UNIVERSIDADE DA CORUÑA

Escola Politécnica Superior

Trabajo Fin de Grado

CURSO 2019/20

MODELADO DE BRAZO ROBÓTICO PARA VEHÍCULO SUBMARINO

Grado en Ingeniería en Tecnologías Industriales

ALUMNA/O

Daniel Romeo Gutiérrez

TUTORAS/ES

Francisco Javier Bellas Bouza

Félix Orjales Saavedra

FECHA

SEPTIEMBRE 2020

1 TÍTULO Y RESUMEN

1.1 Título

Modelado de brazo robótico para vehículo submarino.

1.2 Resumen

El presente TFG se enmarca dentro de la línea de investigación sobre vehículos autónomos del GII (Grupo Integrado de Ingeniería). El objetivo global de dicha línea es el de desarrollar vehículos capaces de realizar tareas de la forma más autónoma posible, desarrollando para ello los sensores e inteligencia necesarios. Uno de los primeros pasos para conseguir dotar a los vehículos de autonomía es contar con modelos precisos que permitan realizar simulaciones. De esta forma, se permite acelerar el desarrollo de controladores y evitar el riesgo del empleo de componentes caros en un entorno agresivo como es el medio acuático. Este trabajo fin de grado se centrará en el desarrollo de un modelo del brazo robótico equipado en el vehículo submarino híbrido ROV/AUV en el software de simulación Gazebo.

Título

Modelado de brazo robótico para vehículo submarino.

Resumo

O seguinte TFG enmárcase dentro da línea de investigación sobre vehículos autónomos do GII (Grupo Integrado de Enxeñaría). O obxetivo global da devandita línea é o de desenvolver vehículos capaces de realizar tarefas da forma máis autónoma posíble, desenvolvendo para iso os sensores e intelixencia necesarios. Un dos primeiros pasos para conseguir dotar ós vehículos de autonomía é contar con modelos precisos que permitan realizar simulacións. De esta forma, permítese acelerar o desenvolvemento de controladores e evitar o risco do emprego de compoñentes caros nun entorno agresivo como é o medio acuático. Este traballo fin de grao centrarase no desenvolvemento dun modelo de brazo robótico equipado no vehículo submarino híbrido ROV/AUV no software de simulación Gazebo.

Tittle

Robotic arm modeling for an underwater vehicle

Summary

The following TFG is part of the autonomous vehicles research line of GII (Integrated Group for Engineering Research). The global goal of this line is the development of vehicles capable of performing tasks in the most autonomous way possible, including the development of the necessary sensors and artificial intelligence. One of the first steps in achieving this autonomy in the vehicles is to have accurate models to perform simulations. By using simulators it is possible to accelerate the development of controllers and to prevent the risk associated with the use of expensive components in agressive environments, such as the acuatic. This final degree project will focus on the development of the robotic arm equipped in the hybrid ROV / AUV underwater vehicle model in Gazebo simulation software.

UNIVERSIDADE DA CORUÑA

Escola Politécnica Superior

TRABAJO FIN DE GRADO/MÁSTER

CURSO 2019/20

MODELADO DE BRAZO ROBÓTICO PARA VEHÍCULO SUBMARINO

Grado en Ingeniería en Tecnologías Industriales

Documento

MEMORIA

ÍNDICES

TABLA DE CONTENIDO

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

2 INTRODUCCIÓN

El Trabajo Fin de Grado que se presenta a continuación se enmarca en el contexto de la robótica autónoma, dentro del proyecto "SAILOR[1]" [1] [2], financiado por el Ministerio de Economía y Competitividad y en el que también participan el Grupo de Inteligencia Artificial Aplicada (GIAA) de la Universidad Carlos III y la empresa IXION Industry & Aerospace. Este proyecto, tal y como se describe más adelante, trabaja en el desarrollo de una plataforma robótica submarina que permita ampliar el uso de los vehículos submarinos no tripulados, conocidos por sus siglas en inglés como UUV (Unmanned Underwater Vehicle).

Como su nombre indica, los UUV son vehículos capaces de operar bajo el agua sin la necesidad de tripulantes. Según su nivel de autonomía, los UUV pueden clasificarse en dos tipos: los ROV (Remote Operated underwater Vehicle), operados remotamente por un piloto, y los AUV (Autonomous Underwater Vehicle), completamente autónomos, sin comunicación con un operador [3].

En la última década, la utilización de este tipo de vehículos para diferentes operaciones y aplicaciones de carácter industrial, científico e, incluso, académico se ha incrementado notablemente gracias a los numerosos avances tecnológicos y la evolución de las capacidades de los múltiples sensores que contienen. Hasta ahora, diversas tareas de construcción y mantenimiento bajo el agua eran realizadas por personal especializado, provisto de prendas especiales para resistir las condiciones adversas del entorno, como el frío y la presión, y equipos de suministro de oxígeno. Estas complejas operaciones tienen lugar a muy grandes profundidades, donde las altas presiones y las bajas temperaturas dificultan enormemente el trabajo de los operarios. La incorporación de este tipo de vehículos ofrece la posibilidad de realizar multitud de tareas, como son la construcción y reparación de estructuras offshore, plataformas marítimas, puertos y aerogeneradores marinos, entre otras, de una forma más rápida y segura, sin comprometer la integridad de los operarios.

El Grupo Integrado de Ingeniería (GII) de la Universidade Da Coruña (UDC), como miembro del proyecto SAILOR, ha trabajado durante los últimos años en un prototipo de submarino autónomo destinado a labores de reparación y mantenimiento subacuáticas. El objetivo de dicho proyecto consiste en el desarrollo de ayudas básicas y avanzadas que proporcionen al submarino la capacidad para ser operado tanto de manera autónoma (tipo AUV) como de manera remota (tipo ROV) de una forma sencilla.

Este prototipo, bautizado por sus desarrolladores como KAI, se encuentra en el Centro de Investigaciones Tecnológicas I del Campus Industrial de Ferrol. Estas instalaciones cuentan con un canal de ensayos hidrodinámicos, equipado con un generador de olas y un carro de remolque, que permite la realización de pruebas en condiciones semejantes a las reales y la toma de datos en tiempo real. No obstante, el transporte y puesta en marcha del vehículo suponen una tarea muy compleja y laboriosa: para transportar e introducir el KAI, de 200 kg de peso, en el canal es necesario emplear un carro de carga y un puente grúa no automatizado que lo sumerja en el interior. A mayores, para su control, se debe contar con un equipo de unos cuatro miembros del GII que monitoricen, controlen y pongan a funcionar el vehículo, junto con sus múltiples sensores y actuadores. Una vez en el canal, cualquier imprevisto o cambio

[1] SAILOR es el acrónimo de Submarino Autónomo para la Inspección de instalaciones Off-shore

necesario en los sensores u actuadores implica que el equipo retire el submarino del tanque, lo transporte hasta el laboratorio y allí realice las correcciones necesarias. Todos estos inconvenientes suponen un importante gasto de tiempo y de recursos, tanto humanos como económicos.

Estas limitaciones llevan a la necesidad de disponer de un simulador que permita representar el comportamiento del vehículo submarino dentro de escenarios y ambientes realistas en los que pueda comprobarse que este modelo simulado cumple con las condiciones del modelo real. Este trabajo requiere la aplicación de diversos conocimientos en el área de mecatrónica: es necesario desarrollar el modelo 3D y el modelo matemático de cada una de las plataformas del vehículo (submarino y brazo), implementar todos los sensores y actuadores necesarios para su funcionamiento y, finalmente, integrar las diferentes plataformas en un único modelo. Una vez validado, dicho modelo virtual será capaz de adaptarse a las posibles futuras modificaciones que se introduzcan en el modelo real, lo que supone una importante reducción en los costes y plazos de tiempo.

Partiendo de los modelos 3D en SolidWorks del vehículo submarino elaborados G. Saavedra Soto [4], este trabajo se centrará en la parte correspondiente al diseño e implementación de un modelo dinámico para el brazo robótico del vehículo submarino en el simulador de Gazebo controlado por ROS, que luego podrá ser exportado y unido al modelo de simulación del submarino, ya realizado por E.M. Corella Solís, también en Gazebo [5].

El simulador desarrollado en este Trabajo Fin de Grado deberá cumplir con los movimientos básicos del brazo robótico y con un entorno realista, en tanto que, como se dijo anteriormente, los resultados obtenidos con el modelo del brazo simulado coincidan, en un margen aceptable, con los del brazo robótico real. En futuros proyectos del GII se espera conseguir la implementación de los sensores y actuadores del vehículo, y la integración completa entre submarino y brazo.

2.1 Robots manipuladores

Según la norma ISO 8373 [6], un robot manipulador o brazo robótico es "un manipulador multifuncional reprogramable, controlado automáticamente, que puede estar fijo o moverse, diseñado para mover materias, piezas, herramientas, o dispositivos especiales, según trayectorias variables, programadas para realizar tareas diversas". Los brazos robóticos son uno de los tipos más comunes y familiares de robots: pueden ser vistos operando en grandes industrias en tareas de mecanizado, soldadura o montaje, entre otras; en salas de quirófano realizando operaciones de cirugía, o en centros logísticos efectuando tareas de clasificación, empaquetado y paletización.

Este tipo de robots está compuesto por dos sistemas o estructuras básicas:

1. **La planta física**, que agrupa a su estructura mecánica, sus transmisiones y su herramienta o efector final.
2. **El sistema de control**, constituido por el conjunto de sensores y actuadores que este tiene.

Dentro de los manipuladores robóticos pueden definirse una serie de conceptos básicos, mostrados a continuación:

- ***Punto terminal (TP)***: Punto de referencia que define el extremo terminal del robot, donde se encuentra su muñeca. Define el espacio de trabajo del robot.

- ***Punto central de la herramienta (TCP)***: Punto donde se ubica el origen del sistema de referencia de la herramienta. Define el espacio operacional del robo (no confundir con el espacio de trabajo.
- ***Espacio de trabajo (WS)***: Volumen espacial donde el robot puede situar su muñeca o extremo terminal**.** No debe tomarse con el efector final, pues este es un añadido del robot.
- ***Grados de libertad (GDL)***: Número de movimientos independientes que puede realizar el robot. Define además la cantidad de parámetros necesarios para determinar la posición y orientación del robot.

2.1.1 Morfología de manipuladores robóticos

Mecánicamente, la planta física de un brazo robótico está formada por un ***efector final***, empleado para manipular objetos, y una ***estructura mecánica articulada***, cuya tarea consiste en situar el extremo del robot en el espacio [7]. Esta estructura, también conocida como ***serial-link***, puede describirse como una cadena cinemática abierta, formada por un conjunto de cuerpos o eslabones, denominados ***links***, unidos a través de articulaciones, conocidas como ***joints*** [8]. Por lo general, uno de los extremos de la cadena se encuentra fijo en el espacio y constituye la base del manipulador, mientras que, el extremo opuesto, puede moverse libremente en el espacio y porta a la herramienta o efector final.

El nombre de "brazo" robótico se debe a la gran similitud de este con un brazo humano. Tal es así que se ha establecido una equivalencia entre la estructura del manipulador y la anatomía del brazo, tal y como muestra la **Figura 2-1**:

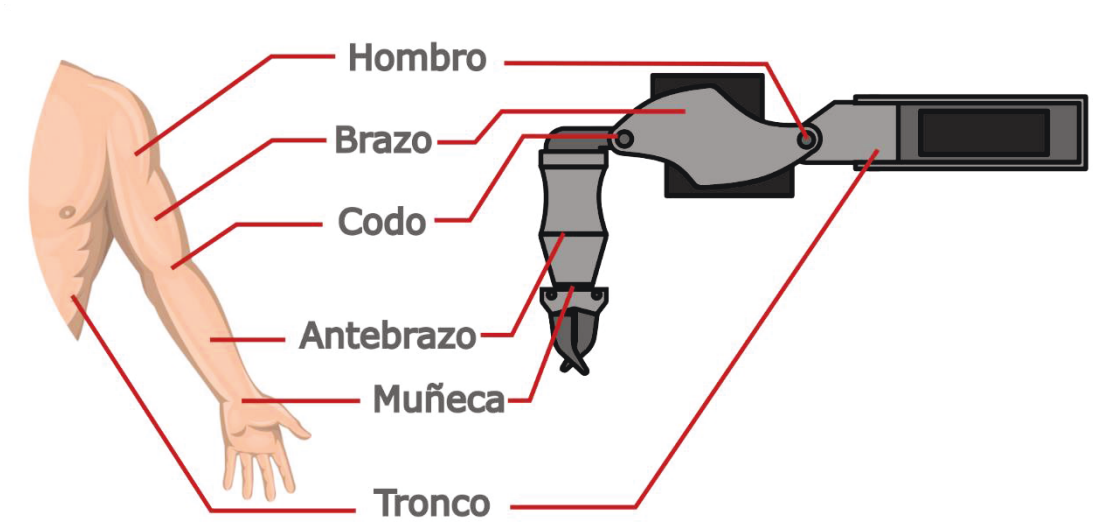


Figura 2-1. Equivalencia brazo robótico y brazo humano

Las articulaciones de un brazo robótico son mecanismos accionados por motores que unen dos eslabones sucesivos y representa un grado de libertad del manipulador. Se distinguen dos tipos básicos de articulaciones:

- ***Articulaciones rotoides***: El movimiento se realiza alrededor de un eje común entre los dos cuerpos unidos. La situación relativa entre ambos viene dada por el ángulo entorno a dicho eje (**Figura 2-2**):

Figura 2-2. Articulación rotoide

- ***Articulaciones prismáticas***: El movimiento se realiza a lo largo de un eje común entre los dos cuerpos. La situación relativa entre ambos viene dada por la distancia a lo largo de este eje (**Figura 2-3**):

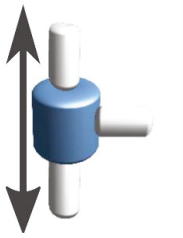


Figura 2-3. Articulación prismática

Las tres primeras articulaciones o grados de libertad del robot forman lo que se conoce como el ***portador*** del robot. Son las articulaciones encargadas de situar el extremo terminal del robot para que este llegue a cualquier punto de su espacio de trabajo. Las articulaciones restantes constituyen la muñeca, encargada de orientar extremo en el espacio.

El movimiento de estas articulaciones provoca una variación en el ángulo o posición relativa entre dos eslabones consecutivos. Como consecuencia, la posición y orientación que el efector final adopta en el espacio tridimensional dependerá de los ángulos o posiciones de estas articulaciones. Existirán por tanto dos espacios diferentes para definir la situación del robot:

- ***Espacio articular (ξ_A)***: Es el espacio que representa la situación de los diferentes cuerpos del robot en cuanto a los valores de sus articulaciones. Su dimensión coincide con el número GDL de la estructura
- ***Espacio operacional (ξ_O)***: Es el espacio en el que se representa la situación del punto central de la herramienta, *TCP*, del robot, definido respecto del extremo final o *TP*.

2.1.2 Criterios de clasificación

Los brazos robóticos pueden clasificarse atendiendo a diferentes criterios o características, entre los que se destacarán los siguientes:

Clasificación según el tipo de actuadores

En función del tipo de energía utilizada por los actuadores para generar movimientos en las articulaciones, se podrán distinguir tres tipos de robots:

- ***Robots Eléctricos***
- ***Robots Neumáticos***

- ***Robots Hidráulicos***

Clasificación según el número de ejes

Según la definición de la ISO 8373, para que un robot pueda ser considerado como manipulador robótico, debe tener por lo menos tres grados de libertad, pues son los necesarios para poder posicionar el extremo terminal en cualquier parte del espacio de trabajo. No obstante, para poder a mayores orientar el extremo en cualquier posición, son necesarios al menos seis grados de libertad. Se distinguen por tanto tres tipos de manipuladores:

- ***Robots no holonómicos*:** Aquellos que poseen menos de seis grados de libertad.
- ***Robos holonómicos***: Aquellos que poseen seis grados de libertad.
- ***Robots redundantes***: Aquellos que tienen más de seis grados de libertad.

Clasificación según la configuración

Las diferentes posibilidades en cuanto a la configuración y el tipo de las articulaciones del *portador*, proporciona un amplio abanico de posibilidades a la hora de describir los robots, entre los que destacan los de configuración **antropomórfica**, **tipo SCARA**, **cartesiana**, **cilíndrica** y **esférica**, mostrados en la **Figura 2-4**:

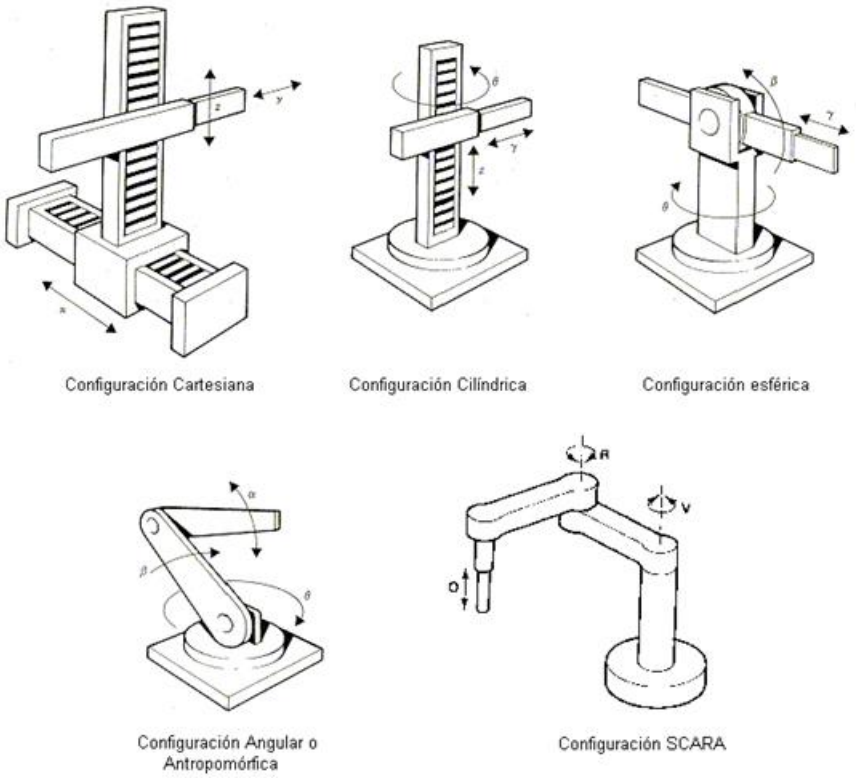


Figura 2-4. Diferentes configuraciones de manipuladores robóticos

2.2 Introducción a los modelos de simulación

Un simulador es un software informático que permite representar virtualmente un sistema o modelo real o hipotético de forma que sea posible estudiar su funcionamiento

o predecir su comportamiento [9]. La simulación es el proceso en el que se diseñan estos modelos lógico-matemáticos, también conocidos como ***modelos de simulación***.

Hoy en día, son muchos los sectores que se aprovechan de las ventajas ofrecidas por la simulación. En el sector del transporte, la simulación se ha convertido una herramienta de gran utilidad, permitiendo predecir flujos de vehículos en determinadas situaciones, así como su aplicación para la formación en conducción. Por otro lado, la simulación de procesos representa el corazón de la industria 4.0, al permitir diseñar, analizar y optimizar diferentes procesos técnicos.

Dentro de la simulación robótica, la mayor parte de sistemas emplean motores de física que permiten de emular las leyes físicas mediante variables como gravedad, masa, fuerzas y colisiones actuando sobre los objetos de la simulación, y así representar acciones reales. No obstante, existen también simuladores que reproducen únicamente los movimientos del cuerpos sin emplear dichos motores, lo que los hace más simples pero más sencillos de controlar.

A la hora de diseñar un determinado modelo de simulación, debe seguirse una metodología básica [10]:

1. **Definición del problema y planificación del estudio.**
2. **Recogida de datos.**
3. **Formulación y diseño del modelo.**
4. **Implementación y verificación del modelo.**
5. **Experimentación: protocolo de pruebas del modelo.**
6. **Validación del modelo.**

Una vez realizadas todas las fases anteriores y validado el modelo, este deberá funcionar sin problemas, pues se ha demostrado experimentalmente que los resultados ofrecidos son correctos.

3 ANTECEDENTES Y OBJETIVOS

3.1 Antecedentes

Atendiendo a la clasificación del capítulo 2, el brazo robótico estudiado en este trabajo, el TITANROB, mostrado en la **Figura 3-1** y del que se hablará con más detalle en el capítulo 4, puede catalogarse como un brazo robótico articulado de cuatro Grados de Libertad, provisto de actuadores eléctricos y con una configuración de tipo antropomórfica.

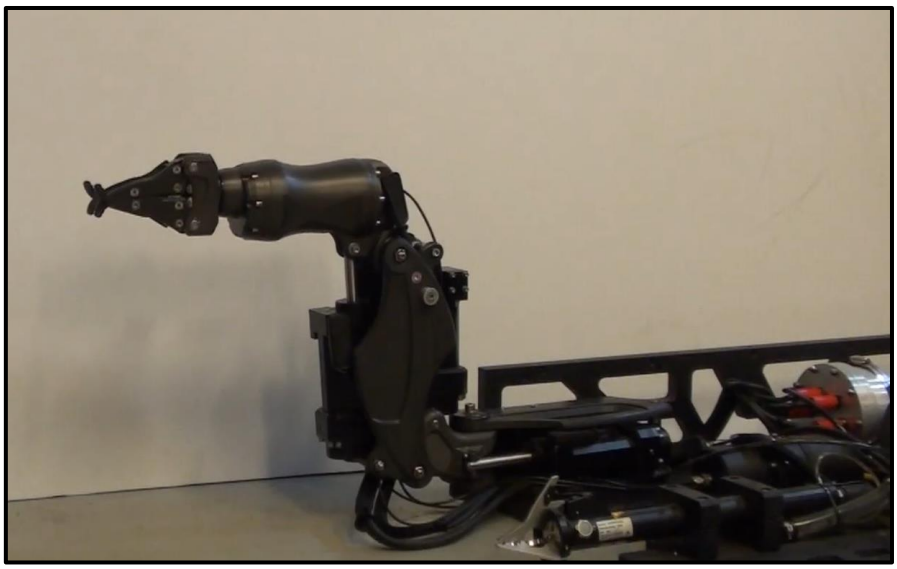


Figura 3-1. Brazo TITANROB E500

Actualmente, es cada vez mayor la presencia de este tipo de manipuladores robóticos en una amplia gama de actividades, desde el sector militar e industrial, hasta la medicina o el sector de servicios. Paralelamente, también se ha incrementado notablemente la elaboración de modelos de simulación de este tipo de robots. La utilización de simulaciones permite hacer pruebas con los equipos sin riesgo de que estos sean dañados y sin necesidad siquiera de disponer de su planta física, cuyo coste suele ser muy elevado.

Hoy en día existe un gran número de modelos de simulación de brazos robóticos empleados para diferentes tareas. A continuación, se presentarán brevemente algunos de los modelos más destacados.

3.1.1 IRB120

El IRB120 es un brazo robótico fabricado por la compañía ABB, multinacional con sede en Zúrich especializada en la automatización industrial. Forma parte de la última generación de robots de la marca, siendo el más pequeño de esta, con tan solo 25 kg de peso. Dispone de una configuración antropomórfica de seis ejes de movimiento, accionados por motores eléctricos paso a paso con una repetibilidad de 10 micras. Tiene una carga máxima de 3 kg y un alcance de 580 mm. Su gran agilidad y ligereza lo hacen

perfecto para tareas de *pick and collect*, siendo principalmente utilizado en la industria electrónica y farmacéutica. Dispone de una versión mejorada, el IRB120T, que proporciona un aumento notable de las velocidades máximas de los ejes 4,5 y 6, aportando mejoras de un 25% en los tiempos de ciclo [11].

