* por parte de unos sensores de posicionamiento.

8.2.- TRABAJO FUTURO

A lo largo del proyecto, y en cuanto vas avanzando en el conocimiento, aparecen inquietudes acerca de las posibles aplicaciones que podría tener el brazo robótico con ciertas implementaciones más avanzadas de ROS. Esto da lugar a algunas ideas sobre las mejoras del hardware y el software que se podrían llevar a cabo.

Una implementación interesante para la detección de los objetos sería la de hacer uso de una cámara 3D, como la kinetic. Esta cámara, obteniendo los datos del entorno de trabajo, se comunicaría con el move_group como un sensor de cámara, trasladando los objetos detectados en forma de pointcloud o nube de puntos. De esta nube de puntos podremos extraer el plano de la mesa junto con las canicas, eliminando todos aquellos objetos que no son de nuestro interés. Aunque sea una implementación bastante más ambiciosa para la simpleza de la labor, es muy interesante y conllevaría un pequeño estudio de *machine learning* para establecer la relación de las distancias (mediante un entrenamiento con los Histogramas obtenidos y sus mediciones). Luego, basándonos en las características de los histogramas detectados, habría que definir las características que definirían el material de cada una de las bolas.

Otra implementación posible, más cercana a la estructura utilizada y fácilmente trasladable a nuestro modelo, sería un programa de visión ideado para clasificar las canicas reales en función de sus materiales de una forma más sólida que la expuesta en este trabajo. Dichos materiales tendrán unas características más complejas que el modelo utilizado en este proyecto para simularlas (basado únicamente en la clasificación según sus dos colores).

Estas dos líneas de desarrollo acercarían definitivamente el proyecto a su aplicación directa en el trabajo real requerido por la planta de la Industria 4.0 de la Universidad de Oviedo.

Como mejora o implementación alternativa, al uso de la comunicación TCP-IP presentada como solución a la ejecución del move_group en nuestro robot, se podría cambiar la Raspberry Pi utilizada por un dispositivo más potente, aumentando su memoria. La opción

que parece más plausible sería el uso de una Raspberry pi 4B+ con 8GB de memoria RAM, que pudiera realizar todo el control en un mismo dispositivo (la utilizada solo tiene 4GB).

El robot, de momento, no tiene implementada ninguna garra. En futuras actualizaciones del sistema robótico debería incluirse la garra y actualizar el modelo URDF utilizando la herramienta *Moveit Setup Assistant* de ROS, con el objetivo de integrar el modelo dentro de nuestra simulación. Esta garra se deberá escoger teniendo en cuenta los pesos con los que puede trabajar el robot (ver en Apartado 7.2.6) y el peso de las canicas.

Por último, otra idea interesante sería la de implementar un hardware robótico que tuviera sensores en las propias articulaciones. Esto permitiría devolver algún tipo de feedback, con objeto de incrementar sus funcionalidades y poder programar una interfaz hardware más simple, que no dependiera de la simulación y que se comunicara con el robot real y el move_group. Además, la existencia de sensores nos permitiría obtener unos mejores datos de calibración y, por consiguiente, de posicionamiento del brazo. En esta línea, el brazo robótico Nyrio One ofrece feedback, siendo un modelo muy atractivo para su implementación en la planta [19].

9. Planificación y presupuesto

9.1.- PLANIFICACIÓN.

El trabajo llevado a cabo durante los pasados 9 meses se resume en las tareas que se encontrarán descritas en la Tabla 10.1.