Figura 3-2. Brazo robótico IRB120

El paquete ***abb_experimental***, disponible en la Wiki de ROS [12], proporciona varias plataformas de simulación para el robot, siendo Gazebo una de las opciones. Contiene todos los datos de configuración y archivos launch necesarios para simular el manipulador en Gazebo y su control a través de ROS. El simulador puede ser utilizado para cualquiera de las versiones del brazo, incluida la del IRB120T. El modelo de simulación de este paquete ha sido utilizado para diferentes trabajos, como la planificación de trayectorias mediante sensores [13] o para representación de tareas de robótica colaborativa [14].

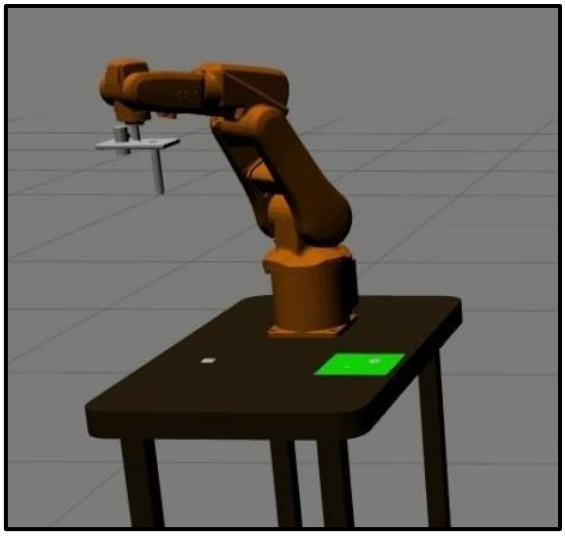


Figura 3-3. Simulación del brazo robótico IRB120 en Gazebo]

3.1.2 OpenMANIPULATOR-X

OpenMANIPULATOR-X es plataforma robótica completamente abierta desarrollada por ROBOTIS, proveedor global en soluciones de robótica, basada en *OpenSoftware*,

OpenHardware y *OpenCR* [15]. Consiste en un brazo robótico de cinco articulaciones movidas por servomotores tipo Dynamixel-X, programables y provistos de sensores para posición y velocidad. Se basa en *OpenSource*[2] y ROS, siendo completamente compatible con el paquete ***turtlebot_arm***. Además, está orientado al hardware abierto, pudiendo los usuarios descargarse las piezas del robot en formato *STL*, modificar el diseño y fabricarlas mediante impresión 3D. Todo ello lo hace ideal para el uso en centros educativos, como colegios y universidades, o para proyectos de investigación de robótica.

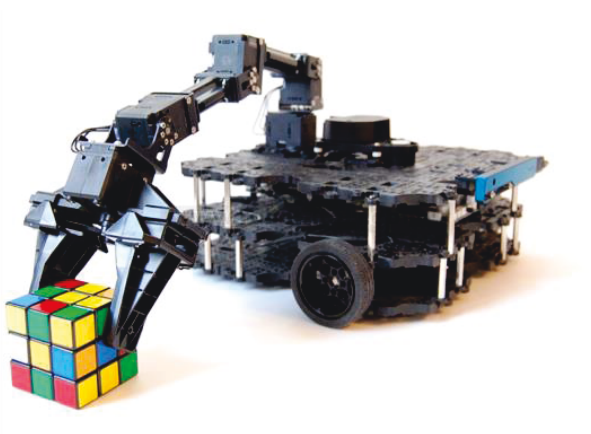


Figura 3-4. OpenMANIPULATOR-X

OpenMANIPULATOR-X dispone de un modelo de simulación en Gazebo, incluido en el paquete ***open_manipulator_gazebo*** disponible en los repositorios de ROS [16], con la que es posible conseguir una representación realista del comportamiento del robot, así como adaptar su hardware a las necesidades del usuario. Las simulaciones del OpenMANIPULATOR-X en Gazebo han sido utilizadas para multitud de proyectos como la implementación de un sistema de recogida de basuras y reciclaje [17], de Gloria del Olmo.

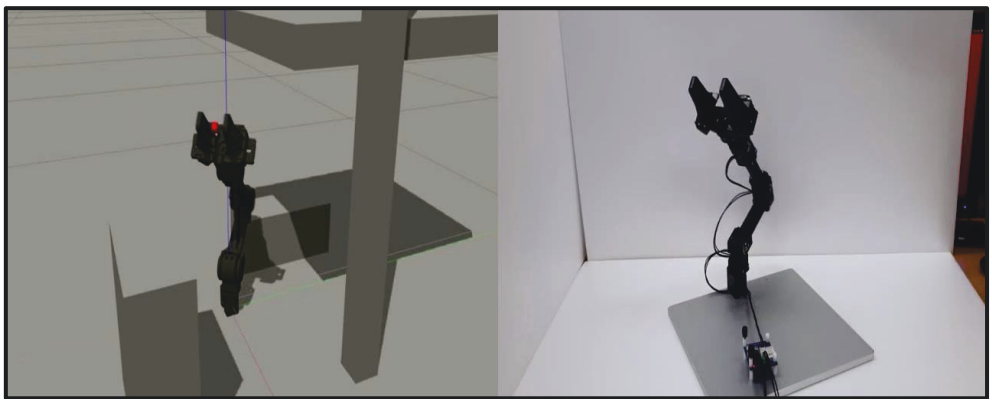


Figura 3-5. Simulación del OpenMANIPULATOR-X en Gazebo]

3.1.3 UR5e

El UR5e es un robot colaborativo industrial construido para aplicaciones industriales de servicio medio, perteneciente a la e-Series de Universal Robots. Presenta una

[2] OpenSource: Código abierto. Modelo de desarrollo de software basado en colaboración abierta.

configuración antropomórfica con 6 grados de libertad, que le proporcionan un alcance máximo de 850 mm y una carga útil de 5 kg [18]. Su versatilidad y adaptabilidad lo hacen perfecto para integrarse en una amplia gama de tareas que requieran poco peso, como pick and place, ensayos o análisis de laboratorio.

Figura 3-6. Brazo robótico UR5

El UR5e está disponible en Gazebo a través del paquete ***ur_gazebo*** disponible en los repositorios de ROS [19]. Algunos de sus modelos de simulación han sido empleados en diferentes proyectos, la mayoría de ellos para tareas de *pick and collect*, aunque también existen ejemplos para la localización y seguimiento de objetos mediante cámara USB [20].

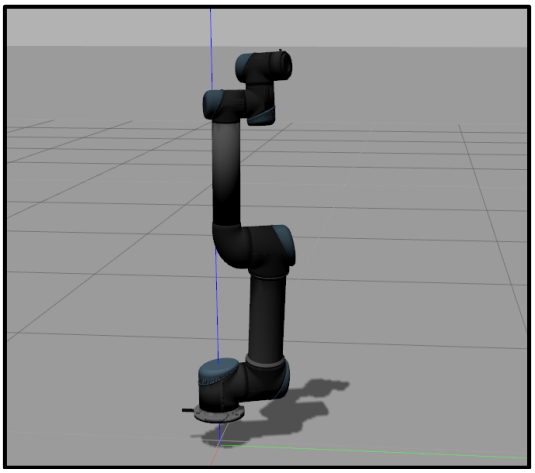


Figura 3-7. Simulación del brazo UR5 en Gazebo

3.1.4 Baxter

El Baxter es un robot industrial humanoide fabricado por Rethink Robotics e introducido al mercado en 2013, principalmente en PYMES, para desempeñar tareas colaborativas en líneas de producción. Dispone de dos brazos robóticos articulados con configuración antropomórfica con siete grados de libertad en cada uno de ellos. Ambos brazos cuentan con grippers en sus extremos, lo que, unido a su redundancia, le permite interactuar con objetos de hasta 2,27 kg, desplazarlos y posicionarlos en cualquier punto de su espacio de trabajo. A mayores, está provisto de una pantalla con cámara y

sensores infrarrojos en los brazos, que le permiten extraer información de su entorno y proyectar diferentes caras, dándole un aspecto más humano y amigable.

Figura 3-8. Robot Baxter

Además de su estructura humanoide, su principal diferencia respecto a otros manipuladores robóticos, por lo general programados para realizar tareas concretas, reside en su capacidad de aprendizaje. El Baxter posee la capacidad de guardar los movimientos que el operador le indica moviendo el extremo de uno de los brazos a los puntos deseados, los cuales memorizará y podrá repetir con gran precisión.

En los repositorios de GitHub, hay disponible una simulación de Gazebo para el robot Baxter con interfaces virtuales para el control del robot por ROS. Este simulador ha sido muy utilizado en GII para diversos proyectos de investigación, tales como la aplicación de técnicas de *Deep Learning* online [21], de Carmen Boado, o aprendizaje en tiempo real [22] para modelos de mundo de robótica cognitiva, de Álvaro Fernández.

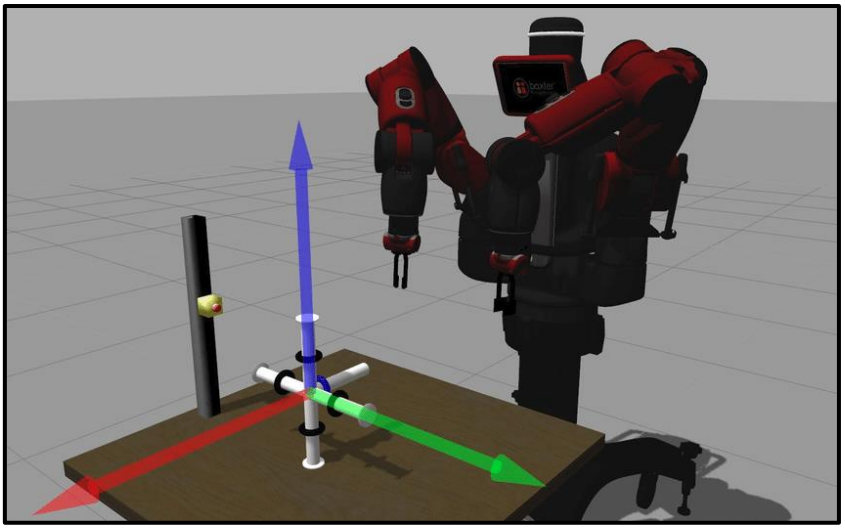


Figura 3-9. Simulación del Baxter en Gazebo

3.1.5 Análisis de los antecedentes

Como se puedo apreciar en las secciones anteriores, todos los robots comentados cuentan con modelos de simulación en Gazebo, lo que los hace compatibles con ROS, con los cuales es posible realizar la mayoría de los movimientos del robot real sin necesidad de disponer de este. Todos estos simuladores fueron empleados para labores de investigación, lo que demuestra la gran utilidad de la plataforma ROS/Gazebo para la simulación de robots.

Queda patente, por tanto, la necesidad de disponer de un modelo de simulación para el brazo TITANROB que sea compatible con ROS y Gazebo. No obstante, actualmente, no se ha desarrollado ningún modelo que cumpla con las especificaciones técnicas del brazo de estudio, lo que lleva a la necesidad de desarrollar un modelo simulación propio específico para el brazo TITANROB que sea compatible con ROS y que emplee el software de Gazebo.

3.2 Objetivos

El objetivo principal de este proyecto consiste en desarrollar el modelo de simulación de un brazo robótico para un vehículo submarino que represente fielmente su comportamiento, permitiendo así la realización de simulaciones realistas.

3.2.1 Subobjetivos

Para lograr el objetivo global, se han planteado los siguientes subobjetivos:

- Definición del modelo 3D de los componentes principales de la estructura del brazo robótico.
- Diseño del modelo matemático del brazo que describa su comportamiento en el espacio.
- Integración del modelo dinámico en el simulador Gazebo.
- Implementación y ajuste del control de las articulaciones del brazo robótico por ROS.
- Validación del modelo desarrollado para demostrar el correcto funcionamiento del brazo simulado.

4 ESPECIFICACIONES DE DISEÑO

A lo largo del presente capítulo se hará una descripción detallada de la metodología seguida durante la realización de este Trabajo de Fin de Grado. Se especificarán todas las herramientas y procedimientos matemáticos, así como los distintos componentes, físicos y virtuales, utilizados para el diseño e implementación de las diferentes partes que constituirán el simulador del Brazo Robótico TITANROB E500. A continuación, se establecerán los criterios y requerimientos esenciales impuestos por el GII para que el proyecto cumpla con los objetivos indicados en el capítulo anterior:

1. Se desarrollará un modelo de simulación del brazo robótico TITANROB E500.
2. El modelo fisicomatemático deberá cumplir con las especificaciones dimensionales y físicas del robot, recogidas en la documentación de G. Saavedra Soto [4].
3. Dicho simulador deberá ser compatible con *ROS Melodic Morenia* en el sistema operativo *18.04 Bionic Beaver*.
4. El simulador desarrollado deberá emplear el software de *Gazebo 9.0*.
5. Todas las articulaciones del modelo deberán ser controlables a través del entorno ROS.

4.1 Marco teórico

En esta sección se presentarán los principales fundamentos teóricos necesarios para comprender el comportamiento de los manipuladores robóticos en el espacio. En primer lugar, se introducirán algunos conceptos básicos sobre la orientación y posicionamiento de sólidos en el espacio. A continuación, se explicarán una serie de métodos, de carácter físico y matemático, que permitirán definir los modelos cinemático y dinámico de un manipulador robótico de n grados de libertad. A través de estas herramientas, se obtendrá un modelo matemático de la planta, con el cuál será posible conocer y representar matemáticamente el comportamiento del robot.

4.1.1 Conceptos básicos para la posición y orientación de sólidos en el espacio

Un requisito fundamental en la robótica es saber representar la posición y orientación de sólidos en espacios euclídeos. Para comprender la **Figura 4-1**, es necesario definir previamente una serie de conceptos geométricos básicos: Sistemas de coordenadas, Matrices de Rotación, Composición de rotaciones y Matrices de Transformación Homogénea.

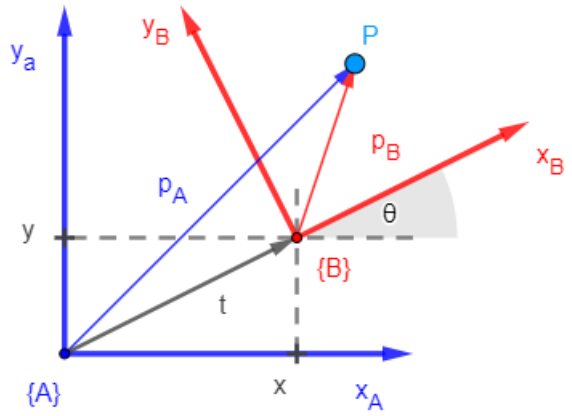


Figura 4-1. Sistemas de coordenadas {A} y {B}.

Sistema de coordenadas

Un sistema de coordenadas es un sistema que emplea un conjunto de valores, conocidos como **coordenadas**, para definir inequívocamente la posición de cualquier punto u objeto en un espacio euclídeo. De esta forma, cualquier punto ***P*** de un espacio tridimensional $\boldsymbol{R^3}$ puede ser definido por un sistema de coordenadas ***{A}*** mediante un vector ***p*** de tres componentes $\boldsymbol{(p_x, p_y, p_z)}$ como el de la **Ec. 4-1**.

$$p = p_x\hat{\imath} + p_y\hat{\jmath} + p_z\hat{k} \quad \textit{(Ec. 4-1)}$$

Tanto las posiciones como los objetivos del robot se definirán a largo de los ejes de distintos sistemas de coordenadas. Distinguiremos cuatro sistemas de referencia diferentes, cada uno de ellos adecuado a distintos tipos de movimientos y programaciones del robot (**Figura 4-2**):

- ***Sistema de coordenadas mundo o global* (*Q_w*):** Su origen se sitúa en un punto fijo del espacio y puede ser utilizado para describir varios robots a la vez
- ***Sistema de coordenadas de la base* (*Q_0*):** Se toma como referencia la base del robot. Por lo general, este sistema suele coincidir con el sistema de coordenadas mundo cuando solo se trabaja con un robot.
- ***Sistema de coordenadas de la herramienta* (*TCP*):** Este sistema posiciona su origen en el centro de la herramienta acoplada al manipulador, definiendo la posición y orientación de esta.
- ***Sistema de coordenadas del objeto de trabajo*:** En ocasiones, cuando se programa un robot, suele ser más adecuado trabajar con un sistema de coordenadas ubicado sobre el plano de trabajo en el que el robot desempeña su función. Con este sistema, al reubicar el objeto de trabajo, solo será necesario cambiar la posición del sistema de coordenadas para que automáticamente se actualicen las trayectorias.

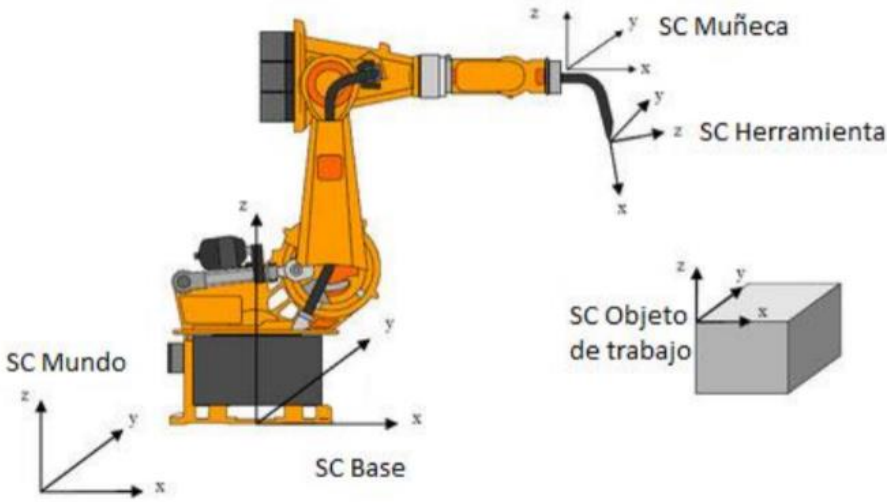


Figura 4-2. Sistemas de coordenadas Robot Industrial.

Volviendo a la **Figura 4-1**, se pueden observar dos sistemas de coordenadas, ***{A}*** y ***{B}***, y un punto ***P***, definido con respecto a ambos sistemas por los vectores de posición $\boldsymbol{p_A}$ y $\boldsymbol{p_B}$. El problema consistirá en definir el sistema de coordenadas ***{B}*** con respecto al sistema de coordenadas de referencia ***{A}*** y así poder determinar la relación entre los vectores $\boldsymbol{p_A}$ y $\boldsymbol{p_B}$. El sistema ***{B}*** se encuentra desplazado un vector ***t*** y posteriormente rotado un ángulo **θ** con respecto a ***{A}***. Por ello, el problema se dividirá en dos partes: **rotación** y **translación**, que serán tratados a continuación.

Matriz de Rotación

Una matriz de rotación ***R*** es una expresión matricial empleada para representar la orientación de un objeto o sistema en el espacio euclídeo con respecto a un determinado sistema de coordenadas que actúa como sistema de referencia [23].

A continuación, se estudiará la rotación para el caso de 2 dimensiones y posteriormente se comentará la extensión para 3 dimensiones.

A. Rotación en 2 Dimensiones

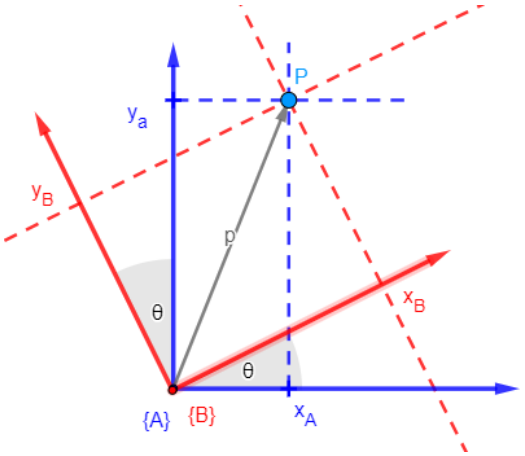


Figura 4-3. Rotación en el plano XY del sistema {B} con respecto al sistema {A}.

En la **Figura 4-3**, pueden distinguirse dos sistemas de coordenadas: un sistema ***{A}*** fijo en el espacio, que actúa como sistema de referencia, y otro sistema ***{B}*** que gira un ángulo **θ** con respecto al primero. Como se indicó anteriormente, el punto ***P*** puede

representarse respecto a los sistemas ***{A}*** y ***{B}*** con los correspondientes vectores de posición, $\boldsymbol{p_A}$ y $\boldsymbol{p_B}$:

$$p_A = x_A\,\hat{\imath}_A + y_A\,\hat{\jmath}_A = [\hat{\imath}_A \quad \hat{\jmath}_A]\begin{bmatrix} x_A \\ y_A \end{bmatrix} \qquad \textit{(Ec. 4-2)}$$

$$p_B = x_B\,\hat{\imath}_B + y_B\,\hat{\jmath}_B = [\hat{\imath}_B \quad \hat{\jmath}_B]\begin{bmatrix} x_B \\ y_B \end{bmatrix} \qquad \textit{(Ec. 4-3)}$$

El sistema de coordenadas ***{B}*** está completamente caracterizado por sus ejes ortogonales, definidos por los vectores unitarios $\boldsymbol{\hat{\imath}_B}$ y $\boldsymbol{\hat{\jmath}_B}$. Estos vectores pueden expresarse en función del sistema ***{A}*** como:

$$\hat{\imath}_B = \cos\theta\ \hat{\imath}_A + \sin\theta\ \hat{\jmath}_A$$

$$\hat{\jmath}_B = -\sin\theta\ \hat{\imath}_A + \cos\theta\ \hat{\jmath}_A \qquad \textit{(Ec. 4-4)}$$

Factorizando en forma matricial y despejando se obtiene:

$$[\hat{\imath}_B \quad \hat{\jmath}_B] = [\hat{\imath}_A \quad \hat{\jmath}_A]\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \longrightarrow$$

$$\begin{bmatrix} \hat{\imath}_A \\ \hat{\jmath}_A \end{bmatrix}[\hat{\imath}_B \quad \hat{\jmath}_B] = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \qquad \textit{(Ec. 4-5)}$$

Teniendo en cuenta que, al estar en el mismo origen, $\boldsymbol{p_A}$ y $\boldsymbol{p_B}$ representan el mismo vector, se tendrá:

$$[\hat{\imath}_A \quad \hat{\jmath}_A]\begin{bmatrix} x_A \\ y_A \end{bmatrix} = [\hat{\imath}_B \quad \hat{\jmath}_B]\begin{bmatrix} x_B \\ y_B \end{bmatrix} \longrightarrow$$

$$\begin{bmatrix} x_A \\ y_A \end{bmatrix} = \begin{bmatrix} \hat{\imath}_A \\ \hat{\jmath}_A \end{bmatrix}[\hat{\imath}_B \quad \hat{\jmath}_B]\begin{bmatrix} x_B \\ y_B \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}\begin{bmatrix} x_B \\ y_B \end{bmatrix} \qquad \textit{(Ec. 4-6)}$$

De esta manera, se transforman las componentes del punto ***P*** del sistema ***{B}*** al sistema ***{A}*** cuando el primero realiza una rotación de ángulo **θ** en sentido antihorario respecto al segundo. A este tipo de matriz se le conoce como **matriz de rotación** y se denota como $\boldsymbol{{}^{A}R_{B}(\theta)}$.

$$\prescript{A}{B}{R}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \qquad \textit{(Ec. 4-7)}$$

Algebraicamente, la matriz de rotación cumple dos propiedades: primero, es una matriz ortonormal, y segundo, es de determinante unitario:

$$R^T = R^{-1} \quad y \quad \det R = 1$$

Al ser una matriz ortonormal, es posible demostrar que cada una de sus columnas representan los vectores unitarios del sistema ***{B}*** con respecto a ***{A}***; del mismo modo, cada una de sus filas representan los vectores unitarios del sistema ***{A}*** con respecto a ***{B}***. Por su parte, tener determinante igual a 1 hace que el módulo del vector transformado (***p***) no varíe, esto es, ∀θ, $|\boldsymbol{p_A}|=|\boldsymbol{p_B}|$.

Para el caso de dos dimensiones, el grupo de rotaciones es conmutativo, es decir, el orden en que se realicen las rotaciones no influye en el resultado final. No así es, como se verá a continuación, para el caso de tres dimensiones, donde el conjunto de las rotaciones deberá seguir un orden determinado.

B. Rotación en 3 Dimensiones

La rotación en 3 dimensiones es una extensión del caso de 2 dimensiones en el que se añade un eje de coordenadas extra, ***Z***, de dirección perpendicular a los ejes ***X*** e ***Y*** según la regla de la mano derecha.

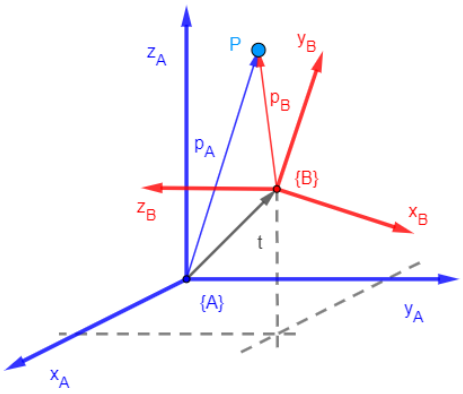


Figura 4-4. Rotación en R 3 del sistema {B} con respecto al sistema {A}.

Al igual que para el caso de 2 dimensiones, es posible representar la orientación de un sistema de coordenadas ***{B}*** expresando los vectores unitarios de sus ejes con respecto al sistema de referencia ***{A}***. Cada vector unitario del sistema ***{B}*** (con respecto al ***{A}***) forma una columna de una matriz 3x3 ortonormal ${}^{A}\boldsymbol{R_B}$:

$$\begin{bmatrix} x_A \\ y_A \\ z_A \end{bmatrix} = {}^{A}_{B}R(\theta) \begin{bmatrix} x_B \\ y_B \\ z_B \end{bmatrix} \qquad (Ec.\ 4\text{-}8)$$

Las siguientes matrices de rotación realizan rotaciones de vectores en sentido antihorario alrededor de los ejes ***X, Y,*** y ***Z***, en el espacio tridimensional:

$$R_X(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_Y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_Z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad (Ec.\ 4\text{-}9)$$

Como se destacó en el apartado anterior, a diferencia de las rotaciones en el plano, en tres dimensiones la rotación no cumple la propiedad conmutativa. Como consecuencia, el orden en el que se efectúe el producto de las matrices influirá en el resultado final obtenido. Esto se debe a que, en cada rotación, los ejes del sistema de coordenadas ***{B}*** varían su orientación respecto de los ejes del sistema de referencia ***{A}***.

Resulta por tanto necesario seguir un determinado convenio a la hora de efectuar varias rotaciones consecutivas. En el siguiente apartado, se explicarán las secuencias de rotaciones más utilizadas en el campo de la robótica.

Composición de Rotaciones

La composición de rotaciones consiste en una serie de métodos utilizados para descomponer matemáticamente un cambio de orientación arbitrario (de un cuerpo o sistema) en un conjunto de matrices de rotación en torno a diferentes ejes. Así, se obtiene una secuencia de rotaciones más simples en forma de producto matricial, en la cual el orden de los factores determinará el resultado final.

Uno de los métodos más ampliamente utilizados en robótica y navegación es el Teorema de Rotación de Euler. Este teorema establece que toda rotación arbitraria entre dos sistemas puede describirse como una secuencia de no más de tres rotaciones en torno a los ejes del sistema, tal que dos rotaciones sucesivas nunca se efectúen sobre el mismo eje. Estas tres rotaciones están definidas por tres ángulos de rotación, **ϕ**, **θ** y **Ψ**, asociados a los correspondientes ejes, conocidos como ángulos de Euler.

En total, existen doce posibles secuencias diferentes agrupadas en dos tipos:

- **Las secuencias Eulerianas**, que implican la repetición, no sucesiva, de rotaciones sobre un eje. Existen seis posibles configuraciones: ***XYX***, ***XZX***, ***YXY***, ***YZY***, ***ZXZ*** o ***ZYZ***.
- **Las secuencias Roll-Pitch-Yaw (RPY)**, caracterizadas por tres rotaciones en torno a tres ejes diferentes. Existen seis posibles configuraciones: ***XYZ***, ***XZY***, ***YXZ***, ***YZX***, ***ZXY*** o ***ZYX***.

Para este trabajo, se emplearán las secuencias de ángulos Roll-Pitch-Yaw. Los tres ángulos empleados son el **alabeo (*roll* θ_r)**, la **elevación (*pitch* θ_p)** y la **dirección (*yaw* θ_y)**. Estos tres ángulos, roll, pitch y yaw, son equivalentes a tres maniobras de rotación consecutivas en torno a los ejes ***X***, ***Y*** ***Z*** respectivamente. El orden en el que se realicen estas maniobras definirá la secuencia y determinará el resultado final. Por comodidad, y siguiendo la configuración por defecto del simulador de Gazebo, se empleará la secuencia ***ZYX*** (**Figura 4-5**):

$$R = R_Z(\theta_y)R_Y(\theta_p)R_X(\theta_r) \qquad (Ec.\ 4\text{-}10)$$

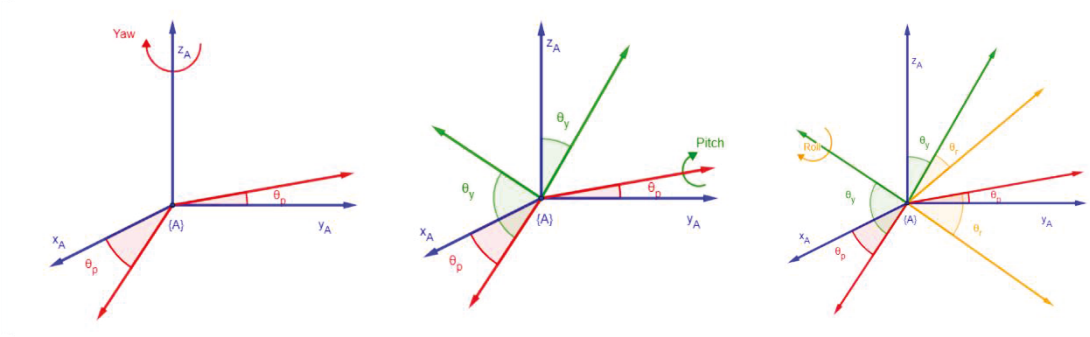


Figura 4-5. Ángulos de Euler RPY en configuración ZYX.

Vector de Translación

En contraposición con la rotación, una traslación es un movimiento en el cual se modifica la posición de un objeto o sistema sin varias su orientación en el espacio. Puede representarse mediante lo que se conoce como **vector de translación**.

Un vector de traslación ***t*** es un vector que permite representar el desplazamiento de un objeto o sistema en el espacio con respecto a un determinado sistema de referencia.

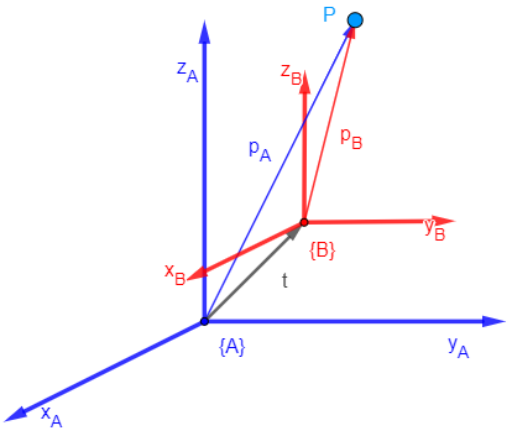


Figura 4-6. Traslación del sistema {B} con respecto al sistema {A}.

En la **Figura 4-6**, puede observarse que el sistema de coordenadas ***{B}*** se encuentra desplazado por un vector ***t*** del sistema de referencia ***{A}***. La posición del punto ***P*** con respecto al sistema ***{A}*** vendrá dada por la siguiente expresión:

$$p_A = p_B + t = x_A\hat{\imath} + y_A\hat{\jmath} + z_A\hat{k} =$$

$$= (x_B + t_x)\hat{\imath} + (y_B + t_y)\hat{\jmath} + (z_B + t_z)\hat{k} \qquad (Ec.\ 4\text{-}11)$$

En forma matricial:

$$\begin{bmatrix} x_A \\ y_A \\ z_A \end{bmatrix} = \begin{bmatrix} x_B \\ y_B \\ z_B \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix} = \begin{bmatrix} x_B + t_x \\ y_B + t_y \\ z_B + t_z \end{bmatrix} \qquad (Ec.\ 4\text{-}12)$$

Donde ***t*** es el vector de translación del sistema ***{B}*** con respecto al ***{A}***.

En resumen, para representar completamente la situación de un cuerpo en el espacio es necesario definir tanto su posición como su orientación con respecto a un determinado sistema de referencia mediante vectores de traslación y matrices de rotación. Toda esta información puede expresarse de forma más compacta en una única expresión conocida como **matriz de transformación homogénea**.

Matriz de Transformación Homogénea

Se define como matriz de transformación homogénea ***T*** a una matriz que permite la representación conjunta de la traslación y la rotación de un sistema de coordenadas respecto a otro. Esta matriz estará compuesta a su vez por cuatro submatrices de diferente tamaño que representarán la rotación, la translación, la perspectiva y el escalado [7].

$${}^A_B T = \begin{bmatrix} Rotación & Traslación \\ Perspectiva & Escalado \end{bmatrix} = \begin{bmatrix} {}^A_B R_{3x3} & t_{1x3} \\ f_{1x3} & w_{1x1} \end{bmatrix} \qquad (Ec.\ 4\text{-}13)$$

- ***${}^A_B R$***: Matriz de rotación entre el sistema ***{A}*** y el sistema ***{B}***.
- ***t***: Vector de translación
- ***f***: Vector de perspectiva (se emplea en visión artificial). Para robótica **(0, 0, 0)**.
- ***W***: Factor de escalada. Será igual a **1**.

Al igual que ocurría con las matrices de rotación, una transformación homogénea entre dos sistemas también puede descomponerse en el producto matricial de una

secuencia de transformaciones (rotaciones y translaciones) más simples, en el cual el orden de los factores será determinante en el resultado final. En concreto, la transformación entre dos sistemas, ***{0}*** y ***{n}***, puede descomponerse en el producto de ***n*** transformaciones homogéneas ${}^{j-1}\boldsymbol{T}_j$ de la siguiente forma:

$$ {}^{0}_{n}T = \prod_{j=0}^{n} {}^{j-1}_{j}T = {}^{0}_{1}T\ {}^{1}_{2}T \dots {}^{n-2}_{n-1}T\ {}^{n-1}_{n}T \qquad (Ec.\ 4\text{-}14) $$

4.1.2 Cinemática de Manipuladores Robóticos

La cinemática es la rama de la mecánica que estudia el movimiento de los cuerpos, o sistemas de cuerpos, sin considerar sus masas o las fuerzas que actúan sobre estos. Como se comentó en el Capítulo 2, un robot manipulador (*serial-link*), es una cadena cinemática abierta formada por un conjunto de cuerpos o eslabones (*links*), unidos a través de articulaciones (*joints*). Por lo general, uno de los extremos de la cadena se encuentra fijo en el espacio y constituye la base del manipulador, mientras que, el extremo opuesto, puede moverse libremente en el espacio y porta la herramienta o efector final (*end-effector*).

Cada articulación representa un grado de libertad del manipulador, y podrá ser de dos tipos: rotoide (*R*) o prismática (*P*). El movimiento de estas articulaciones provoca una variación en el ángulo o posición relativa entre dos links consecutivos y, como consecuencia, el efector final adopta una determinada posición y orientación en el espacio.

Dentro de la cinemática de manipuladores robóticos, existen dos problemas básicos: el *problema cinemático directo* y el *problema cinemático inverso* (**Figura 4-7**) [8].

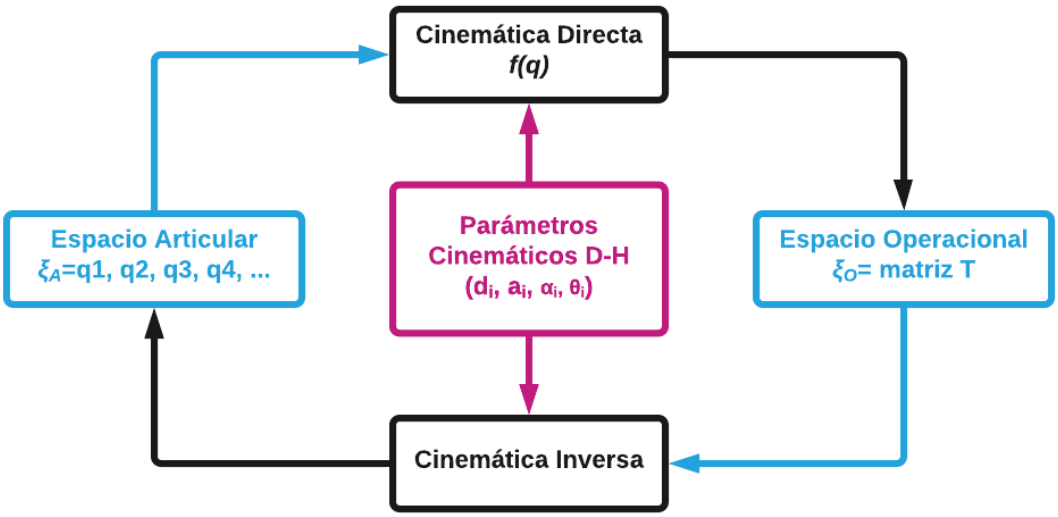


Figura 4-7. Esquema Cinemática de Manipuladores Robóticos.

1. El ***problema cinemático directo*** consiste en determinar la posición y orientación del efector final del robot respecto de un sistema de coordenadas fijo (por lo general el sistema de coordenadas la base) en función de sus coordenadas articulares, así como sus parámetros geométricos. De esta forma, la cinemática directa permite expresar el espacio operacional del

robot (posición del efector final) como una función del espacio articular (coordenadas articulares) de la forma:

$$\xi_E(x, y, z) = f(q) \qquad (Ec.\ 4\text{-}15)$$

Donde ξ_E representa el espacio operacional y $f(q)$ es una función dependiente de las variables del espacio articular ξ_A.

2. El ***problema cinemático inverso***, por otro lado, determinará los valores de las coordenadas articulares del robot para los que su extremo se sitúe en una determinada posición y orientación respecto a la base.

   Sin embargo, dado que el brazo robótico del trabajo únicamente puede ser controlado eje a eje, no será necesario estudiar este problema.

Por tanto, para la resolución del problema cinemático, será necesario poder expresar la posición y orientación de cada link del manipulador con respecto a su anterior, por lo que se le asignará un sistema de referencia local a cada uno de los eslabones, pudiendo así relacionar estos por medio de transformaciones homogéneas. Concatenando las diferentes transformaciones entre eslabones de manera secuencial, será posible expresar la posición del efector final respecto de la base del robot.

Aunque para describir la relación que existe entre dos elementos contiguos se puede hacer uso de cualquier sistema de referencia ligado a cada elemento, la forma más habitual para describir la geometría de una cadena cinemática y abierta en robótica es la conocida como notación de Denavit-Hartenberg [24].

Método de Denavit-Hartenberg

Este método, propuesto en 1955 por J. Denavit y R. Hartenberg, introduce algunos conceptos clave sobre cinemática de manipuladores robóticos en cadena abierta. Se trata de un método matricial que establece la localización que debe tomar cada sistema de coordenadas asociado a cada eslabón de la cadena articulada y así sistematizar la obtención de las ecuaciones cinemáticas de la cadena completa en una única matriz de transformación homogénea.

Para un manipulador con **N** articulaciones, numeradas de ***1*** a ***N***, hay ***N + 1*** links, numerados de ***0*** a ***N***. El link ***0*** corresponde a la **base** del robot mientras que el link ***N*** corresponde al eslabón que porta la **herramienta** o **efector final**. La articulación ***j*** conecta el link ***j - 1*** con el link ***j***. Sobre cada articulación ***j*** se sitúa un sistema de referencia ***{S_{j-1}}***, que será el sistema asociado al link ***j - 1***, y se moverá solidariamente a este.

Escogiendo los sistemas de coordenadas ***{S_{j-1}}*** asociados a cada eslabón ***j - 1*** según la representación propuesta por Denavit-Hatenberg será posible pasar de uno al siguiente mediante cuatro transformaciones básicas que dependen exclusivamente de las características geométricas del eslabón. Si bien una matriz de transformación homogénea queda definida por 6 grados de libertad, el método de Denavit-Hartenberg, permite, en eslabones rígidos, reducir este a cuatro grados de libertad, siempre y cuándo se elijan correctamente los sistemas de coordenadas.

Estas cuatro transformaciones básicas consisten en una sucesión de rotaciones y translaciones que permiten relacionar el sistema de referencia del eslabón ***j - 1*** con el sistema del eslabón ***j***. Las transformaciones en cuestión son las siguientes:

1. Rotación alrededor del eje $\boldsymbol{Z_{j-1}}$ un ángulo $\boldsymbol{\theta_j}$.
2. Traslación a lo largo de $\boldsymbol{Z_{j-1}}$ una distancia $\boldsymbol{d_j}$.
3. Traslación a lo largo de $\boldsymbol{X_j}$ una distancia $\boldsymbol{a_j}$.

4. Rotación alrededor del eje X_j un ángulo α_j.

Dichas transformaciones deben realizarse en el orden indicado, quedando así:

$$
{}^{j-1}_{j}T(\theta_j, d_j, a_j, \alpha_j) = R_Z(\theta_j)t_Z(d_j)t_X(a_j)R_X(\alpha_j) =
$$

$$
\begin{bmatrix} cos\,\theta_j & -sin\,\theta_j & 0 & 0 \\ sin\,\theta_j & cos\,\theta_j & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_j \\ 0 & 0 & 0 & 1 \end{bmatrix}
\begin{bmatrix} 1 & 0 & 0 & a_j \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\,\alpha_j & -sin\,\alpha_j & 0 \\ 0 & sin\,\alpha_j & cos\,\alpha_j & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} =
$$

$$
\begin{bmatrix} cos\,\theta_j & -cos\,\alpha_j\,sin\,\theta_j & sin\,\alpha_j\,sin\,\theta_j & a_j\,cos\,\theta_j \\ sin\,\theta_j & cos\,\alpha_j\,cos\,\theta_j & -sin\,\alpha_j\,cos\,\theta_j & a_j\,sin\,\theta_j \\ 0 & sin\,\alpha_j & cos\,\alpha_j & d_j \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \textit{(Ec. 4-16)}
$$

Donde θ_j, d_j, a_j y α_j son los parámetros D-H para el eslabón j.

Sin embargo, para que la matriz *T* obtenida relacione a los sistemas $\{S_{j-1}\}$ y $\{S_j\}$ es necesario que los sistemas se hayan escogido siguiendo unas determinadas normas. Estas normas, junto con la definición de los cuatro parámetros de Denavit Hartenberg, conforman el siguiente algoritmo para la resolución del problema cinemático directo que se comenta a continuación:

1. Numerar los eslabones del manipulador de ***0*** (base fija del manipulador) a ***n*** (eslabón de la herramienta).
2. Numerar cada articulación de ***1*** (que unirá los eslabones ***0*** y ***1***) a ***n*** (que unirá los eslabones ***n-1*** y ***n***).
3. Localizar el eje de rotación (articulaciones rotoides) o translación (articulaciones prismáticas) de cada articulación ***j*** y situar sobre cada uno el eje z_{j-1} correspondiente, con ***j*** de ***1*** a ***n***.
4. Situar el origen del sistema de referencia asociado a la base $\{S_0\}$ en cualquier punto del eje z_0. Los ejes x_0 e y_0 se situarán de manera que formen un sistema dextrógiro con z_0.
5. Con ***j*** de ***1*** a ***n***, situar los orígenes de los sistemas $\{S_j\}$ (solidarios al eslabón ***j***) en la intersección del eje z_j con la perpendicular común a z_{j-1} y z_j, que partirá del origen del sistema $\{S_{j-1}\}$.
6. La asignación de los ejes x_j sigue unas reglas muy precisas en función de la geometría del brazo robótico. Dependiendo de la relación espacial entre z_{j-1} y z_j, pueden distinguirse dos casos:
7. Si z_{j-1} y z_j no son paralelos, situar el eje x_j en la perpendicular común a ambos ejes.
8. Si z_{j-1} y z_j son paralelos, situar el eje x_j en el plano que contiene a ambos ejes y perpendicular a estos.
9. Situar el eje y_j de modo que forme un sistema dextrógiro con x_j y z_j.
10. Los parámetros d_j y a_j vendrán dados por las distancias, medidas sobre los ejes z_{j-1} y x_j respectivamente, entre los orígenes de $\{S_{j-1}\}$ y $\{S_j\}$.
11. El parámetro α_j vendrá dado por el ángulo en torno al eje x_j entre los ejes z_{j-1} y z_j. Por su parte, el parámetro θ_i vendrá dado por el ángulo en torno al eje z_{j-1} entre los ejes x_{j-1} y x_j.

12. Situar el sistema ***{S_n}*** en el extremo de la cadena cinemática de modo que ***z_n*** coincida con ***z_{n-1}***.

13. Obtener las matrices de transformación ***$^{j-1}T_j$*** entre los sistemas ***{S_{j-1}}*** y ***{S_j}*** y, a partir de estas, calcular la matriz de transformación ***^0T_n*** entre los sistemas ***{S_0}*** y ***{S_n}***.

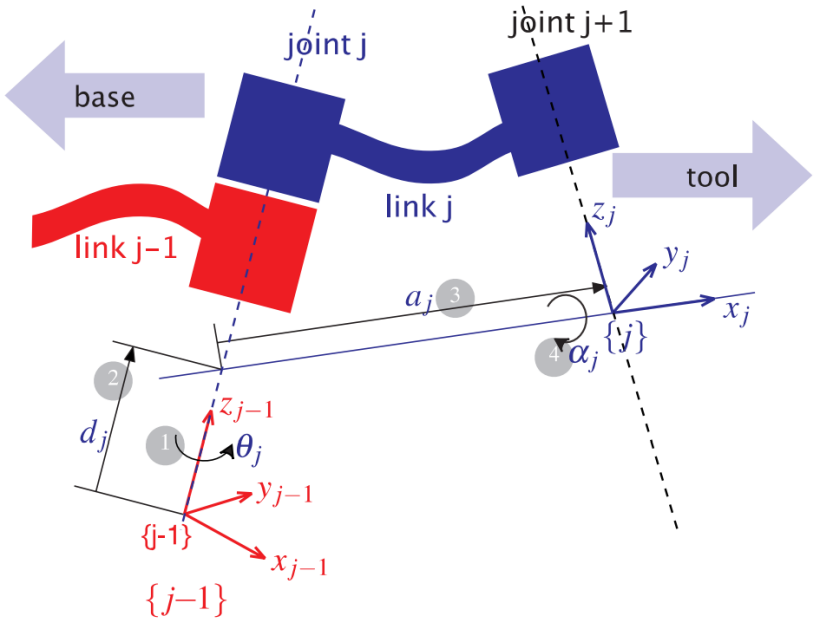


Figura 4-8. Definición parámetros D-H de un eslabón de articulación rotoide.