Concepto	Horas de trabajo
Instalación de ROS	**8 horas**
Instalación Máquina Virtual Ubuntu 16.04 en el ordenador	1 hora
Instalación de ROS en la máquina virtual	3 horas
Instalación de ROS en Raspberry Pi	2 horas
Instalación de los paquetes ROS	2 horas
Aprendizaje previo	**83 horas**
Aprendizaje de Linux	3 horas
Aprendizaje de ROS	80 horas
Simulación del robot	**26 horas**
Diseño de las piezas en CAD	15 horas
Creación de paquete de descripción del robot	5 horas
Creación del paquete Moveit!	1 hora
Configuración de la integración Moveit! en Gazebo	5 horas
Sistema de localización de objetos	**20 horas**
Estudio del problema de percepción	10 horas
Estudio de OpenCV Python	4 horas
Creación del paquete de visión	5 horas
Instalación y calibración de la cámara	1 hora
Accionamiento de los motores	**20 horas**
Estudio del problema del accionamiento con rosserial y Arduino	10 horas
Programación del Arduino para integración en ROS	3 horas
Calibración de los servomotores	1,5 horas
Creación del paquete acciona_motor	5 horas
Pruebas del robot con la generación de trayectorias	0,5 horas
Demostración del pick & place	**27 horas**
Estudio de la cinemática del robot	20 horas
Estudio del algoritmo pick & place	0,5 horas
Creación del paquete "nodos_ik"	3 horas
Pruebas del paquete "nodos_ik" en simulación y en la realidad	1,5 hora
Puesta en funcionamiento de la demostración	2 horas
Otras actividades	**96 horas**
Pruebas localización	2 horas
Pruebas posicionamiento del robot	2 horas

Pruebas demostración	2 horas
Redacción de la memoria	80 horas
Grabación y edición de videos	10 horas
TOTAL	280 horas

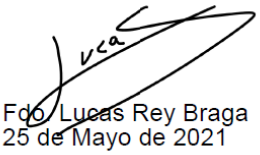
La mano de obra llevada a cabo es de DOSCIENTAS OCHENTA HORAS de trabajo.

9.2.- PRESUPUESTO

Contando con las horas de trabajo solicitadas y con los materiales utilizados en el presente trabajo, se procede a calcular el presupuesto del trabajo desarrollado. Para los equipos se considera una amortización de 3 años.

Concepto	Coste unitario	Medición	Coste
Mano de obra	25€/h	280h	7000 €
Ordenador (Intel i7 8th gen 16 GB RAM,512GB SSD y gráfica GTX de 2GB	31.91€/mes	9 meses	287,19 €
Raspberry Pi 4B+ 4GB RAM	1,7€/mes	9 meses	15,24 €
Sainsmart robot 6dof Desktop	3,11€/mes	9 meses	38 €
Arduino Mega 2560 REV3	0,98€/mes	9 meses	8,82 €
Raspicam v2.0	0,84€/mes	9 meses	7,56 €
Driver PCA9685	0,18€/mes	9 meses	1,62 €
Cables y accesorios	0,84€/mes	9 meses	7,56 €
Pantalla Auxiliar MSI Optix G27C4	5€/mes	9 meses	45 €
Ubuntu Xenial 16.04	0€/h	278h	0 €
ROS Kinetic	0€/h	278h	0 €
AutoCAD 2021	291€/mes	1 mes	291 €
Microsoft Office	5,75€/mes	9 meses	51,75 €
SUBTOTAL			7.800,56 €
Gastos generales (5%)			390.03 €
Beneficio (7%)			546.04 €
IVA (21%)			1.638.12 €
TOTAL + IVA			10.374.72 €

El Importe Total del proyecto incluyendo el IVA es de DIEZ MIL TRESCIENTOS SETENTA Y CUATRO EUROS CON SETENTA Y DOS CÉNTIMOS.



Fdo. Lucas Rey Braga
25 de Mayo de 2021

10. ANEXOS

10.1.- CONTENIDO DEL ANEXO I

Dentro de este anexo encontraremos la carpeta "src", en cuyo interior se encontrarán los paquetes de nuestro *catkin workspace*. Dentro de la carpeta que se llama Sainsmart Robot se podrán encontrar las carpetas que serán nuestros paquetes ROS. Estas carpetas serán las siguientes:

- mr_description: carpeta que contendrá la descripción URDF de nuestro robot dentro de la carpeta urdf con el nombre "myrobot.xacro" junto con sus macros en el archivo "links_joints.xacro". Dentro de la carpeta launch encontraremos los archivos launch necesarios para lanzar nuestra simulación. El archivo launch que tiene el lanzamiento de todos los archivos necesarios para correr la simulación junto con el move_group será el archivo "sm_bringup_moveit.launch". En la carpeta *meshes* se encontrarán los archivos CAD en formato *.dae* y *.stl*. En la carpeta *yaml* se encontrarán los archivos de configuración de nuestra simulación.