Por tanto, los cuatro parámetros de D-H **(θ_j, d_j, a_j y α_j)** dependerán únicamente de las características geométricas de los eslabones y articulaciones. Los parámetros ***a_i*** y ***α_i*** son siempre constantes. Para articulaciones rotoides **θ_j** es variable y ***d_j*** constante, mientras que para articulaciones prismáticas **d_i** es variable y **θ_j** constante.

Una vez obtenidos los parámetros de Denavit-Hartenberg, la relación entre cada par de eslabones consecutivos (***j-1*** e ***j***) vendrá dada por las matrices ***$^{j-1}T_j$***, obtenidas a través de la expresión de la **Ec. 4-16**. Finalmente, la relación entre la *base* y el eslabón ***n*** podrá expresarse como el producto de las ***n*** transformaciones homogéneas ***$^{j-1}T_j$*** (**Ec. 4-14**).

La matriz ***^0T_n*** obtenida definirá la orientación y posición del efector final respecto de la base en función de las coordenadas articulares que, para el caso de un robot antropomorfo, serán los ángulos **θ_j** asociados a cada articulación, quedando así resuelto el problema cinemático directo y, con ello, definido el modelo cinemático del robot.

4.1.3 Dinámica de Manipuladores Robóticos

La dinámica es la rama de la mecánica que estudia el movimiento de los cuerpos debido a las fuerzas externas e internas que actúan sobre estos. La dinámica de manipuladores robóticos estudia el movimiento del efector final del robot como la composición de los movimientos de cada uno de sus eslabones, los cuales a su vez son desplazados por las fuerzas y pares ejercidos sobre las articulaciones del robot.

Considérese el ejemplo de la **Figura 4-8**, donde la articulación ***j*** conecta a los eslabones ***j-1*** y ***j***. Cada eslabón del robot somete a una serie de fuerzas y pares a los eslabones anteriores que lo soportan, a la vez que es sometido a otros esfuerzos por parte de los eslabones consecutivos que él debe soportar, además de a su propio peso.

El actuador de la articulación ***j*** produce un par que genera una aceleración angular sobre el link ***j***, pero también provoca un par de reacción sobre el link ***j-1***. El peso resultante de la acción de la gravedad sobre los eslabones ***j*** a ***N*** ejerce un par sobre la articulación ***j***. A mayores, también se producen otras fuerzas giroscópicas y de inercia en función de la configuración del robot, de la velocidad y de la aceleración de las articulaciones.

La relación entre este conjunto de esfuerzos sobre los elementos del robot y su comportamiento dinámico final puede describirse mediante una serie de ecuaciones dinámicas conocidas como *ecuaciones del movimiento*, que conformarán su m*odelo dinámico*.

El *modelo dinámico* de un manipulador robótico establece una relación entre los diferentes esfuerzos que actúan sobre los elementos del robot y su respuesta articular, es decir, la posición, velocidad y aceleración que adquieren sus articulaciones. Dentro del *modelo dinámico* pueden considerarse, al igual que en la *cinemática de manipuladores*, dos problemas básicos: el ***problema dinámico inverso*** y ***problema dinámico directo*** (**Figura 4-9**) [8].

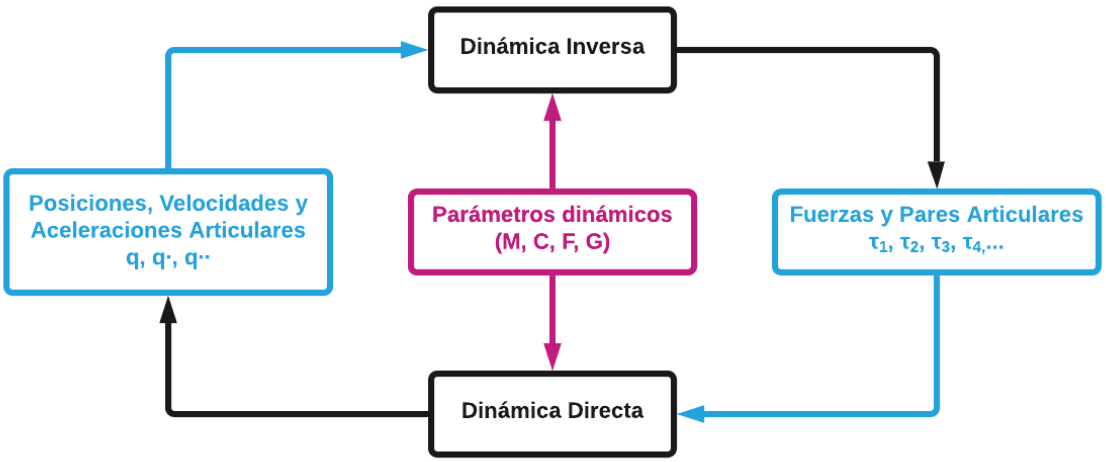


Figura 4-9. Esquema Dinámica de Manipuladores Robóticos.

El ***Problema Dinámico Inverso*** consiste en determinar las fuerzas y/o pares que actúan sobre cada articulación del robot a partir de la evolución temporal de las coordenadas articulares (posición, velocidad y aceleración de las articulaciones). Siguiendo la documentación de P. Corke [8], la ecuación de movimiento que rige el comportamiento de un manipulador robótico tiene la siguiente forma:

$$Q = M(q)\ddot{q} + C(q,\dot{q})\dot{q} + F(\dot{q}) + G(q) \qquad \textit{(Ec. 4-17)}$$

Donde:

- $Q = \begin{bmatrix} \tau_1 \\ \vdots \\ \tau_n \end{bmatrix}$ es la matriz de esfuerzos asociada a las coordenadas articulares.
- M es la matriz de inercia del robot.
- C es la matriz que agrupa las fuerzas centrífugas y de Coriolis.
- F describe las fuerzas de fricción viscosa y de Coulomb en los actuadores.
- G es el vector de cargas gravitacionales.
- $q\ \dot{q}\ \ddot{q}$ son los vectores de coordenadas, velocidades y aceleraciones articulares, respectivamente.

La **Ec. 4-17** puede desarrollarse a partir de métodos de dinámica clásica como la formulación de Newton-Euler de balance de fuerzas y momentos [25] o la formulación de Euler-Lagrange a partir del balance de la energía [26], la cual ha sido escogida para este proyecto y se demostrará más adelante.

Por otro lado, el ***Problema Dinámico Directo*** consiste en describir la evolución en el tiempo de las coordenadas articulares del robot en función de las fuerzas y/o pares que actúan sobre sus articulaciones.

La obtención del modelo no es demasiado compleja en robots de uno o dos GDL, pero a medida que el número de grados de libertad aumenta, el planteamiento y obtención del modelo se complica. El problema de la obtención del modelo dinámico de un robot es, por lo tanto, uno de los aspectos más complejos de la robótica, estudiado desde 1995 por numerosos investigadores, con diferentes enfoques y algoritmos. Sin embargo, a pesar de resultar complejo, el modelo dinámico es imprescindible para:

1. Dimensionamiento de los actuadores de las articulaciones
2. Diseño y evaluación de la estructura mecánica del robot.
3. Diseño y control del control dinámico del robot.
4. Simulaciones fiables del movimiento del robot.

Este último punto es el de mayor interés para este trabajo, pues su objetivo consiste en simular de manera realista los movimientos de un **brazo antropomórfico de cuatro GDL**, lo cual requiere un diseño óptimo y preciso de su correspondiente modelo dinámico. La gran complejidad, ya comentada, existente en la obtención del modelo ha motivado que se realicen ciertas simplificaciones.

Obtención de la Ecuación del Movimiento

Como ya se ha mencionado anteriormente, el modelo dinámico de un robot manipulador puede obtenerse mediante el desarrollo de la Ecuación de Euler-Lagrange [27]:

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{q}}\right) - \frac{\partial \mathcal{L}}{\partial q} = Q \qquad \textit{(Ec. 4-18)}$$

Donde $\mathcal{L}$ es el Langragiano y representa el balance de la energía almacenada en términos de energía cinética, K, y potencial, U:

$$\mathcal{L}(q, \dot{q}) = \mathrm{K}(q, \dot{q}) - U(q) \qquad \textit{(Ec. 4-19)}$$

El problema constará de cuatro pasos:

Paso 1. Cálculo de la Energía Cinética (K)

En primer lugar, se deberá obtener la energía cinética, K, la cual será igual a la suma de las energías cinéticas de los eslabones del manipulador, K_i:

$$\mathrm{K}(q, \dot{q}) = \sum_{i=1}^{n} \mathrm{K}_i \qquad \textit{(Ec. 4-20)}$$

Por ello, se deberán obtener las energías cinéticas, K_i, de cada eslabón del manipulador. La energía cinética de cada eslabón se compone de dos términos: la **energía cinética traslacional**, debida a la velocidad lineal, y la **energía cinética rotacional**, debida a la velocidad angular:

$$\mathrm{K}_i(q, \dot{q}) = \frac{1}{2} \boldsymbol{m_i v_i^T v_i} + \frac{1}{2} \boldsymbol{\omega_i^T I_i \omega_i} \qquad \textit{(Ec. 4-21)}$$

Donde:

- m_i es la masa del eslabón *i*.
- v_i y ω_i son las velocidades lineal y angular del CM del eslabón *i*, respectivamente.
- I_i es la matriz de inercia respecto del CM del eslabón *i*.

De la cinemática diferencial para manipuladores robóticos [28] sabemos que:

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} J_v \\ J_\omega \end{bmatrix} \dot{q} \quad \textit{(Ec. 4-22)}$$

Despejando de la **Ec. 4-20** y de la **Ec. 4-21**, la energía cinética total, K, quedará como:

$$\mathrm{K}(q,\dot{q}) = \frac{1}{2}\dot{q}^T \sum_{i=1}^{n} \{m_i [J_{vi}(q)\dot{q}]^T J_{vi}(q)\dot{q} + [J_{\omega i}(q)\dot{q}]^T I_i J_{\omega i}(q)\}\, \dot{q} \quad \textit{(Ec. 4-23)}$$

Expresado matricialmente:

$$\mathrm{K}(q,\dot{q}) = \frac{1}{2}\dot{q}^T M(q)\, \dot{q} \quad \textit{(Ec. 4-24)}$$

Donde $M(q)$ es una matriz cuadrada de orden n.

Paso 2. Cálculo de la Energía Potencial (U)

A continuación, deberán obtenerse la energía potencial, U, como el sumatorio de las energías potenciales de cada eslabón, U_i:

$$U(q) = \sum_{i=1}^{n} U_i \quad \textit{(Ec. 4-25)}$$

$$U_i(q) = m_i g^T p_{CMi} = m_i |g| L_{CMi} \sin q_i = m_i |g| h_{CMi} \quad \textit{(Ec. 4-26)}$$

Donde:

- g es el vector de gravedad.
- p_{CMi} es vector de posición del CM del eslabón *i* respecto del sistema mundo, de módulo L_{CMi}, y de componente vertical h_{CMi}.

Aplicando el sumatorio anterior (**Ec. 4-25**):

$$U(q) = \sum_{i=1}^{n} m_i |g|\, h_{CMi} \quad \textit{(Ec. 4-27)}$$

Paso 3. Cálculo del Langragiano ($\mathcal{L}$)

El tercer paso consiste en obtener el Langragiano del sistema, definido como la diferencia entre la energía cinética (**Ec. 4-23**) y potencial (**Ec. 4-27**):

$$\mathcal{L}(q,\dot{q}) = \frac{1}{2}\dot{q}^T M(q)\, \dot{q} - U(q) =$$

$$\frac{1}{2}\sum_{i=1}^{n} \dot{q}_i \sum_{j=1}^{n} M_{ij}(q)\, \dot{q}_j - \sum_{i=1}^{n} m_i |g| h_{CMi} \quad \textit{(Ec. 4-28)}$$

Paso 4. Cálculo de la Ecuación del Movimiento

Para el cálculo de la ecuación del movimiento a partir del Langragiano, en primer lugar, se calcula cada uno de los términos de la ecuación de Euler Lagrange por separado:

$$\frac{\partial \mathcal{L}}{\partial q_k} = \frac{1}{2}\sum_{i=1}^{n} \dot{q}_i \sum_{j=1}^{n} \frac{\partial M_{ij}}{\partial q_k}\, \dot{q}_j - \sum_{i=1}^{n} m_i |g| \frac{\partial h_{CMi}}{\partial q_k} \qquad \textit{(Ec. 4-29)}$$

$$\frac{\partial \mathcal{L}}{\partial \dot{q}_k} = \sum_{j=1}^{n} M_{kj}(q)\, \dot{q}_j \qquad \textit{(Ec. 4-30)}$$

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{q}_k}\right) = \sum_{j=1}^{n} M_{kj}(q)\, \ddot{q}_j + \sum_{i=1}^{n} \dot{q}_i \sum_{j=1}^{n} \frac{d}{dt} M_{kj}(q)\, \dot{q}_j \qquad \textit{(Ec. 4-31)}$$

Agrupando todos los términos:

$$\sum_{j=1}^{n} M_{kj}(q)\, \ddot{q}_j + \sum_{i=1}^{n} \dot{q}_i \sum_{j=1}^{n} \left(\frac{d}{dt} M_{kj}(q) - \frac{1}{2}\frac{\partial M_{ij}}{\partial q_k}\right) \dot{q}_j - \sum_{i=1}^{n} m_i |g| \frac{\partial h_{CMi}}{\partial q_k} =$$

$$\sum_{j=1}^{n} M_{kj}(q)\, \ddot{q}_j + \sum_{i=1}^{n} \dot{q}_i \sum_{j=1}^{n} c_{ijk} \dot{q}_j - \phi(q) = Q_k \qquad \textit{(Ec. 4-32)}$$

Expresando en forma matricial:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = Q \qquad \textit{(Ec. 4-33)}$$

Como se puede comprobar, la expresión obtenida es bastante similar a la seleccionada por Corke para describir el modelo dinámico de manipuladores robóticos (**Ec. 4-17**), con la salvedad de que en este caso no se incluye el termino relativo a las fuerzas de fricción viscosa y de Coulomb, $F(\dot{q})$. Esto se debe a que en el desarrollo anterior (**Ec. 4-33**) el término Q hace referencia a los pares desarrollados en las articulaciones del robot una vez sustraídos los esfuerzos internos del actuador (como estas fuerzas de fricción o las inercias internas del motor) y no al par total que desarrollan los motores, como es el caso de la **Ec. 4-17.**

Esto es justamente lo que ocurre en el simulador de Gazebo, los pares están referidos a las articulaciones, y no a los actuadores. Por tanto, la ecuación que definirá al modelo dinámico en el simulador será la **Ec. 4-33**.$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = Q$
(Ec. 4-33

En ambas ecuaciones, el esfuerzo predominante corresponde al término de la gravedad, $G(q)$. Este depende la posición del modelo, de tal forma que el esfuerzo será mayor, para un determinado eslabón, cuanto mayor sea la distancia de su centro de masas a la articulación que une con el anterior, en la dirección perpendicular a la gravedad (eje x del sistema global).

4.2 Componentes físicos

El robot de estudio de este trabajo es un brazo robótico articulado modelo TITANROB E500 (**Figura 4-10**), también conocido como TITAN, adquirido a la empresa ACSM Agencia Marítima S.L. en el año 2015, como parte del convenio "A-TEMPO[3]" [29], de la UDC con el Ministerio de Economía y Competitividad. Este brazo robótico se encuentra anclado al módulo intercambiable del submarino KAI, situado en la parte inferior del vehículo, sobresaliendo por

[3] A-TEMPO es el acrónimo de Avances en Tecnologías Marinas -Propulsión Naval y Offshore

la proa de este. Todas las especificaciones han sido tomadas en base a las exigencias establecidas en el Anexo II "Necesidades y especificaciones técnicas" al Documento descriptivo del brazo manipulador [30] y la documentación de G. Saavedra Soto [4].

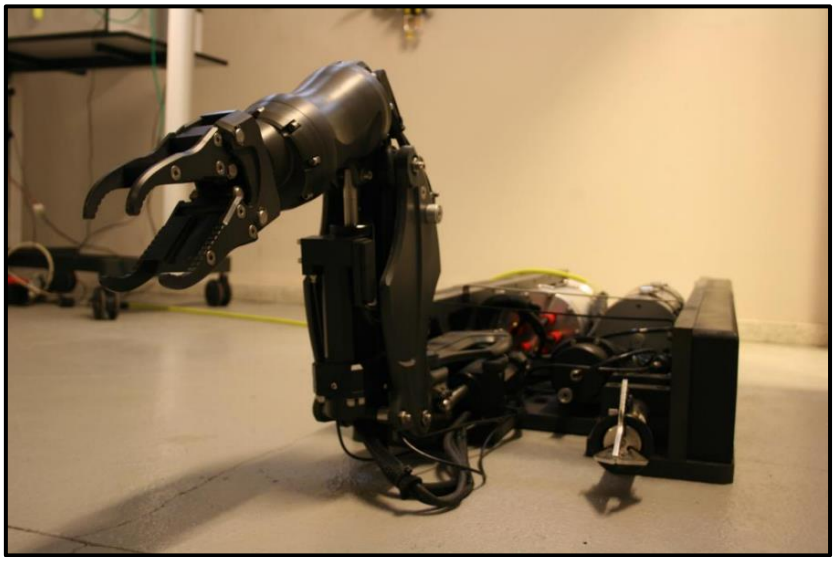


Figura 4-10. Brazo articulado TITANROB E500.

Se trata de un manipulador tipo ROV con la capacidad de manipular diversas herramientas, destinado a tareas de mantenimiento bajo el agua e intervención offshore[4]. Sin embargo, en este caso, al encontrarse todavía el sistema en una fase de desarrollo, podría clasificarse como un robot dedicado a la investigación.

Posee una configuración antropomórfica formada por cuatro articulaciones rotoides que le proporcionan sus cuatro grados de libertad: las tres primeras para poder situar el extremo en cualquier posición de su espacio de trabajo y una cuarta para el giro completo de la muñeca, donde se incorporará el efector final.

Respecto a los actuadores, las articulaciones están controladas por cuatro servomotores, uno por cada articulación. Se desconoce los sistemas de transmisión utilizados por los actuadores debido a la falta de especificaciones por parte del fabricante. No obstante, visualmente puede apreciarse que las articulaciones 1, 2 y 3, correspondientes a la cintura, el hombro y el codo del brazo, utilizan un sistema de piñón-cremallera para el movimiento de una serie de vástagos, uno anclado a la base del robot, para el movimiento de la articulación 1, y otros dos al eslabón del antebrazo, para el movimiento de las articulaciones 2 y 3 (**Figura 4-11**).

[4] offshore: “Alejado de la costa”, comúnmente referido a operaciones realizadas en alta mar.

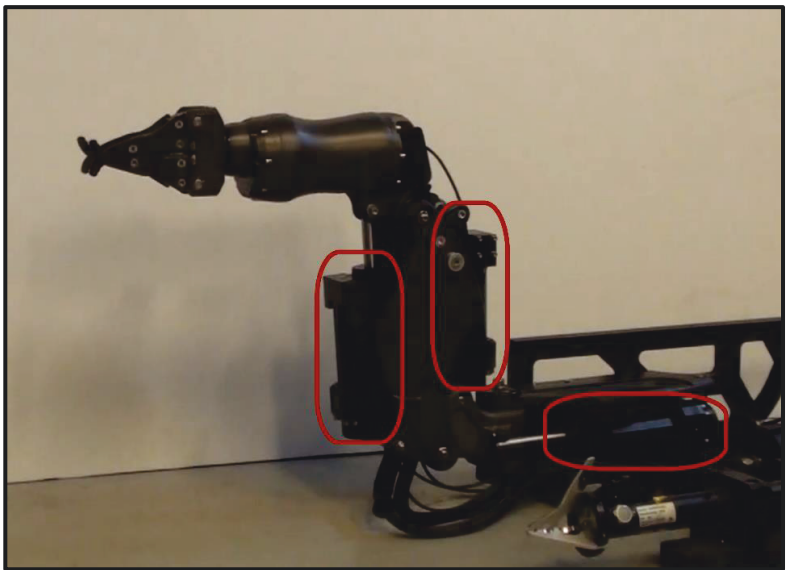


Figura 4-11. Vástagos de las articulaciones 1, 2 y 3.

Fabricado con una aleación de aluminio y con un peso de 23 kg, el TITANROB E500 puede sumergirse a una profundidad máxima de 300 m. Su alcance máximo desde la base hasta la muñeca es de 685 mm, abarcando 128° aproximadamente y estando el brazo completamente extendido, pudiendo manipular en esta posición cargas de hasta 25 kg en el agua. En la **Figura 4-12** se muestra el espacio de trabajo del brazo desde la muñeca para los planos horizontal y vertical.

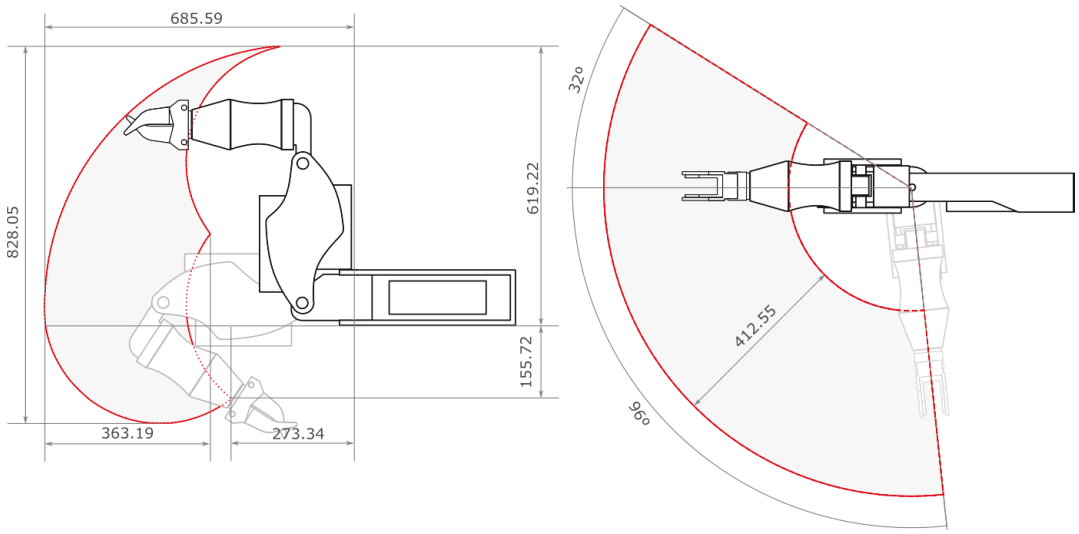


Figura 4-12. Espacio de trabajo desde el centro de la muñeca del TITANROB.

En cuanto a la alimentación, al tratarse de un módulo independiente del resto de componentes del submarino, el brazo cuenta con su propia batería, la cual alimenta al brazo mediante un cable umbilical, permitiéndole operar durante un tiempo mínimo de dos horas. La batería se encierra en un recipiente cilíndrico fabricado en aleación de aluminio 6061-T6, herméticamente sellado para asegurar su estanqueidad. El controlador del brazo está formado por una tarjeta de relés ethernet, que gobierna la batería, y una controladora RS485, para el movimiento del brazo, ambas alojadas en el interior de otro contenedor hermético para evitar que el agua entre en contacto con los componentes. Para un control remoto y en tiempo real del brazo, todo ello se conecta mediante un swtich ethernet a la estación del

operador dispuesta en superficie, en la cual también se conecta un controlador de tipo joystick, como el de la **Figura 4-13**, que permite gobernar la velocidad de giro de las articulaciones.

Figura 4-13. Joystick de control del Brazo TITANROB.

4.3 Componentes de Software

En esta sección, se presentarán los elementos de software más importantes empleados en el desarrollo del simulador para el brazo robótico Titan.

En primer lugar, se hará una introducción a ROS en la que se explicará que es, sus diferentes niveles de conceptos, su arquitectura, sus herramientas principales y algunos de los paquetes más importantes utilizados para este proyecto. A continuación, se hablará de Gazebo, un simulador multiplataforma de entornos 3D integrado en el sistema de ROS, muy utilizado para la simulación de robots articulados. Se comentarán sus arquitectura y componentes, así como las diferentes posibilidades que ambos ofrecen.

Además, también se mencionará otros programas de gran utilidad, como SolidWorks o Blender.

4.3.1 ROS

Figura 4-14. Logotipo de ROS.

ROS (*Robot Operating System*) [31] es un framework[5] utilizado para el desarrollo de aplicaciones robóticas que provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes. No es un sistema operativo, sino que proporciona las funcionalidades de este en un clúster[6] heterogéneo.

Entre las principales características de ROS destacan [32]:

- ***Computación distribuida***: En muchas ocasiones, es común dividir el software del robot en pequeñas partes independientes que cooperan entre sí para lograr un objetivo. ROS proporciona las herramientas que facilitan la implementación de este tipo de comunicación entre los programas.
- ***Reutilización de software***: Los avances en robótica han proporcionado numerosos algoritmos para resolver tareas típicas. No obstante, estos solo son útiles si hay alguna forma de aplicarlos a nuevos contextos sin necesidad de volver a implementarlos. ROS incluye paquetes donde ya se han implementado y probado versiones estables de muchos algoritmos importantes en robótica, sin restricciones de código e independientemente del lenguaje de programación.
- ***Facilidad para realización de pruebas***: ROS separa los sistemas de bajo nivel, para control de hardware, de los de alto nivel. Con ello, el software de bajo nivel se puede remplazar por simuladores, permitiendo concentrare en la parte de alto nivel del sistema.
- ***Popularidad***: Las principales fortalezas de ROS residen en su carácter de *OpenSource* y en la gran cantidad de usuarios que este tiene dentro del campo de la robótica. Esto ha propiciado la creación de una comunidad que se encarga de actualizar constantemente los repositorios y corregir cualquier mal funcionamiento de la plataforma.

En resumen, el uso de ROS permite trabajar con una única plataforma, la cual integra la comunicación del sistema, sin necesidad de procesadores ni mecanismos ajenos al software que ya trae integrado ROS.

Para su aprendizaje, ROS dispone de una página web [33] con tutoriales desde la instalación hasta la unión de ROS con otras plataformas, además de un foro [34] donde los usuarios pueden realizar y responder diferentes cuestiones relacionadas con el entorno.

Niveles de conceptos de ROS

ROS posee tres niveles de conceptos: el nivel del Sistema de archivos de ROS, el nivel de la Gráfica de Computación de ROS, y el nivel de la Comunidad ROS [35]. A continuación, se resumirán estos tres niveles.

Sistema de archivos de ROS (*ROS Filesystem*)

Este nivel cubre los recursos ROS que se encuentran en el disco, como:

- **Paquetes (*package*):** Son la unidad básica de organización del software de ROS. Un paquete puede contener nodos, librerías de dependencias, bases

[5] **framework**: Espacio de trabajo. En desarrollo de software, se refiere a una estructura conceptual y tecnológica de asistencia definida por un conjunto de herramientas y módulos de software

[6] **clúster**: sistemas distribuidos en granjas de computadoras, conectados entre sí por una red de alta velocidad, comportándose como un único servidor.

de datos, archivos de configuración, etc. Los paquetes son el elemento de construcción y ejecución más pequeño de todo el sistema ROS.

- **Metapaquetes (*metapackage*):** Son paquetes especializados que únicamente sirven para representar a otro grupo de paquetes relacionados entre sí.
- **Manifiestos de paquetes (*package manifest*):** Proporcionan metadatos sobre un paquete, como por ejemplo su nombre, versión, información de las licencias, dependencias, etc. El archivo *package.xml* se encuentra presente en cada paquete y recoge toda esta información.
- **Repositorios (*repositories*):** Son una colección de paquetes que comparten un mismo sistema VCS (Versión Control System), permitiéndoles tener una misma versión y ejecutarse conjuntamente. Pueden contener a un único paquete.
- **Tipos de Mensajes (*msg*):** Describen la estructura de datos de los mensajes en ROS
- **Tipos de Servicios (*srv*):** Describen las estructuras de petición- respuesta de los servicios en ROS.

Gráfica de Computación de ROS (*ROS Computation Graph*)

La Gráfica de Computación de ROS es la red de comunicación de todos los procesos de ROS que se ejecutan conjuntamente. Entre los conceptos básicos, destacan:

- **Nodos (*nodes*):** Son procesos ejecutables que realizan diferentes funciones. El sistema de control de un robot está compuesto por numerosos nodos. Los nodos de ROS se programan mediante el uso de las distintas librerías de ROS, como *roscpp* o *rospy*.
- **Máster (*Master*):** Proporciona el registro y búsqueda de nombres al resto de la Gráfica de Computación. Sin el Master, los nodos no podrían encontrarse, enviarse mensajes ni solicitar servicios.
- **Servidor de Parámetros (*parameter server*):** Permite el almacenamiento de datos en una ubicación central. Forma parte del Master.
- **Mensajes (*message*):** La comunicación entre nodos se realiza mediante el envío de mensajes. Los mensajes son estructuras de datos que comprenden diferentes campos de caracteres. Pueden incluir estructuras y matrices arbitrariamente anidadas.
- **Temas (*topics*):** Es el *nombre* empleado para identificar el contenido de un mensaje. Los mensajes son dirigidos a través de un sistema de transporte con una estructura de publicación/suscripción. Un nodo envía un mensaje al *publicarlo* en el *topic* determinado (*editor*). Cuando un nodo está interesado en un tipo de datos deberá *subscribirse* al *topic* apropiado (*subscriptor*).
- **Servicios (*service*):** Se emplean en las interacciones del tipo petición/respuesta. Están definidos por un par de estructuras de mensajes: una para la petición y otra para la respuesta. Un nodo proveedor ofrece un servicio bajo un determinado *nombre*. Un nodo cliente utiliza el servicio enviando un mensaje de petición y esperando la respuesta.

- **Bolsas (*bags*):** Son un formato empleado para guardar y reproducir información de mensajes de ROS, como, por ejemplo, la información de un sensor.

En este sistema de comunicación, el ROS Master actúa como un servicio de nombres. Almacena la información de registro de topics y servicios para los nodos ROS. Los nodos se conectan al Master para enviar su información de registro. Una vez se comunican con el Master, estos nodos pueden recibir información sobre otros nodos registrados y realizar conexiones según corresponda. Esta conexión entre nodos es directa; el Master únicamente proporciona la información de búsqueda de cada nodo, igual que un servidor DNS.

En esta arquitectura, los nombres juegan un papel esencial, pues son la forma en la que se conectan y comunican los nodos, topics, servicios y parámetros. En la **Figura 4-15** puede verse de forma esquemática cómo funciona la Gráfica de Computación:

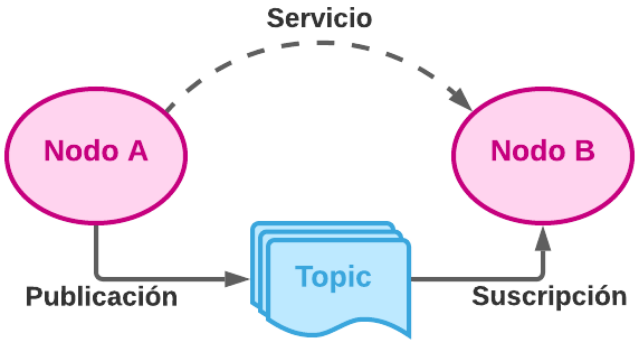


Figura 4-15. Gráfica de Computación de ROS.

Comunidad de ROS (ROS Community)

La Comunidad de ROS permite el intercambio de software y conocimiento entre diferentes usuarios y comunidades separadas a través de un único foro común. Entre sus recursos destacan:

- **Distribuciones:** Las Distribuciones de ROS son colecciones versionadas del sistema ROS que pueden instalarse. Son similares a las distribuciones de Linux
- **Repositorios:** ROS posee una red para el repositorio de código, en la que diferentes usuarios e instituciones pueden desarrollar y lanzar sus propios componentes de software
- **ROS Wiki:** La Wiki de ROS es el foro principal para documentar información sobre ROS. Cualquiera puede registrarse y contribuir con su propia documentación.
- **ROS Answers:** Un sitio de para responder y formular cuestiones relacionadas con ROS

Herramienta catkin

catkin es el sistema de construcción y compilación de ficheros oficial de ROS y el sucesor del sistema de construcción original, *rosbuild*. Este sistema combina macros de *CMake* y scripts de Python para proporcionar funcionalidades adicionales a las *CMake*, como la capacidad de construcción simultánea de múltiples proyectos dependientes o el soporte a la búsqueda automática de paquetes.

La introducción de la herramienta *catkin* permite la construcción de un tipo especial de paquete conocido como *catkin package* (paquete *catkin*). Para que un paquete pueda considerarse como paquete *catkin*, debe presentar las siguientes características:

- Debe contener un archivo *package.xml* compatible con *catkin* que proporcionará la metainformación del paquete.
- Debe contener un *CMakeList.txt* que use *catkin*.
- Cada paquete *catkin* debe tener una carpeta propia, no pudiendo existir los paquetes anidados.

Para su construcción se emplea el comando *catkin_create_pkg*. Estos paquetes *catkin* pueden ser construidos como un proyecto independiente. No obstante, *catkin* introduce un nuevo concepto que facilita la gestión y organización de los paquetes: los *workspaces* (espacios de trabajo). **ACP**.

Un *workspace* es un fichero en el que es posible construir simultáneamente múltiples paquetes *catkin* interdependientes. Es esencial para generar cualquier programa en ROS, pues ahí se guardará y trabajará con el código. Mediante el empleo del comando *catkin_make* se creará tres directorios: *src*, donde se almacenan los paquetes, *build*, donde se guardarán los resultados, y *devel*, que posee los ficheros y directorios de configuración del *workspace*. Dentro del directorio *src* se crearán los ficheros *CMakeList.txt* y *package.xml*, que proporcionan la información necesaria para poder trabajar con los paquetes.

Comandos principales de ROS

A la hora de utilizar ROS, existen tres comandos principales, los cuales se explican a continuación:

roscore: Es el primer comando que se ha de lanzar a la hora de ejecutar cualquier aplicación de ROS. Este comando ejecuta el Master y el servidor de parámetros, ya explicados anteriormente.

rosrun: Permite lanzar los programas utilizando únicamente el nombre del paquete y del programa a ejecutar. Para ejecutarlo, es necesario tener activo *roscore*.

roslaunch: Se utiliza para ejecutar los archivos de extensión *.launch*, los cuales contienen la información necesaria para el lanzamiento simultáneo de varios nodos.

Paquetes importantes de ROS

Para que sea posible implementar los controles del brazo robótico TITANROB en el simulador, será necesario instalar los siguientes paquetes, disponibles en los repositorios de ROS:

- *robot*: Metapaquete con las librerías necesarias para modelar el hardware de cualquier robot.
- *ros_control*: Paquete que incluye diversas interfaces y administradores de control, transmisiones e interfaces de hardware.
- *ros_controllers*: Librería que incluye los controladores proporcionados por ROS.

Estos paquetes proporcionarán las librerías y herramientas necesarias para controlar cada una de las articulaciones del modelo robótico de simulación como si del modelo real se tratará.

robot – URDF

robot es un metapaquete de ROS que proporciona un conjunto de librerías para el modelado cinemático y dinámico de manipuladores robóticos, además de un conjunto de herramientas para su validación y conversión a otros formatos.

Su paquete principal es el denominado *urdf*, que contiene los intérpretes de C++ necesarios para especificar el modelo de un robot en formato *URDF* (Unified Robot Definition Format). Se trata de un formato de archivos *XML*[7] empleado por ROS para describir todos los elementos de un robot. Este paquete contiene todas las especificaciones *XML* necesarias para el modelado de 3D de robots, sensores, escenas y otros elementos [36].

El formato *URDF* consiste en una serie de etiquetas escritas en *XML* que permiten especificar los diferentes atributos de cada eslabón y articulación del robot. A pesar de ser un sistema bastante genérico, pues permite describir desde un UAV hasta un brazo manipulador, posee ciertas limitaciones. Por ejemplo, no puede representar cadenas cinemáticas cerradas, permitiendo únicamente robots en serie (cadena cinemática abierta), ni eslabones flexibles, asumiendo en todo momento que el robot está formado por eslabones rígidos. No obstante, se empleará el formato *URDF*, pues el robot de estudio de este proyecto no se ve afectado por las restricciones anteriores: es cadena cinemática abierta de eslabones rígidos. Así, el *URDF* nos permitirá especificar los modelos cinemático y dinámico del robot, su representación visual (mediante un *STL*[8] de cada eslabón) y su modelo de colisión.

A continuación, se explicará en detalle como describir un robot en formato *URDF*. Este se tratará como un *serial-link*: un conjunto de eslabones (*links*) unidos por articulaciones (*joints*), tal y como se muestra en la **Figura 4-16**.

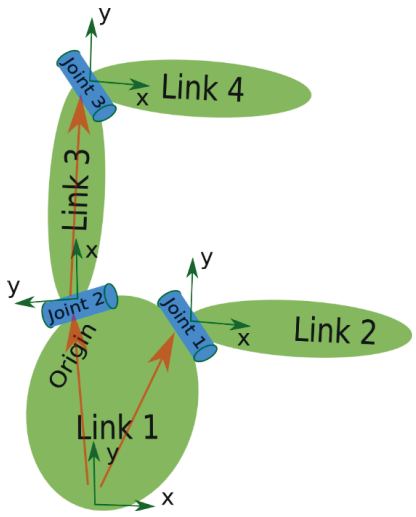


Figura 4-16. Sistema de links unidos por joints.

El elemento raíz del archivo URDF deberá ser un elemento <*robot*>, que pondrá contener cuatro tipos de elementos: <*link*>, <*joint*>, <*transmisión*> y <*gazebo*>. La etiqueta que abre el elemento deberá especificarse el atributo nombre (name).

```
<robot name="manipulador">
   <link> ... </link>
```

[7] **XML** siglas de *eXtensible Markup Language*, es un macrolenguaje que funciona como un estándar para el intercambio de información entre diferentes plataformas.

[8] **STL**, siglas de *STereoLithography*, es un formato de archivo par diseño asistido por computadora (CAD) que permite definir geometrías 3D.

```
    <joint> ... </joint>

    <transmission> ... </transmission >

    <gazebo> ... </gazebo>
</robot>
```

Cada eslabón estará definido por un elemento <*link*>. Este describirá un cuerpo rígido con inercia, representación visual y colisiones, por lo que se compondrá de tres bloques: <*inertial*>, <*visual*> y <*collision*>, tal y como muestra la **Figura 4-17**. Todos ellos tendrán un elemento <*origin*> que definirá su posición respecto al origen del eslabón. Al igual que para el elemento <*robot*>, deberá especificarse el nombre de cada eslabón.

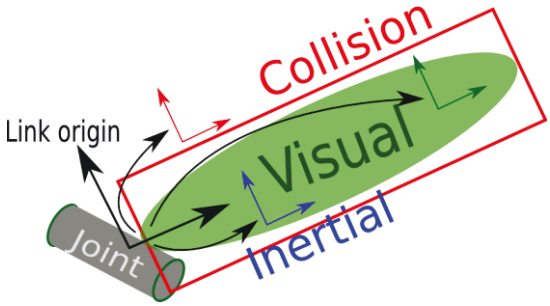


Figura 4-17. Link de URDF.

Por su parte, las articulaciones estarán definidas por elementos <*joint*>. Estos elementos conectan dos eslabones consecutivos, estableciendo entre ellos una relación de dependencia padre-hijo, donde el *link* hijo se mueve respecto a un eje ubicado en el *link* padre. A mayores, también especifican los límites y las propiedades dinámicas de cada articulación. Se tendrán por tanto los bloques <*child*>, <*parent*>, <*axis*>, <*limits*> y <*dynamics*>. Será necesario además especificar como atributos el nombre de la articulación y el tipo de ésta, que podrá ser fija (*fixed*), de revolución (*revolute*), continua (*continuous*), prismática (*prismatic*) o planar (*planar*).

Hay que señalar que cada eslabón *j* tendrá su origen en la articulación *j*, es decir, en el sistema de referencia *j-1*, a diferencia de lo que establecía el método de Denavit-Hartenberg, donde el origen del eslabón *j* se encontraba en la articulación *j+1*, en el cual se encontraba el sistema de referencia *j*. Esto se debe a que en el formato *URDF* el origen de cada *link* se sitúa en la unión con su *link* padre, como puede verse en la **Figura 4-18:**

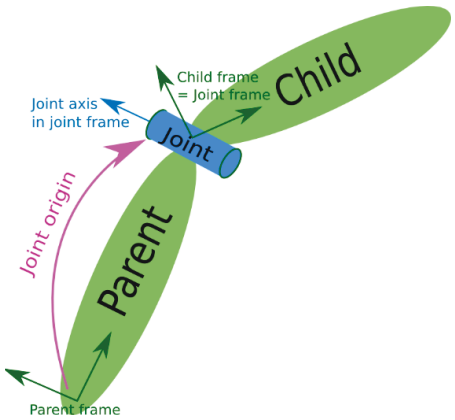


Figura 4-18. Joint de URDF.

Los elementos *<transmission>* y *<gazebo>* se describirán en secciones posteriores de este capítulo.

A mayores, los archivos *URDF* pueden ser adaptados a otro formato más eficiente conocido como *XACRO* [37]. Se trata de un lenguaje macro de *XML*, muy utilizado en *ROS* como forma de mejorar y optimizar los archivos *URDF*.

ros_control y ros_controller

El paquete *ros_control* proporciona un conjunto de librerías que permiten implementar y manejar los controladores necesarios para el manejo de un robot manipulador. Un controlador puede describirse como un sistema de control en bucle cerrado, más concretamente un *PID*, como el de la **Figura 4-19**. La señal de entrada *R(s)* es un punto tomado de referencia, como puede ser una posición, velocidad o aceleración deseadas en el robot. El *PID* controla la señal de salida $U_C(s)$, por lo general un esfuerzo, que es enviada a los actuadores del robot (real o simulado), cuya planta está modelizada por la función $G_P(s)$. La señal de entrada está realimentada por la señal de estado de las articulaciones *Y(s)* (posición, velocidad y aceleración de estas) proporcionada por los encoders de los actuadores para obtener la señal *E(s)*.

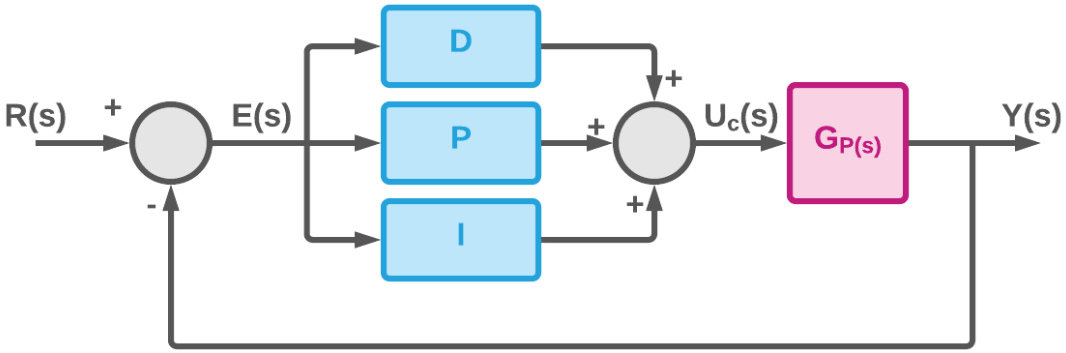


Figura 4-19. Controlador de ROS de PID.

Para el caso de estudio, la función de transferencia de la planta, $G_p(s)$, corresponde a la ecuación de la dinámica directa del robot manipulador. Se trata de un sistema de control que toma como entrada un vector con los pares aplicados sobre las articulaciones en un determinado instante, dando como respuesta las posiciones, velocidades y aceleraciones articulares en ese mismo instante.

Estas librerías implementan abstracciones en interfaces de hardware para poder enviar las salidas de los controladores al robot real o a su simulación en Gazebo empleando un plugin, del que se hablará en el siguiente apartado.

En la **Figura 4-20** se explica con mayor detalle todos los componentes de *ros_control*, los cuales pueden agruparse en dos bloques principales:

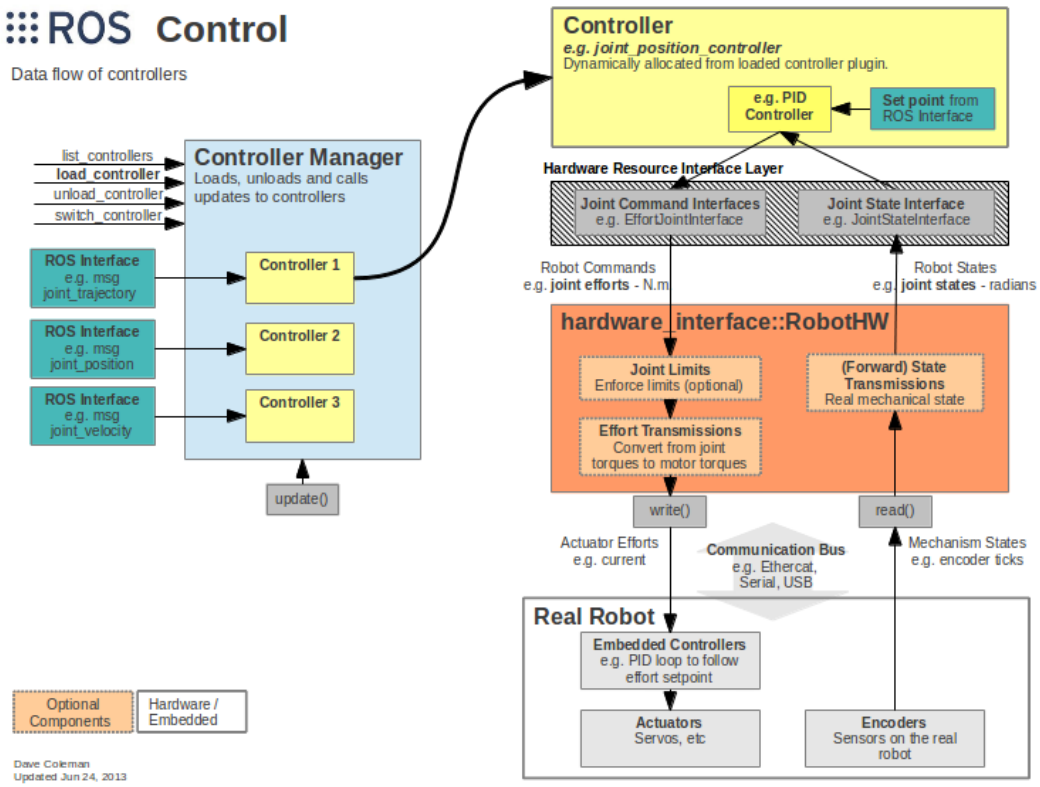


Figura 4-20. Diagrama de funcionamiento del paquete ros_control.