- sm_moveit_pkg: carpeta creada por el *Moveit Setup Assistant* que contendrá archivos necesarios para el lanzamiento de la integración Moveit y Gazebo. Gracias a este paquete se podrá generar el SRDF y sus controladores en Gazebo para poder usar la generación de trayectorias en la simulación. Estos archivos se encuentran en launch y los necesarios para esta labor los podemos encontrar también en la carpeta launch de mr_description para posibilitar la simulación.

- vision_trabajo: carpeta que contendrá los programas relacionados con la parte de la percepción del sistema robótico. Dentro de la carpeta scripts se encontrará el nodo encargado de realizar la percepción, denominado nodo_percepcion.py. También se encontrará, dentro de la carpeta *include* de nuestro paquete, la librería empleada para realizar la percepción, que

contendrá las funciones utilizadas por el nodo “VisionTrabajo.py”. Dentro de *include* también se encontrará el script de Python desarrollado para calcular la homografía “Homografia.py” y el programa “filtro_HSV.py”, que calculará la salida para cada rango de umbralizado en HSV. Junto con los dos programas previamente mencionados se encontrarán las imágenes “HomografiaPrueba.png” y “Escenario.png” que se lanzarán de forma predefinida con cada uno de los *scripts* para poder ser probados. Dentro de la carpeta *msg* del paquete encontraremos el mensaje utilizado para la transmisión de las posiciones de los círculos “Vectorpos.msg”.

- acciona_motor: este paquete contendrá en la carpeta scripts el nodo generaPWM programado en el script “nodo_simple.py”.

- nodos_ik: dentro de la carpeta scripts encontraremos el nodo encargado de la demostración dentro del script “publicaIk.py”. Dentro de la misma carpeta, se encontrará el programa de pruebas de la librería MoveitCommander “MoveitCommander.py”. Lanzando dicho script se puede situar nuestro modelo en simulación y en RVIZ en la posición de prueba del script. Dentro de la carpeta *include* se puede encontrar la librería “FuncionesFkIk.py”, que contiene las funciones para calcular la cinemática inversa del robot Sainsmart. Esta librería es utilizada por el nodo “publicaIk.py” para mandar las órdenes en la demostración.

- servo: esta carpeta no es un paquete de ROS como tal, pero contiene el programa que deberá utilizar el Arduino, además de un Readme.md con las instrucciones para correr Arduino en ROS con rosserial.

En el Anexo I también se encontrará la carpeta “videos” en la que se pueden ver los videos del funcionamiento y lanzamiento de nuestros diferentes nodos como muestra de los diferentes desarrollos llevados a cabo en ROS, con comentarios de cada paso. También se encontrarán algunas de las calibraciones llevadas a cabo. Se podrán encontrar los siguientes videos:

– CalibracionCamara.mp4

- ComunicacionOrdenadorRaspberry.mp4
- Demostración.mp4
- NodoPercepcion.mp4
- PruebaSimulacion.mp4

10.2.- CONTENIDO DEL ANEXO II

En la carpeta CAD podremos encontrar todos los archivos relacionados con el diseño de las piezas para su simulación en Gazebo. Dentro de la carpeta CAD, en la carpeta "Archivos", encontraremos los archivos de Autocad y los *.dae* de cada una de las piezas. En la carpeta "Planos" encontraremos los planos de las piezas en PDF también realizados con Autocad.

Código Matlab utilizado para la comprobación y cálculo de la cinemática directa e inversa como programa auxiliar:

- CinematicaInversa.m
- Directa.m

11. Bibliografía

[1] ROBOTIS, "ROBOTIS e-Manual," 2021, [Online]. Available: https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/.