1. Los ***controladores*** (*controllers*) y el ***Controller Manager*** (contenidos en los bloques amarillo y azul, respectivamente). Estos controladores pueden agruparse en función del tipo de señal $U_C(s)$ enviada al hardware o simulador:

   - ***effort_controllers*****:** empleados para controlar las articulaciones por medio de señales $U_C(s)$ de esfuerzos de par motor. En función del tipo de control deseado por el usuario y definido por la señal $E(s)$:
     joint_effort_controller: control por par motor
     joint_position _controller: control de posición
     joint_velocity_controller: control de velocidad

   - ***position_controllers*****:** empleados para controlar las articulaciones por medio de señales $U_C(s)$ de posición.
     joint_position _controller
     joint_group_position _controller

   - ***velocity_controllers*****:** empleados para controlar las articulaciones por medio de señales $U_C(s)$ de velocidad.
     joint_velocity_controller
     joint_group_velocity_controller

   A mayores, se define otro controlador para publicar el estado de las articulaciones:

   - **joint_state_controller**

   Los controladores son gestionados por el *Controller Manager*, responsable cargar, detener y alternar entre ellos. Este es el encargado de proporcionar los

mecanismos y la infraestructura necesarios para interactuar con los controladores anteriores. Así, desde ROS el usuario manda una serie de mensajes, que son debidamente recibidos por el correspondiente controlador gracias a la correcta administración del *Controller Manager*.

2. La ***Capa de Abstracción de Hardware*** (*Hardware Abstraction Layer*, que agrupa los bloques gris y naranja). Los controladores de ROS no se comunican directamente con el hardware, sino que lo hacen a través del Nivel de Abstracción de Hardware, compuesto por dos niveles:

   Nivel 1. Las **interfaces de *ros_control***, dentro de la **capa de Recursos de la Interfaz de Hardware** (*Hardware Resource Interface Layer*), conectan los controladores con la interfaz de hardware, entre las que destacan:

   - *Joint Command Interface*: Interfaz diseñada para poder gobernar un conjunto de articulaciones. En función del tipo de controlador de las articulaciones, existen tres clases derivadas: Effort Joint Interface, Velocity Joint Interface y Position Joint Interface.
   - *Joint State Interfaces*: Interfaz diseñada para poder recibir e interpretar el estado de un conjunto de articulaciones, cada una con su correspondiente posición, velocidad y aceleración.

   Nivel 2. La **interfaz de hardware** conocida como ***RobotHW***, encargada de establecer la comunicación entre los controladores de ROS y la planta del robot real o simulador. Puede haber una o varias interfaces, todas ellas derivadas de la clase *hardware_interface::RobotHW*. Por un lado, se comunica con la planta tanto para leer el estado de sus articulaciones como para enviar órdenes a estas. Por otro, se comunica con la *Hardware Resource Interface* ya sea para enviar el estado de las articulaciones a la *Joint State Interface* como para recibir los comandos de las *Joint Command Interfaces*.

En el siguiente apartado se presentará el software de simulación empleado para este proyecto: Gazebo. Para lograr una conexión entre los controles de ROS y el software de Gazebo será necesario instalar una serie de paquetes que permitirán interactuar con la simulación de Gazebo a través de ROS.

4.3.2 Gazebo

Figura 4-21. Logotipo de Gazebo.

Gazebo (**Figura 4-21**) es un simulado de entornos 3D con la capacidad de representar de forma precisa y eficiente el comportamiento dinámico de numerosos modelos de robots (articulados, aéreos, submarinos, etc.) dentro de entornos complejos, tanto de interior como exterior. Ofrece la posibilidad de trabajar con modelos en tiempo real, simulando masas, inercias, fricciones, velocidades, aceleraciones y otros atributos

que le permiten conseguir una correcta aproximación de las condiciones del modelo real [38].

Entre las características que lo diferencian de otros simuladores disponibles y han hecho que sea idóneo para este trabajo, podrían destacarse las siguientes:

- ***Es de software libre***: Su carácter *OpenSource* hace que sea un software completamente gratuito que además puede ser reconfigurado, ampliado y modificado.
- ***Puede simular de manera realista la física de los cuerpos***: Tiene la capacidad de trabajar con diferentes variables, como gravedad, fricción o viento, entre otras. Además, los robots pueden interactuar con el mundo y viceversa.
- ***Permite desarrollar y simular modelos de robots propios***: Es posible crear modelos de robots personalizados a través del *Model Editor*, que permite incorporar al modelo formas simples como cubos, cilindros o esferas. Además, también es posible importar archivos de mallado STL, *meshes*, elaborados en otros programas como SolidWorks para una representación más realista.
- ***Permite crear escenarios de simulación propios***: Además del aspecto visual, es posible modificar las condiciones del entorno, como la fricción de contacto con el suelo, el viento o incluso la gravedad y así adaptar el mundo a las necesidades de cada proyecto.
- ***Contiene plugins para la incorporación de sensores***: En sus repositorios, Gazebo incluye numerosos sensores de odometría (GPS), fuerza, contacto, cámaras y láseres que pueden ser incorporados al modelo.
- ***Puede ser sincronizado con ROS***: El uso único de Gazebo no permite tener un control sobre los elementos del modelo, razón por la que se consideró necesario implementar ROS para el proyecto. Gazebo puede ser ejecutado desde ROS y así utilizar las APIs de este último para controlar los movimientos del robot en la simulación. De esta forma, el robot podrá publicar la información de sus sensores en nodos de ROS y se implementará una lógica que de las ordenes de control pertinentes al robot.

A mayores de todo lo anterior, hay que destacar que la elección de utilizar una integración de Gazebo y ROS para implementar el simulador del brazo TITANROB ha estado motivada en gran parte por el hecho de que se ha utilizado la misma combinación para el modelo dinámico del submarino KAI, elaborado por Eva M. Corella. De esta forma, la unión de ambos modelos (brazo y submarino) en posibles proyectos futuros será mucho más sencilla y directa.

Arquitectura de Gazebo

Gazebo emplea una arquitectura distribuida en diferentes librerías para física, renderizado, comunicación, generación de sensores e interfaz gráfica (**Figura 4-23**).

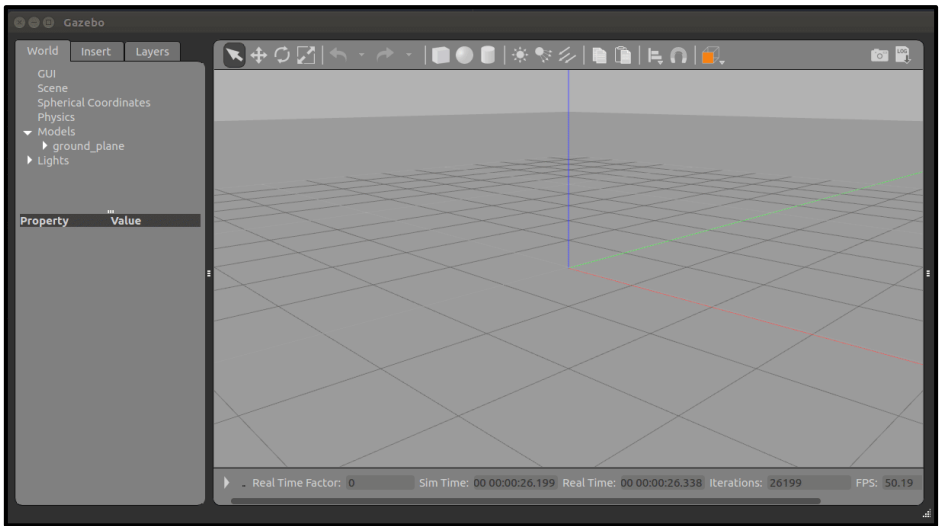


Figura 4-22. Interfaz gráfica de Gazebo.

Librería de Comunicación: La librería de comunicación es usada por el resto de las librerías, pues proporciona los mecanismos de comunicación y transporte de información de Gazebo.

Librería de Física: La librería física proporciona una interfaz para la simulación de componentes fundamentales como solidos rígidos o cuerpos de colisión y está integrada por cuatro motores de física de código abierto: *ODE* (*Open Dynamics Engine*), *Bullet*, *Simbody* y *DART* (*Dynamics Animation and Robotics Toolkit*).

Librería de Renderizado: El renderizado gráfico se realiza mediante *OpenGL* (*OGRE*) para la representación de escenas 3D. Incluyen iluminación, texturas y simulación de cielos.

Generación de sensores: La librería de generación de sensores se encarga de implementar todos los tipos de sensores disponibles y detecta las variaciones en el estado del mundo.

GUI: La librería de *GUI* (*Graphical User Interface*) hace uso de la arquitectura *Qt* para crear aplicaciones gráficas que permitan a los usuarios interactuar con la simulación. Es posible modificar el escenario añadiendo, modificando o bien eliminando modelos de este. A mayores, también existen herramientas para visualizar la información de los sensores.

A mayores, Gazebo provee dos ejecutables diferentes que se comunican a través de la librería de comunicación:

- el **servidor**, ***gzserver***, empleado para simular la física, los sensores y los renderizados.
- el **cliente**, ***gzclient***, que proporciona la interfaz gráfica necesaria para visualizar e interactuar con la simulación.

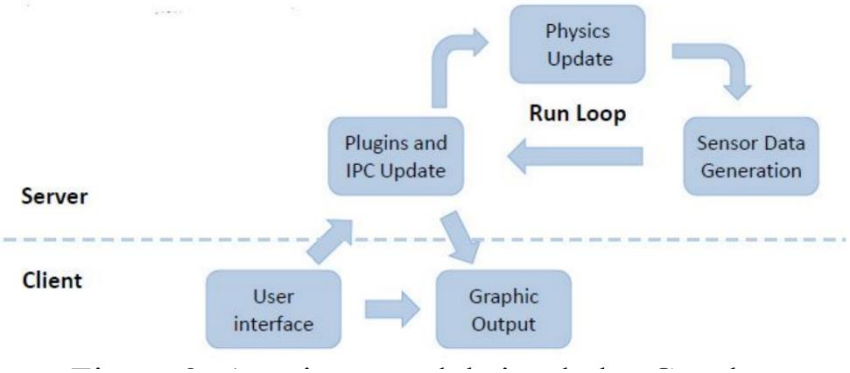


Figura 4-23. Arquitectura del simulador de Gazebo.

Componentes de Gazebo

En la ejecución de toda simulación de Gazebo intervienen una serie de componentes, los mundos y los modelos, que se describen a continuación.

Archivos mundo (.world): Contienen todos los elementos de la simulación, como los modelos de robots, luces, sensores y otros objetos estáticos.

Archivos de modelo: Representan elementos individuales como robots, objetos estáticos, obstáculos, etc. permitiendo simplificar los archivos mundo pudiendo ser reutilizados e incluidos en estos.

Ambos archivos están escritos en formato *SDF* (*Spatial Data File*), un formato de archivos *XML* desarrollado para describir los elementos de un modelo de Gazebo. A pesar de tener una estructura muy similar, este formato de archivos pretende cubrir algunas de las deficiencias del formato *URDF*, proporcionando una definición mucho más amplia de los elementos de la simulación. Los archivos *URDF* solo pueden especificar las propiedades cinemáticas y dinámicas de un único robot, pero no permite definir otros aspectos como la posición del robot en el mundo o la fricción superficial de los cuerpos. Por su parte, *SDF* es descripción completa del entorno 3D, desde el mundo hasta el robot.

No obstante, *URDF* es el lenguaje principal empleado por ROS para la definición de robots, por lo que deberá utilizarse este formato si se desean implementar los controladores de ROS. Por suerte, Gazebo realiza una conversión del formato *URDF* al *SDF* de manera automática siempre y cuando todos los elementos <*link*> tengan un elemento <*inertial*> configurado. Además, para definir todos aquellos atributos incluidos en el formato *SDF*, pero no en el *URDF*, existe el elemento <*gazebo*>. El elemento <*gazebo*> es una extensión de *URDF* que permite especificar atributos de *SDF* no presentes en *URDF*. Incluir o no dicho elemento es opcional pues, de no estar presente, Gazebo asignará los valores de dichos atributos por defecto. El elemento <*gazebo*> puede incluirse en elementos como <*robot*>, <*link*> y <*joint*>, para cada uno de los cuales se incluyen distintos atributos [39].

Integración ROS con Gazebo

Como se comentó en apartado anterior sobre la descripción del software de ROS, para conseguir una integración completa entre este y Gazebo, son necesarios un conjunto de paquetes llamados *gazebo_ros_pkgs*. Estos proporcionan las interfaces necesarias para simular el robot en Gazebo mediante el uso de estructuras de datos, como mensajes, servicios o mecanismos de cálculos dinámicos de ROS [40].

En el diagrama de la **Figura 4-24** se muestra una descripción general del paquete *gazebo_ros_pkgs*.

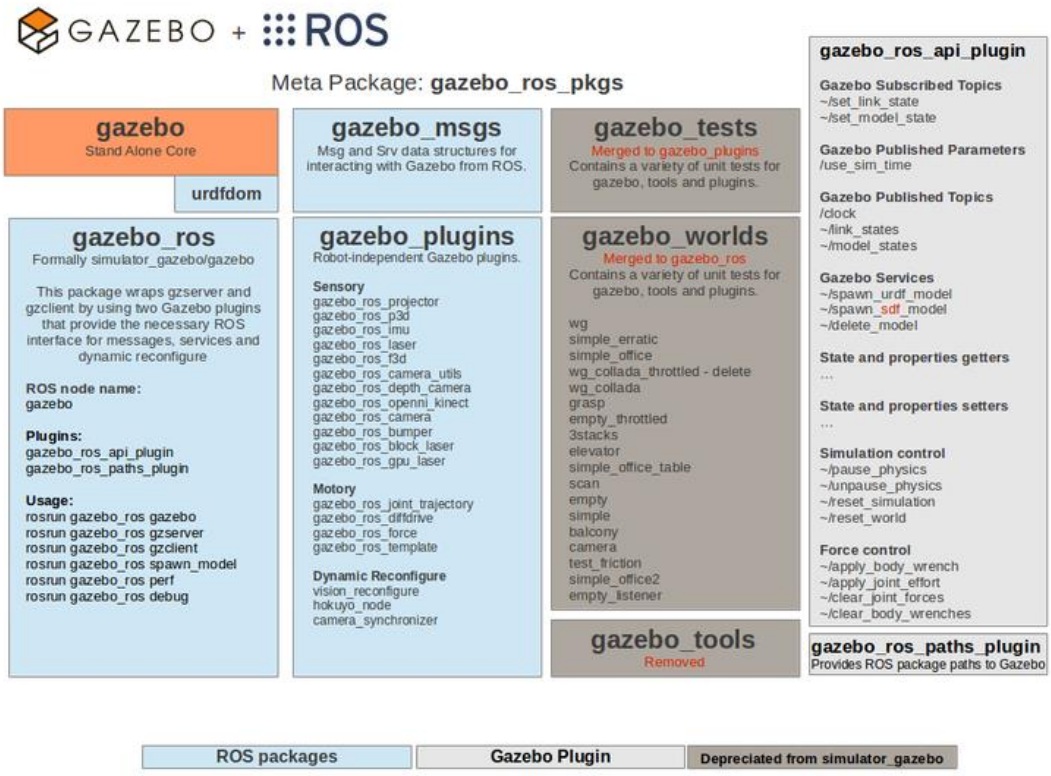


Figura 4-24, Metapaquete gazebo_ros_pkgs.

Junto con este, también será necesario instalar el paquete *gazebo_ros_control*, un plugin necesario para comunicar los modelos de Gazebo en formato *URDF* con la interfaz de ROS, que debe ser incluido en los archivos *URDF* como un elemento *<gazebo>*. Una vez generados los controladores de ROS, para que estos sean capaces de comunicarse con los actuadores del modelo, deberá indicársele a Gazebo cuales son las articulaciones que se desean controlar. Para ello, se especificará un elemento *<transmission>* por cada articulación, el cual conectará cada una de ellas con su respectivo controlador a través de las diferentes interfaces de hardware de ROS (Effort Joint Interface, Velocity Joint Interface y Position Joint Interface) [41].

En la **Figura 4-25** se muestra un diagrama que particulariza lo comentado sobre *ros_control* a una simulación de Gazebo:

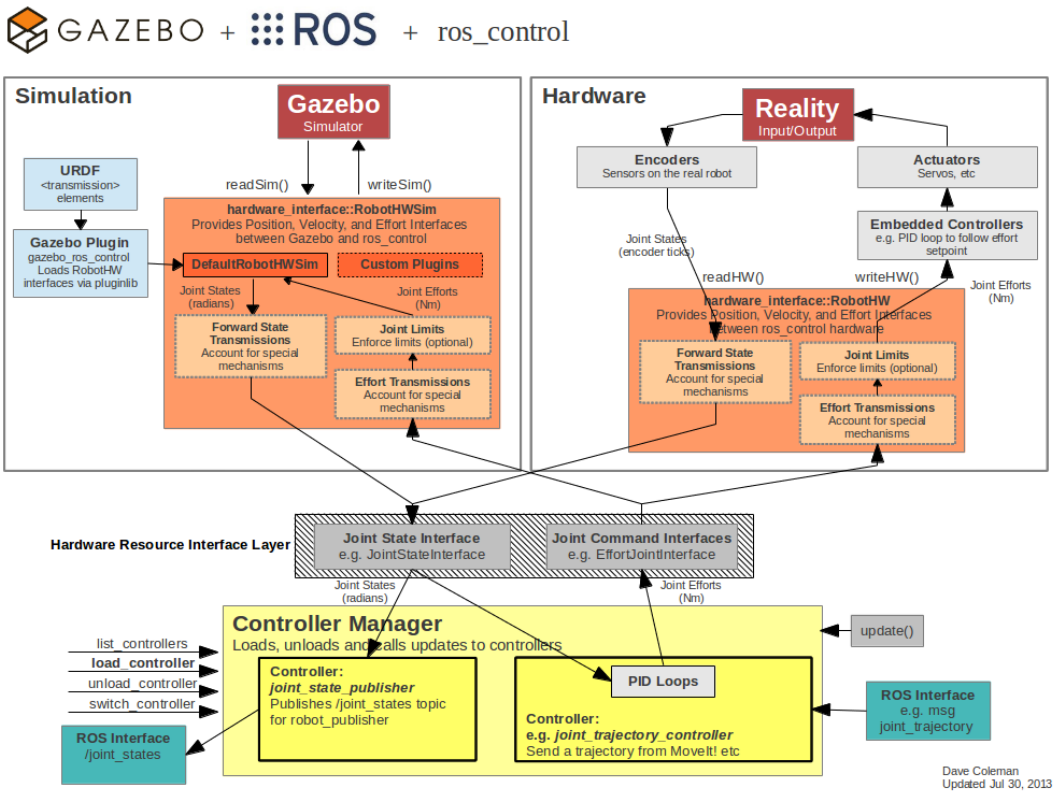


Figura 4-25. Diagrama de comunicación entre ROS y Gazebo con ros_control.

Con esto, queda completamente explicada la estructura de comunicación entre ROS y Gazebo para el control de robots. Esta estructura permite al usuario tanto gobernar el funcionamiento del robot como obtener información sobre sus articulaciones en cada instante, haciendo uso de los mecanismos de comunicación expuestos en el apartado de ROS.

4.3.3 SolidWorks y Blender

SolidWorks es un software de *CAD* para modelado mecánico 2D y 3D. Es muy utilizado para la elaboración de modelos 3D de piezas y conjuntos, pues permite generar las superficies, ensambles, uniones y ejes de referencia necesarios para representar fielmente el brazo robótico. Además, ofrece la posibilidad de extraer de los modelos diseñados información como planos técnicos o diversas propiedades físicas.

Para este proyecto, la principal ventaja que tiene SolidWorks frente a otros programas es la posibilidad de exportar el archivo *CAD* del modelo robótico a un paquete ROS mediante una herramienta denominada *SW2URDF*. Esta herramienta genera un paquete ROS en los que destacan los siguientes directorios:

- ***urdf***: En esta carpeta se encuentran los archivos *URDF* correspondientes a cada una de las piezas o eslabones que conforman el modelo en SolidWorks. Estos archivos son generados por la herramienta SW2URDF a partir de los cálculos de las propiedades físicas realizados por SolidWorks sobre sus modelos.
- ***meshes***: Contiene los archivos *STL* de cada eslabón empleados tanto para la representación visual como para los modelos de colisión.

No obstante, para poder utilizar la herramienta *SW2URDF*, es necesario definir antes una serie de parámetros dentro del modelo de SolidWorks, enumerados a continuación, pues, sin una configuración previa, no será posible para la herramienta crear una relación entre las distintas piezas:

- ***Orígenes y Sistemas de Referencia***: Por cada eslabón deberá colocarse un sistema de referencia adecuado a lo establecido por el Método de Denavit-Hartenberg, con la diferencia de que a cada eslabón *j* le corresponderá el sistema de referencia *j-1* en lugar del sistema *j*, debido a las referencias del formato URDF.
- ***Ejes***: Sobre cada sistema de referencia *j-1* se debe establecer un eje de manera que cada eslabón *j* pueda moverse respecto a dicho eje.
- ***Tipos de Articulación***: Para el brazo robótico TITANROB solo habrá articulaciones de tipo rotoide.

Durante la generación del paquete de ROS, los archivos *STL* de cada eslabón se exportan con sus orígenes ubicados en el origen global del modelo y no en el origen de su correspondiente sistema de referencia, de manera que al añadirlos al archivo *URDF* estos aparecen completamente desplazados. Para resolver este problema, se hará uso de Blender, otro software de modelado que, además de reubicar el origen de cada *STL* a la posición deseada, permite transformar estos a archivos *Collada* (*.dae*), compatibles con ROS y Gazebo y que permiten agregar numerosos efectos visuales no disponibles en los *STL*.

5 DISEÑO DEL SIMULADOR

En el siguiente capítulo se mostrará el diseño completo del simulador virtual desarrollado para el brazo TITANROB. En base a lo explicado en el capítulo anterior, se hará una descripción completa del modelo 3D en SolidWorks y del modelado matemático, donde se estudiarán la cinemática y dinámica del brazo robótico. Posteriormente, se describirán los diferentes paquetes ROS creados para la descripción, lanzamiento y control del submarino en Gazebo.

5.1 Diseño del modelo 3D del brazo

En lo referente al modelo 3D de este proyecto, se ha utilizado el modelo del brazo TITANROB E500 incluido entre los modelos de SolidWorks del submarino KAI elaborados por G. Saavedra Soto durante su Trabajo de Fin de Grado [4].

Este modelo, presente en la **Figura 5-1**, posee una estructura 3D que representa los principales elementos del brazo TITANROB, cumpliendo en todo momento con los requerimientos dimensionales, físicos y funcionales que posee el brazo real. El modelo representa los cuatro eslabones del robot, correspondientes al hombro (L1), el antebrazo (L2), el brazo (L3) y la muñeca (L4), las pinzas de la herramienta (PE y PI), sujetas a la muñeca, y la base (B0).

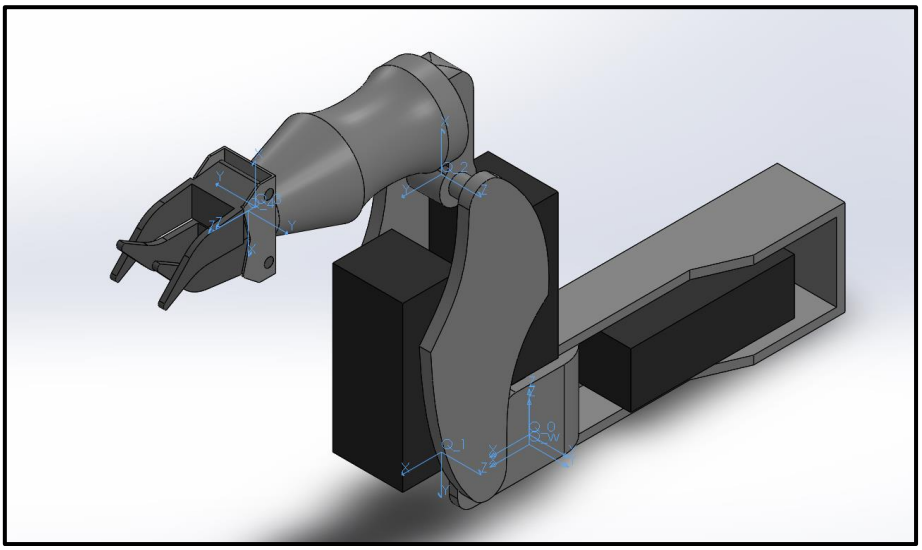


Figura 5-1. Modelo 3D del brazo TITANROB.

Todos los ensamblajes son articulaciones de tipo revolutiva, a excepción de la unión entre brazo y la muñeca, que es de tipo continua. En la **Tabla 5-1** se muestran los límites de cada articulación, obtenidos experimentalmente, junto con las velocidades articulares máximas:

ARTICULACIÓN	1	2	3	4	PINZA

LÍMITE SUPERIOR (°)	96	90	57	-	45
LÍMITE SUPERIOR (RAD)	1,676	1,571	0,995	-	0,785
LÍMITE INFERIOR (°)	-32	0	-60	-	0
LÍMITE INFERIOR (RAD)	-0,559	0,000	-1,047	-	0,000
VELOCIDAD MÁXIMA (RAD/S)	0,116	0,082	0,113	3,860	0,238

Tabla 5-1. Límites y velocidades articulares máximas brazo TITANROB.

5.2 Diseño del modelado matemático del brazo

Con el modelo 3D de la planta del brazo TITAN en SolidWorks, es posible obtener los valores numéricos necesarios para la definición del modelado matemático que permitirá describir el comportamiento del brazo robótico en el espacio. Para la obtención de este modelo, se realiza un estudio de los modelos cinemático y dinámico del robot, haciendo uso de los valores obtenidos del modelo 3D y de los fundamentos teóricos expuestos en el capítulo anterior.

5.2.1 Modelo cinemático

Como ya se ha comentado, mediante el estudio del problema cinemático es posible establecer una relación entre las coordenadas articulares del robot y las coordenadas de su efector final.

Para la obtención del modelo cinemático, en primer lugar, se realiza un análisis de la morfología del brazo robótico. Deben identificarse y enumerarse todos y cada uno de los eslabones que componen al robot, a la vez que se determina el tipo de articulación que une cada par de eslabones. Como se puede comprobar en la **Figura 5-2**, el brazo TITANROB está formado por cuatro eslabones unidos entre sí por cuatro articulaciones rotoides. A mayores, en la muñeca, pueden apreciarse otros dos eslabones, correspondientes a las pinzas de la herramienta.

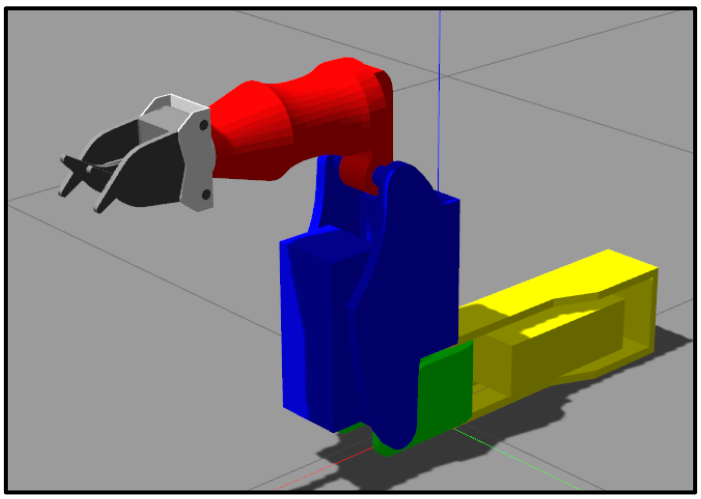


Figura 5-2. Eslabones del brazo TITAN.

Con todos los eslabones y las articulaciones numerados e identificados, se procede a localizar los sistema de referencia correspondientes a cada una de las cuatro

articulaciones, según lo establecido por el método de D-H. Una vez ubicados los sistemas de referencia, mostrados en la **Figura 5-3**, y a partir de las mediadas del brazo robótico, presentes en los planos incluidos en el **Anexo 1: Planos del Brazo TITANROB E500**, deben obtenerse los cuatro parámetro dimensionales **(θ_j, d_j, a_j y α_j)** necesarios para el algoritmo de D-H, con los que se construye la **Tabla 5-2**.

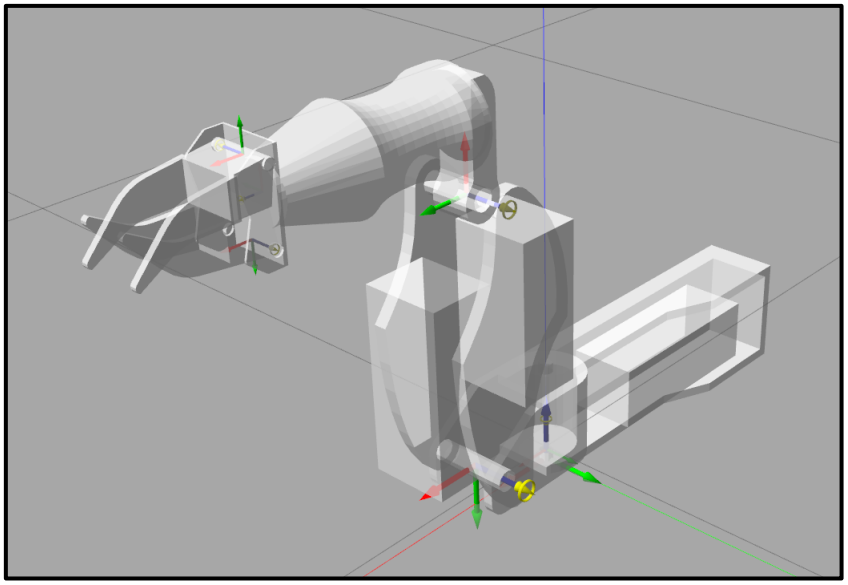


Figura 5-3. Sistemas de Coordenadas de D-H para el brazo TITANROB E500.

PARÁMETRO D-H	ESLABONES DEL ROBOT				
	Base	1 (link_1)	2 (link_2)	3 (link_3)	4 (link_4)
θ_j (°)	-	0,00	-90,00	0,00	-90,00
d_j (mm)	0,0112	37,87	0,00	0,00	244,29
a_j (mm)	-	115,44	315,84	85,50	0,00
α_j (°)	-	-90,00	0,00	-90,00	0,00

Tabla 5-2. Parámetros de D-H para el brazo TITANROB E500.

Ya obtenidos los cuatro parámetros de cada eslabón, se calculan las correspondientes matrices de transformación. Finalmente, la relación entre la base y el efector final será el resultado de la combinación de estas transformaciones (**Ec. 4-14**). Para la posición de origen del brazo, esta relación vendrá dada por la matriz ***${}^0T_4$***:

$$ {}^0_4T = \begin{bmatrix} 0 & 0 & 1 & 359.73.7 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 401.3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad (Ec.\ 5.1) $$

Con ello, queda definido el modelo cinemático del brazo TITANROB para este proyecto, que posteriormente será validado en el capítulo 7.

5.2.2 Modelo dinámico

De manera similar a lo que ocurría con el modelo cinemático, el modelo dinámico del brazo robótico permite establecer una relación entre los pares ejercidos sobre sus

articulaciones y la posición, velocidad y aceleración que estas adquieren en cada instante.

El estudio del modelo dinámico requiere incorporar los parámetros dinámicos del brazo real: masas, centros de gravedad o momentos de inercia. Estos pueden ser obtenidos en SolidWorks mediante el cálculo de las propiedades físicas del modelo 3D ya elaborado. Todos estos datos pueden compactarse junto con los valores del modelo cinemático en una matriz de 18 filas y 5 columnas, como la de la **Tabla 5-3**, que define completamente al robot.

Parámetros Modelo Dinámico			**ESLABONES DEL ROBOT**				
Parámetro	**Uds**	**Descripción**	**Base**	**1**	**2**	**3**	**4**
θ_j	°	Parámetros Denavit-Hatenberg	-	0,00	-90,00	0,00	-90,00
d_j (mm)	m		-	37,87	0,00	0,00	244,29
a_j (mm)	m		-	115,44	315,84	85,50	0,00
α_j (°)	°		-	-90,00	0,00	-90,00	0,00
m (kg)	kg	masa	6,1853	1,413	10,827	5,003	0,157
rx (m)	m	Centro de masas	-0,19298	0,025	-0,001	0,082	0,000
ry (m)	m		0,0075955	0,000	-0,154	0,106	0,000
rz (m)	m		0,0625	0,049	0,000	0,000	0,012
Ixx	kg·m^2	Inercias respecto del eje de rotación (10^4)	121,840	31,914	715,050	285,180	1,049
Ixy	kg·m^2		9,022	0,000	-117,140	-20,223	0,000
Ixz	kg·m^2		0,000	2,670	0,000	0,000	0,000
Iyy	kg·m^2		562,360	47,943	520,680	69,557	1,319
Iyz	kg·m^2		0,000	0,000	0,000	0,000	0,000
Izz	kg·m^2		523,830	48,415	960,600	299,838	1,940
Q lim	°	Límites de su articulación	-	96	90	57	-
Q lim	°		-	-32	0	-60	-
V. max.	rad/s		-	0,116	0,082	0,113	3,860

Tabla 5-3. Parámetros de D-H y dinámicos del brazo TITANROB E500.

Todos estos datos son utilizados por SolidWorks para generar el archivo en formato URDF que describirá la estructura del brazo robótico. Posteriormente, Gazebo utilizará la información contenida en este *URDF* para calcular, mediante sus librerías de física, los esfuerzos ejercidos sobre las articulaciones. A partir de estos esfuerzos, Gazebo podrá realizar la dinámica directa del brazo y así obtener el estado de sus articulaciones en cada instante. Mediante las librerías de renderizado, Gazebo transforma toda la información para su representación visual en la *GUI*.

La ecuación utilizada por Gazebo para modelar la dinámica directa del robot será la inversa de la **Ec. 4-33**, referida a los pares desarrollados en sus articulaciones. Esta función, $G_R(s)$, representa la planta del robot simulado en Gazebo, que toma como señal de entrada los pares articulares y obtiene como salida las aceleraciones articulares (y en consecuencia las velocidades y posiciones). Para un control de la velocidad de las articulaciones, el sistema de control equivalente se ajustaría al de la **Figura 5-4**, equivalente al de la **Figura 4-19**:

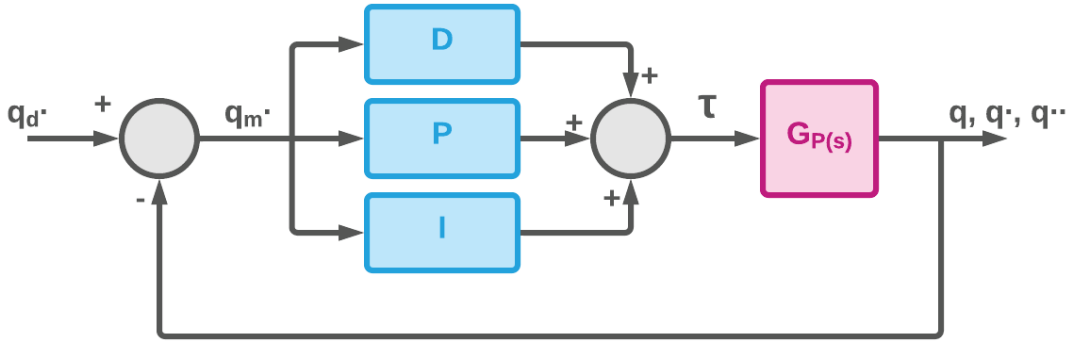


Figura 5-4. Sistema de control del brazo TITANROB.

En **capítulos posteriores**, se estudiará la posibilidad de anular los efectos del término relativo a la cargas gravitatorias, $G(q)$, sobre las articulaciones, dependiendo de si los motores del brazo real son o no capaces de mantener unas velocidades de giro constantes. Para que estas velocidades no varíen, los motores deben adaptar continuamente el par ejercido sobre las articulaciones en función de las cargas gravitatorias, pues estas varían con la posición y son las de mayor importancia en la dinámica del robot. Por tanto, se darán dos posibles situaciones:

I. **Las velocidades se mantienen constantes**. El efecto de la gravedad sobre la velocidad de giro de las articulaciones es despreciable, dando a entender que los actuadores disponen de un mecanismo de compensación que adapta el par ejercido para mantener la velocidad. En tal caso, el sistema de control del modelo podría ajustarse al de la **Figura 5-5**:

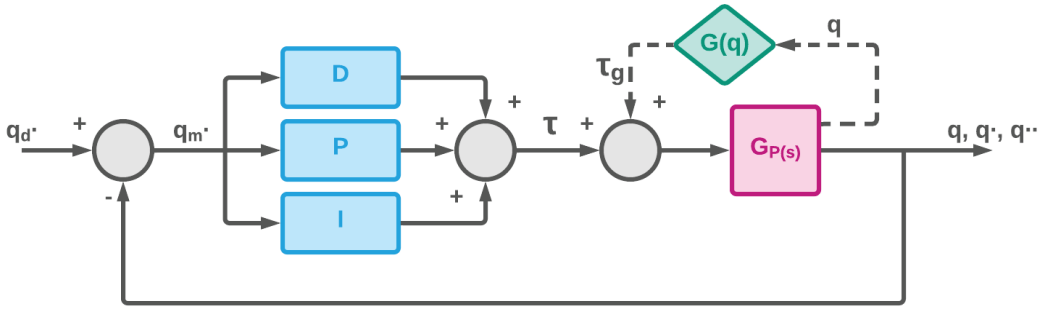


Figura 5-5. Sistema de control del brazo TITANROB.

II. **Las velocidades no se mantienen constantes.** Los efectos de la gravedad provocan variaciones apreciables en la velocidad de giro de las articulaciones del robot. En este caso, será necesario estudiar otras soluciones, como introducir fuerzas de rozamiento en los actuadores o reducir la gravedad para que el robot pueda mantenerse en sus posiciones en reposo.

5.3 Diseño de los paquetes ROS

En esta sección se describirá detalladamente el sistema de archivos construido para ejecutar el simulador del brazo TITANROB.

El sistema de archivos del simulador está compuesto por tres paquetes ROS, los cuales contienen todos los elementos necesarios para la correcta implementación e integración del modelo del brazo robótico, sus controladores y la aplicación joystick para controlar sus movimientos. Todos estos elementos están construidos a partir de archivos escritos en diferentes formatos, procedentes de plantillas modificadas según las necesidades del simulador. Estos archivos conforman el conjunto de nodos y parámetros ROS que se comunican entre sí para hacer posible el funcionamiento del simulador.

Para una mejor comprensión de su funcionamiento, la **Figura 5-6** contiene un diagrama que muestra gráficamente el diseño de la estructura del simulador, paquete a paquete.

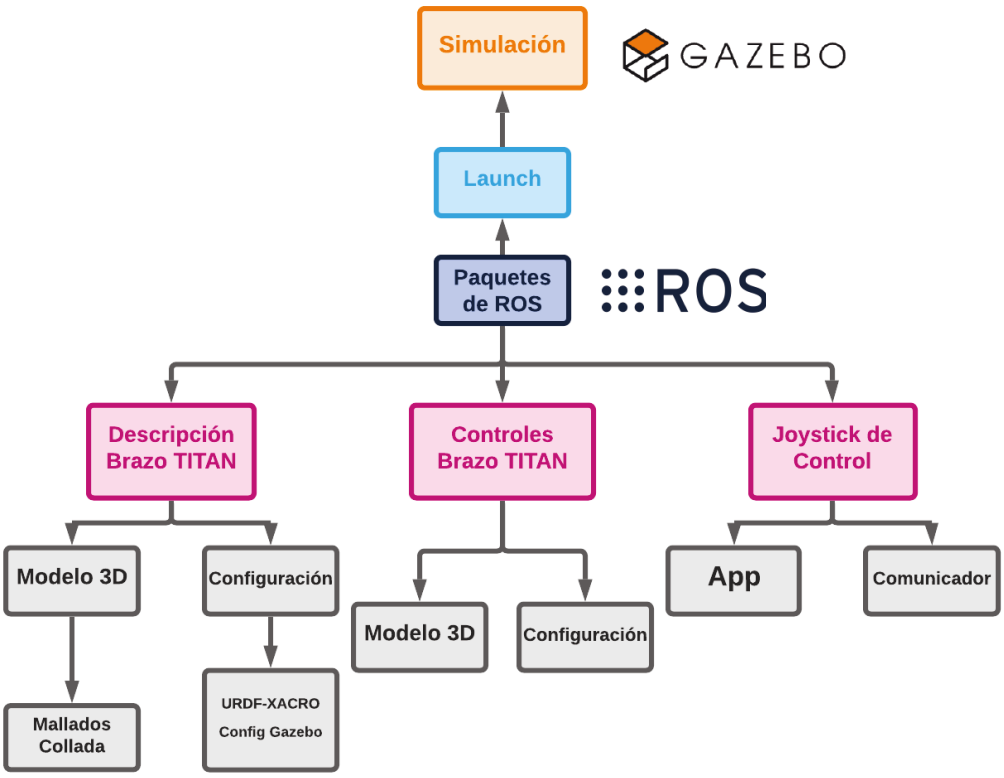


Figura 5-6. Diagrama de diseño del simulador.

Como se puede apreciar, hay tres paquetes ROS: uno para la descripción del modelo del brazo, otro para la configuración de los controles y un último para la generación de un joystick virtual, que dará órdenes a los controladores del brazo. Los tres paquetes son ejecutados por un *launch*, que establece la comunicación entre los nodos y obtiene los parámetros requeridos por el simulador para su lanzamiento. Haciendo uso de los paquetes de *gazebo_ros_pkgs*, se logra la integración entre ROS y Gazebo, dando como resultado un simulador funcional del brazo robótico en la plataforma Gazebo y controlado por ROS.

5.3.1 Diseño del paquete de descripción del brazo

Este paquete, llamado ***bt_description***, contiene todos los archivos y ficheros necesarios para la descripción y ejecución del modelo del brazo robótico TITANROB E500. Consta de tres directorios, descritos a continuación: ***urdf***, ***meshes*** y ***launch***.

Directorio *urdf*

Esta carpeta contiene toda la información relativa al modelo del brazo robótico, como son la definición de su estructura y las propiedades de sus elementos:

- **brazo_titan.xacro**

  Este archivo, en formato *XACRO*, contiene toda la información necesaria para la definición del modelo del robot en el software de ROS. Es uno de los archivos más importantes pues, en sí mismo, establece la configuración y morfología del brazo robótico. Se trata del archivo *URDF* generado por SolidWorks con SW2URDF, una vez ha sido ajustado a *XACRO*.

  Contiene los datos para definir los cuatro eslabones que componen al robot, nombrados *link_1*, *link_2*, *link_3* y *link_4*, la base, *base_link*, y las pinzas del efector final, *pinza_ext* y *pinza_int*. Para cada eslabón, incluye un bloque para las propiedades inerciales, <*inertial*>, otro para la representación visual, <*visual*>, y un último para los cuerpos de colisión, <*collision*>:

```
<link name="___">
  <inertial>
  <origin xyz=". . ." rpy=". . ." />
  <mass value=". " />
  <inertia ixx=". " iyy=". " izz=". "
   ixy=". " ixz=". "
   iyz=". " />
  </inertial>
  <visual>
    <origin xyz=". . ." rpy=". . ." />
    <geometry>
    <mesh filename="package://____" />
    </geometry>
  </visual>
  <collision>
    <origin xyz="0 0 0" rpy="0 0 0" />
    <geometry>
      <mesh filename="package://____" />
    </geometry>
  </collision>
</link>
```

  También define las cuatro articulaciones que une cada par de eslabones, identificadas por *joint_1*, *joint_2*, *joint_3* y *joint_4*, así como las de las pinzas, *joint_pext* y *joint_int*, especificando las relaciones padre-hijo entre estos:

```
<joint name="___" type="revolute">
  <origin xyz=". . ." rpy=". . ." />
  <parent link="___" />
  <child link="___" />
  <axis xyz=". . ." />
  <limit
   lower=". "
   upper=". "
   effort=". "
   velocity=". " />
</joint>
```

Por último, junto con las articulaciones se incorporan las transmisiones, denominadas *trans_1*, *trans_2*, *trans_3* y *trans_4*, que modelan los actuadores de cada articulación y generan las interfaces ROS para conectarlos con su correspondiente controlador:

```
<transmission name="___">
  <type>transmission_interface/SimpleTransmission</type
  >
  <actuator name="____">
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
  <joint name="____">
    <hardwareInterface>hardware_interface/EffortJointIn
    terface</hardwareInterface>
  </joint>
</transmission>
```

Desde esta archivo se hace llamada a otros elementos del fichero, como las propiedades de Gazebo, que se explicarán a continuación, o los mallados, del directorio *Meshes*.

- **brazo_titan.gazebo**

  En este archivo se recogen todas aquellos atributos que no pueden ser incluidos en el formato *URDF*, como por ejemplo el plugin *gazebo_ros_control* para comunicar el modelo de *brazo_titan.xacro* con Gazebo:

```
<gazebo>
  <plugin    name="gazebo_ros_control"    filename    =
  "libgazebo_ros_control.so">
    <robotNamespace>/rrbot</robotNamespace>
    <robotSimType>gazebo_ros_control/DefaultRobotHWSim<
    /robotSimType>
  </plugin>
</gazebo>
```

  Como atributo opcional, puede asignarse a colores y materiales a cada eslabón:

```
<gazebo reference="___">
  <material>Gazebo/___</material>
</gazebo>
```

  Estos elementos pueden ser incluidos directamente en el archivo *brazo_titan.xacro*, pero, para una mejor organización, se decidió incluirlos en un fichero aparte.

Directorio *meshes*

En esta carpeta se encuentran los archivos de mallado en formato *Collada* del brazo TITANROB, obtenidos a partir de los archivos STL generados con la herramienta SW2URDF. Todos ellos fueron modificados en Blender para el ajuste de su origen y orientación, y, posteriormente, exportados a formato *Collada*.

- ***base_link.dae***: Archivo de mallado de la base del robot.
- ***link_1.dae***: Archivo del eslabón 1 del robot, correspondiente al eslabón del hombro.

- ***link_2.dae***: Archivo del eslabón 2 del robot, correspondiente al eslabón del brazo.
- ***link_3.dae***: Archivo del eslabón 3 del robot, correspondiente al eslabón del antebrazo.
- ***link_4.dae***: Archivo del eslabón 4 del robot, correspondiente al eslabón de la muñeca.
- ***pinza_ext.dae***: Archivo de la pinza exterior del robot, correspondiente a la pinza de 56 mm de ancho.
- ***pinza_int.dae***: Archivo de la pinza interior del robot, correspondiente a la pinza de 40 mm de ancho.

Directorio *launch*

Esta carpeta contiene únicamente el archivo de lanzamiento, ***brazo_titan.launch***. Dicho archivo es necesario para generar el modelo del brazo TITANROB, definido en el archivo *brazo_titan.xacro*, sobre la plataforma de Gazebo junto con los diferentes controladores de las articulaciones del robot.