[2] C. D. S. Al., "The Quest for Natural Machine Motion: An Open Platform to Fast-Prototyping Articulated Soft Robots," *IEEE Robot. Autom. Mag.*, vol. 24, pp. 48–56.

[3] maxon motor ag, "Una serpiente robótica que se introduce reptando en cualquier rincón.," 2012. https://www.maxongroup.es/maxon/view/application/Una-serpiente-robotica-que-se-introduce-reptando-en-cualquier-rincon.

[4] ANYbotics, "AN," *Repositories of the Anymal C Robot*, 2021. https://github.com/ANYbotics (accessed Apr. 04, 2021).

[5] M. B. et Al., "Keep Rollin'—Whole-Body Motion Control and Planning for Wheeled Quadrupedal Robots," *IEEE Robot. Autom. Lett.*, vol. 4, pp. 2116–2123.

[6] A. P. S. A. Sajee Mathew, "Overview of Amazon Web Services," 2021. https://docs.aws.amazon.com/whitepapers/latest/aws-overview/document-details.html#doc-history (accessed Apr. 04, 2021).

[7] G. Aguero, C. E., Koenig, N., Chen, I., Boyer, H., Peters, S., Hsu, J., … Pratt, "Inside the Virtual Robotics Challenge: Simulating Real-Time Robotic Disaster Response," *IEEE Trans. Autom. Sci. Eng.*, vol. 12, pp. 494–506, 2015.

[8] Amazon, "AWS RoboMaker," 2021. .

[9] L. S. Dalenogare, G. B. Benitez, N. F. Ayala, and A. G. Frank, "The expected contribution of Industry 4.0 technologies for industrial performance," *Int. J. Prod. Econ.*, vol. 204, pp. 383–394, Oct. 2018, doi: 10.1016/j.ijpe.2018.08.019.

[10] Aggity, "No Title," *Tendencias en el sector Industry 4.0 para 2020*, 2020. https://aggity.com/tendencias-en-el-sector-industry-4-0-para-2020/ (accessed Feb. 07, 2021).

[11] Jose Luis del Val, "Industria 4.0: la transformación digital de la industria," *Rev. Deusto Ing.*, 2016, [Online]. Available: https://revistaingenieria.deusto.es/tag/industria-4-0/.

[12] Roboteurs, "ROS MQTT Client," *Github*, 2018, [Online]. Available: https://github.com/Roboteurs/ros_mqtt_client.

[13] C. Gutiérrez, L. Juan, I. Ugarte, and V. Vilches, "Towards a distributed and real-time framework for robots: Evaluation of ROS 2.0 communications for real-time robotic applications." 2018.

[14] M. Zhang, Z. Zhang, N. Lotfi, and S. Esche, "Development of Automatic Reconfigurable Robotic Arms using Vision-based Control," 2017, doi: 10.18260/1-2--28170.

[15] K. Wyrobek, "The Origin Story of ROS, the Linux of Robotics - IEEE Spectrum," *IEEE Spectrum Automaton*, 2017.

[16] uFactory, "uArm repositories," 2019. https://github.com/uArm-Developer.

[17] P. Angelakis, "Generic Hardward Interface Repository," 2020, [Online]. Available: https://github.com/panagelak/Generic_Arm.

[18] M. A. Attyah, "Robotic Arm Pick & Place Project." https://github.com/mkhuthir/RoboND-Kinematics-Project.

[19] Nyrio One, "Nyrio One Repository," 2020. https://github.com/NiryoRobotics/niryo_one_ros (accessed May 07, 2021).

[20] Nyrio, "Documentation Nyrio One," 2021. https://niryo.com/docs/niryo-one (accessed May 07, 2021).

[21] wiki.ros.org, “ros_comm,” 2021. http://wiki.ros.org/ros_comm (accessed Apr. 06, 2021).

[22] R. Tellez, “A History of ROS (Robot Operating System),” 2019. https://www.theconstructsim.com/history-ros/ (accessed Apr. 06, 2021).

[23] L. Joseph and J. Cacace, *Mastering ROS for Robotics*. 2018.

[24] M. A. Eitan Marder-Eppstein, Vijay Pradeep, “actionlib,” 2013. http://wiki.ros.org/actionlib (accessed Apr. 06, 2021).

[25] D. Thomas, “ros_base,” 2017. http://wiki.ros.org/ros_base?distro=kinetic (accessed Apr. 06, 2021).

[26] O. S. R. Foundation, “Tutorial: Using a URDF in Gazebo,” 2014. http://gazebosim.org/tutorials/?tut=ros_urdf (accessed Apr. 06, 2021).

[27] ROS.org, “Create your own urdf file.” http://wiki.ros.org/urdf/Tutorials/Create your own urdf file (accessed Apr. 06, 2021).

[28] Moveit!, “Moveit Concepts,” 2021. https://moveit.ros.org/documentation/concepts/ (accessed Apr. 10, 2021).

[29] P. Bouchier and M. Purvis, “rosserial.” http://wiki.ros.org/rosserial (accessed Apr. 06, 2021).

[30] Ubiquity Robotics, “Github repository raspicam_node,” 2020. https://github.com/UbiquityRobotics/raspicam_node (accessed Apr. 06, 2021).

[31] Sainsmart, “Wiki Sainsmart 6 dof,” 2016. http://wiki.sainsmart.com/index.php/File:SainSmart_6.jpg (accessed Apr. 06, 2021).

[32] A.- Florinnicolescu, F.-M. Ilie, and T.-G. Alexandru, “Forward and Inverse Kinematics Study of Industrial Robots Taking Into Account Constructive and Functional Parameter’S Modeling,” *Proc. Manuf. Syst.*, vol. 10, no. 4, p. 164, 2015, [Online]. Available: http://icmas.eu/Journal_archive_files/Vol_10-Issue4_2015_PDF/157-164_NICOLESCU.pdf.

[33] A. Renfrew, “Book Review: Introduction to Robotics: Mechanics and Control,” *Int. J. Electr. Eng. Educ.*, vol. 41, no. 4, pp. 388–388, 2004, doi: 10.7227/ijeee.41.4.11.

[34] A. A. Hayat, R. G. Chittawadigi, A. D. Udai, and S. K. Saha, “Identification of Denavit-Hartenberg parameters of an industrial robot,” *ACM Int. Conf. Proceeding Ser.*, 2013, doi: 10.1145/2506095.2506121.

[35] O. G. Vinogradov, “Fundamentals of kinematics and dynamics of machines and mechanisms,” *Fundamentals of kinematics and dynamics of machines and mechanisms*. CRC Press, Boca Raton [etc, 2000.

[36] A. Ghediri, “Design and Construction of Robotic Palletizer,” *Public Work.*, no. Reaffirmed 2003, pp. 1–86, 2017.

[37] Ubiquity Robotics, “Raspberry Pi ROS Images,” 2021. https://downloads.ubiquityrobotics.com/pi.html (accessed Apr. 17, 2021).

[38] S. BIRCHFIELD, *Image Processing Analysis*. Cengage Learning, 2017.

[39] R. J. Radke, “CHAPTER 1 - Multi-View Geometry for Camera Networks,” in *Multi-Camera Networks*, H. Aghajan and A. Cavallaro, Eds. Oxford: Academic Press, 2009, pp. 3–27.

[40] S. Suzuki and K. A. be, “Topological structural analysis of digitized binary images by border following,” *Comput. Vision, Graph. Image Process.*, vol. 30, no. 1, pp. 32–46, Apr. 1985, doi: 10.1016/0734-189X(85)90016-7.

[41] Adafruit, “Documentation Adafruit-PWM-Servo-Driver-Library,” 2020. https://github.com/adafruit/Adafruit-PWM-Servo-Driver-Library (accessed Apr. 10, 2021).

[42] A. Bensoukehal, “Perspective Rectification Using Homography Planar: Plane Measuring,” 2015.