En primer lugar, el archivo carga los argumentos necesarios y, a continuación, ejecuta un mundo vacío de Gazebo (*empty_world.launch*), localizado en el paquete *gazebo_ros*. Es necesario indicar donde se situará el brazo robótico en el mundo y las posición de sus articulaciones en el momento de la inicialización:

```
<arg name="x" default="0"/>
<arg name="y" default="0"/>
<arg name="z" default="0"/>
<arg name="j1" default="0.0" />
<arg name="j2" default="0.0" />
<arg name="j3" default="0.0" />
<arg name="j4" default="0.0" />
```

Para generar el modelo dentro del mundo, se carga el archivo URDF en el servidor de parámetros y se ejecuta el nodo *spawn_model*, también localizado en el paquete *gazebo_ros*. Para encontrar el URDF, ubicado en este paquete (*bt_description*), se emplea la herramienta *ROS find*:

```
<param name="robot_description"
  command="$(find     xacro)/xacro     --inorder     '$(find
  bt_description) /urdf/brazo_titan.xacro'"/>

<node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model"
respawn="false" output="screen"
  args="-urdf -model bt -param robot_description
  -x $(arg x) -y $(arg y) -z $(arg z)"/>
```

Por último, se ejecutan los controles. Para ello, se incluyen su correspondiente launch ubicado en el paquete *bt_control*:

```
<include file="$(find bt_control)/launch/bt_control.launch"/>
```

5.3.2 Diseño del paquete de del brazo

El paquete ***bt_control*** contiene todos los archivos necesarios para la implementación y el lanzamiento de los controladores del robot, a través del *Controller Manager*, y, por otro lado, establece las conexiones con las interfaces ROS creadas por las transmisiones. Dispone de dos directorios: ***config***, con los archivos que describen los controladores, y ***launch***, con el archivo de ejecución de los controles.

Directorio *config*

Esta carpeta contiene los archivos *yaml* que especifican la configuración de los diferentes controladores que se desean ejecutar.

- **bt_joint_state.yaml**

  Carga el controlador necesario para que el modelo del brazo generado en Gazebo publique el estado de todas sus articulaciones en sistema de ROS. Debe indicarse la frecuencia de publicación de datos:

```
joint_state_controller:
  type: "joint_state_controller/JointStateController"
  publish_rate: .
```

- **bt_joint_controllers.yaml**

  En este archivo se describen los controladores de cada una de las articulaciones, incluidas las de las pinzas. Como se comentó en el capítulo 3, existen tres tipos distintos según la señal enviada al simulador. Para un control por velocidad articular mediante el envío de señales de par motor, se emplea el controlador de tipo ***effort_controllers/JointVelocityController***. Es necesario establecer la articulación a controlar los valores de las ganancias proporcional, integral y derivativa del PID:

```
joint__controller:
  type: "effort_controllers/JointVelocityController"
  joint: _____
  pid: {p: . , i: . , d: . }
```

Directorio *launch*

En este directorio se encuentra el archivo de ejecución de los controles, ***bt_control.launch***, que da la orden al Controller Manager para inicializar y administrar los controladores de ROS, definidos en los archivos *yaml* anteriores.

En primer lugar, el *launch* carga la configuración de los controladores, especificados en los archivos *yaml,* dentro del servidos de parámetros de ROS:

```
<rosparam file="$(find brazo_titan)/config/bt_joints.yaml"
command="load" ns="/bt"/>

<rosparam file="$(find
brazo_titan)/config/bt_joint_controllers.yaml" command="load"
ns="/bt"/>
```

A continuación, el nodo *controller_spawner*, del paquete *controller_manager*, manda una petición de servicio al *Controller Manager* para iniciar los controladores anteriormente guardados en el servidor de parámetros:

```
<node name="controller_spawner" pkg="controller_manager"
type="spawner" respawn="false" output="screen" ns="/bt"
args="joint_state_controller
      joint1_controller
      joint2_controller
      joint3_controller
      joint4_controller
      joint_pext_controller
      joint_pint_controller "/>
```

Por último, se ejecutan los nodos *joy_control* y *joy_transfer*, del paquete *bt_joystick*, que inician el joystick virtual y lo conectan a las articulaciones del brazo, respectivamente:

```
<node name="joy_control" pkg="bt_joystick" type="joystick"/>
<node name="joy_transfer" pkg="bt_joystick" type="transfer"/>
```

Este archivo *launch* es el incluido dentro del archivo *brazo_titan.launch*, ejecutándose automáticamente con este.

5.3.3 Diseño del paquete del joystick virtual

En el paquete ***bt_joystick*** se encuentran todos los archivos necesarios para el funcionamiento y configuración del joystick virtual, así como para su conexión con los actuares del robot simulado. Destacan los siguientes:

- **joystick**

  Nodo desde el que se ejecuta la aplicación del joystick virtual**.**

- **main.qml**

  Archivo en formato *QML*[9] que define la estructura de la app. A partir de tres archivos de imagen, *background.png, thum1.png y thum2.png*, se crean tres objetos: la base, el joystick 1 y el joystick 2, superpuestos unos a otros mediante relaciones arbóreas padre-hijo.

  Cuando se modifica la posición de uno de los joysticks, la app publica las coordenadas x e y del hijo respecto al padre en un determinado topic mediante mensajes de tipo *PoseStamped*, de la librería *geometry_msgs.msg*. Estos mensajes son transformados por el archivo trasnfer.py.

  A modo de ejemplo, el archivo *main.qml* se encuentra recogido en el **Anexo 2: Código QML de configuración del joystick virtual**

- **transfer.py**

  Nodo que conecta la aplicación con cada una de las articulaciones. Se subscribe a los topics en los que publica la app y transforma los mensajes al formato adecuado para publicarlos en los topics de cada articulación.
  El código de este archivo se encuentra recogido en el

  Anexo 3: Código en Python del archivo *transfer.py*.

[9] **QML:** *Qt Meta Language*, es un lenguaje basado en JavaScript creado para el diseño aplicaciones orientadas a la interfaz de usuario.

6 . IMPLEMENTACIÓN DEL SIMULADOR

En este capítulo, se explicarán todos los pasos necesarios para la implementación completa del simulador desarrollado en el trabajo. En primer lugar, se indicarán las instalaciones necesarias para el correcto funcionamiento de los programas y componentes utilizados. Seguidamente, se expondrá detalladamente el método de exportación de los archivos de SolidWorks, desde su configuración en el programa, hasta su apertura en Gazebo. A continuación, se darán las instrucciones para construir los paquetes que conformarán el simulador. Finalmente, se ajustarán los controles para lograr el movimiento planteado en los objetivos del proyecto.

6.1 Instalaciones de Linux

Algunos de los programas más importantes para la utilización del simulador solo son compatibles con el sistema operativo Linux, como es el caso de ROS y Gazebo. Además, para estos dos programas se recomienda utilizar una determinada combinación de sus distribuciones, pues así es posible un acceso a las funcionalidades de ambos.

Así pues, en primer lugar, debe escogerse que combinación de programas se desea utilizar. Dado que el equipo utilizado dispone de ***Ubuntu 18.04 Bionic Beaver*** [42], la mejor combinación consiste en emplear ***ROS Melodic Morenia*** [43] junto con ***Gazebo 9.0*** [44].

6.1.1 Instalación de ROS

A continuación, se presentará el método de instalación de la distribución ROS Melodic Morenia, disponible en la página web oficial de ROS [45].

Configuración

Antes de empezar, deben configurarse los repositorios de Ubuntu en la ventana de *Software y actualizaciones*, de manera que las casillas *"universe"*, *"restricted"* y *"multiverse"* estén seleccionadas (**Figura 6-1**).

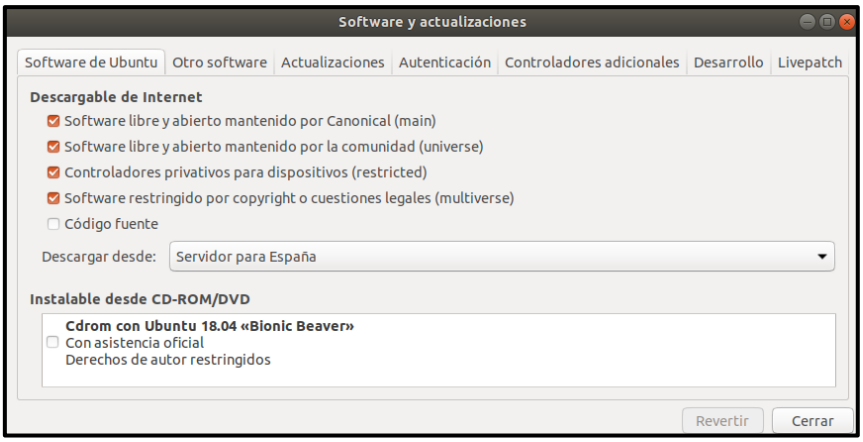


Figura 6-1. Configuración de Software y actualizaciones.

A continuación, en una *shell*[10] de Ubuntu, se configura los permisos para aceptar software procedente de los repositorios de ROS:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu
$(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-
latest.list'
```

Y se configuran los parámetros para la descarga de los paquetes:

```
$ sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' -
-recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Instalación y configuración del entorno

Una vez configurado Ubuntu, en primer lugar, se comprueba que todos los paquetes estén actualizados:

```
$ sudo apt update
```

A continuación, debe decidirse que librerías de ROS se desean instalar. Para el caso de este trabajo, se instalará la versión completa de escritorio, *Desktop-Full Install*:

```
$ sudo apt install ros-melodic-desktop-full
```

La instalación de ROS se iniciará y tardará unos minutos.

Una vez instalado ROS, se instalarán las herramientas y dependencias necesarias para la construcción, compilación y ejecución de paquetes ROS. Para ello, se introduce en la consola el siguiente comando:

Varias herramientas de ROS requieren una de dependencias no incluidas en el sistema. Para su instalación debe inicializarse *rosdep*. Para su instalación, se escribe el siguiente comando:

```
$ sudo apt install python-rosdep
```

Con *rosdep* ya instalado, este debe inicializarse para permitir la instalación de todas las dependencias necesarias:

Finalmente, para que el sistema encuentre todos los archivos necesarios para el proyecto, estos deberán estar incluidos en la variable de entorno, el *PATH*. Lo más recomendable es que el *bash* de ROS se agregue automáticamente al *PATH* cada vez que se abra una nueva terminal *shell* de Ubuntu:

```
$ echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc source
~/.bashrc
```

Sin embargo, si se prefiere agregar el *bash* al *PATH* en cada nueva terminal, puede escribirse lo siguiente:

```
$ source /opt/ros/melodic/setup.bash
```

Instalación de catkin

[10] **shell:** programa informático que provee una interfaz de usuario para acceder a todos los servicios de un sistema operativo.

Por lo general, el paquete catkin se instala automáticamente con ROS. No obstante, para evitar problemas, se instalará nuevamente con el siguiente comando:

```
$ sudo apt install ros-melodic-catkin
```

Instalación de paquetes ROS

Para instalar los paquetes *robot*, *ros_control* y *ros_controllers*, necesarios para implementar los controles del brazo robótico, se introduce el siguiente comando:

```
$ sudo apt install ros-melodic-robot ros-melodic-ros-control
ros-melodic-ros_controllers
```

En ocasiones, el paquete *joint_state_publisher*, del metapaquete *robot*, no se instala correctamente junto con este. Es por eso, como seguridad se vuelve a instalar de manera independiente:

```
$ sudo apt install ros-melodic-joint-state-publisher
```

Con ello, queda completamente instalado el entorno de trabaja de ROS.

6.1.2 Instalación de Gazebo

La distribución de *ROS Melodic Morenia* incluye por defecto la versión de *Gazebo 9.0*. Para comprobar que efectivamente se encuentra en el sistema, se ejecuta Gazebo con el siguiente comando homólogo:

```
$ gazebo
```

En el supuesto de no iniciarse, se procede a instalar la versión *9.0* de Gazebo manualmente desde la *shell*:

```
$ sudo apt install gazebo9
```

Y se comprueba nuevamente:

```
$ gazebo
```

Deberá aparecer una ventana como la de la **Figura 6-2**:

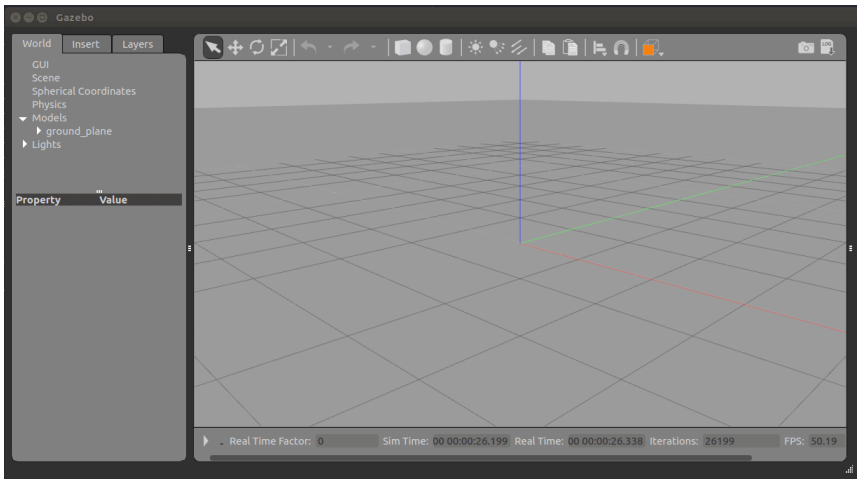


Figura 6-2. Interfaz de Gazebo

6.1.3 Conexión ROS-Gazebo

Para conseguir una correcta integración de ROS y Gazebo, se instalan los paquetes *gazebo_ros_pkgs* y *gazebo_ros_control*:

```
$ sudo apt install ros-melodic-gazebo-ros-pkgs ros-melodic-gazebo-ros-control
```

Para verificar entorno ROS-Gazebo está correctamente configurado, se ejecuta Gazebo mediante el comando *rosrun*, tras haber iniciado *roscore*:

```
$ roscore &
$ rosrun gazebo_ros gazebo
```

6.2 Importación de los archivos de SolidWorks a Gazebo

A lo largo de esta sección, se explicará detalladamente el proceso de obtención de los archivos de descripción del brazo TITANROB E500 a partir de los modelos 3D generados en SolidWorks. Se darán todas las instrucciones necesarias para la configuración, exportación y ajuste del modelo, desde la instalación de la herramienta SW2URDF, hasta la importación en Gazebo de los archivos obtenidos.

6.2.1 Configuración en SolidWorks

Antes de utilizar la herramienta de exportación de SolidWorks, para garantizar que la conversión de los archivos a *URDF* se realiza correctamente, es necesario definir sobre el modelo los *orígenes*, *ejes* y *sistemas de referencia* de cada una de las piezas que lo conforman. Estos atributos permitirán a la herramienta *SW2URDF* establecer las correctas relaciones jerárquicas que definirán los enlaces de unión entre las piezas.

Para la construcción de estos elementos, deberá abrirse cada pieza del ensamblaje por separado. Una vez abierta, en la pestaña “Operaciones”, se seleccionará la operación *“Geometría”*, que mostrará un desplegable con varias opciones (**Figura 6-3**).

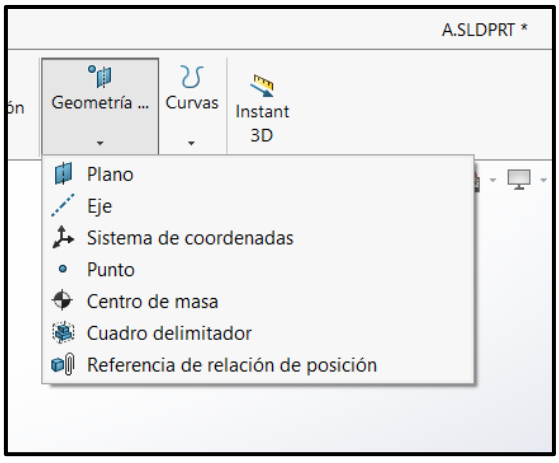


Figura 6-3. Desplegable operación Geometría de SolidWorks.

En primer lugar, en cada eslabón *j* se definirá un *punto* sobre la correspondiente articulación *j* (articulación que lo unirá con su respectivo eslabón hijo), que será el origen del sistema de referencia *j-1*, ***O_j-1***. A continuación, se especificará el eje de rotación de la articulación *j*, ***Z_j-1***, que establecerá el movimiento relativo del eslabón *j* respecto del eslabón *j-1*. Por último, sobre el origen *O_j-1* y el eje *Z_j-1* se construirá el sistema de referencia ***Q_j-1***.

El sistema de referencia del primer eslabón (la base del robot) será el sistema de referencia mundo o global, *Q_w*, que se situará sobre el eje de la articulación 1. Para este caso, el sistema de referencia de la base, *Q_0*, correspondiente al eslabón 1, no coincidirá con el global, sino que estará 11.2 mm encima de este, tal y como se aprecia en la **Figura 6-4**.

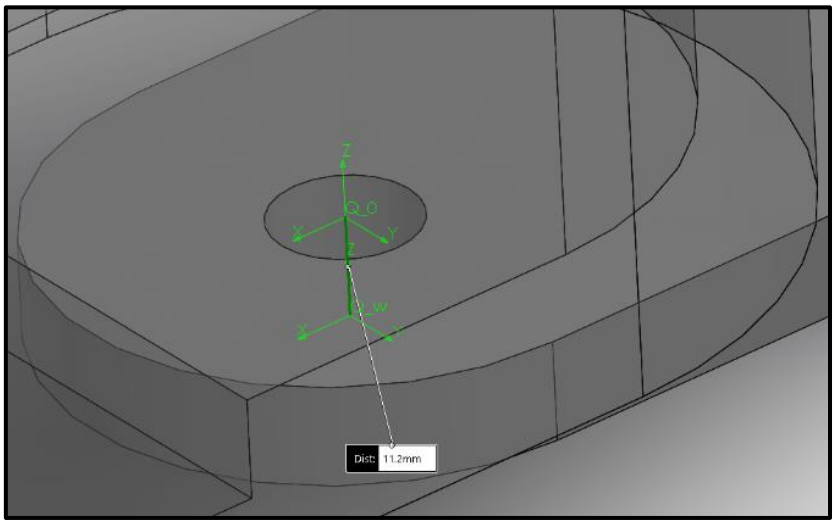


Figura 6-4. Sistema de referencia mundo (Q_w) y base (Q_0) sobre la base del robot.

La localización del resto de sistemas vendrá dada por los parámetros el modelo cinemático, presentes en la **Tabla 5-2**. Respecto a los sistemas de las pinzas, estos se definirán en el punto central de su eje de rotación. Una vez definidos todos los orígenes, ejes y sistemas de referencia, se obtendrá una estructura como la de la **Figura 6-5**:

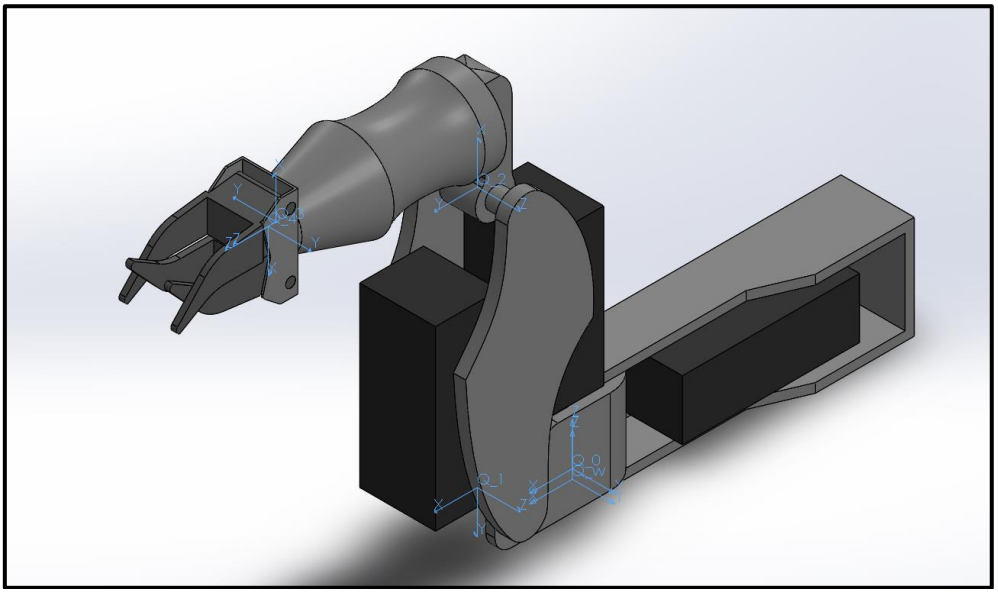


Figura 6-5. Sistemas de referencia definidos sobre el robot.

Pudiendo ahora sí comenzar a exportar los archivos a URDF.

6.2.2 Uso de SW2URDF

El exportador de SolidWorks a *URDF*, ***SW2URDF***, es un complemento de SolidWorks que permite la exportación completa de ensamblajes en formato de modelado 3D, como *STL* o *SLDASM*, a un formato *URDF*, que puede ser utilizado tanto en ROS como en Gazebo.

El paquete de *SW2URDF* está disponible para descargar en la página oficial de ROS como instalador precompilado [46]. Requiere una versión de SolidWorks 2018 o superior instalado en Windows 10 de 64 bits.

Una vez descargado e instalado, *SW2URDF* se encontrará en la pestaña superior "*Herramientas>File>Export as URDF"*, tal y como se ve en la **Figura 6-6**:

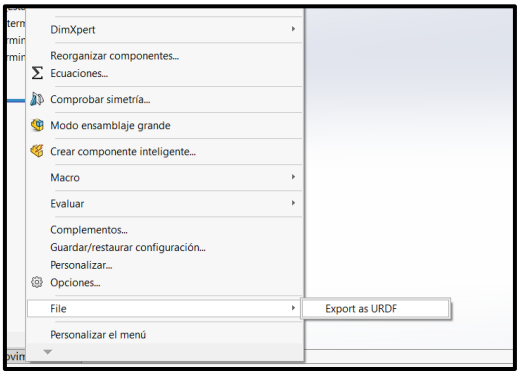


Figura 6-6. Herramienta SW2URDF en SolidWorks.

Al abrir la herramienta, aparecerá un gestor de propiedades en el margen izquierdo de la ventana de SolidWorks, que permitirá configurar el *URDF* pieza a pieza. Para cada eslabón ***j***, se deberá especificar un nombre, ***link_j***, un nombre para la articulación de unión con el *link* padre, ***joint_j***, un sistema de referencia, ***Q_j-1***, el eje de referencia, ***Z_j-1***, el tipo de articulación, los elementos del modelo asociados a ese eslabón y el número de hijos. Todo *URDF* deberá tener un único eslabón base, *base_link*. Una vez configurados todos los eslabones, se hace click en *"Preview and Export"*.

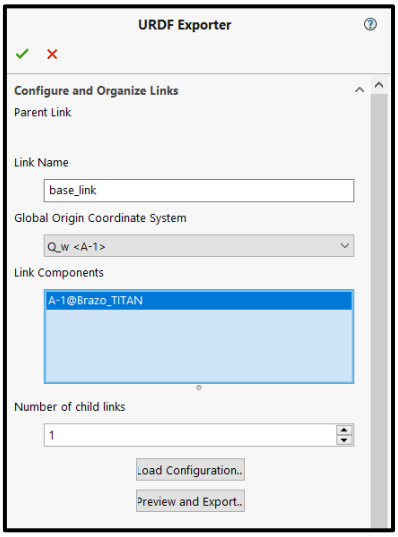


Figura 6-7. Gestor de propiedades de SW2URDF.

A continuación, se abrirá una nueva venta para configurar las propiedades de las articulaciones, como la de la **Figura 6-8**. Únicamente será necesario rellenar los campos del apartado "*Limit"* con los valores de los límites y velocidades máximas de las articulaciones de la **Tabla 5-1**. Finalmente, se hace click en *"Next"* y en *"Export URDF and Meshes"*. De esta forma, la herramienta generará un paquete ROS llamado ***bt_description*** con dos directorios importantes, uno para la descripción del brazo, ***urdf***, que contiene el archivo ***brazo_titan.urdf***, y otro para los mallados de las piezas, ***meshes***. Los archivos de este último son utilizados tanto para el elemento visual como de colisión del robot y deberán ser ajustados en Blender para redefinir sus orígenes.

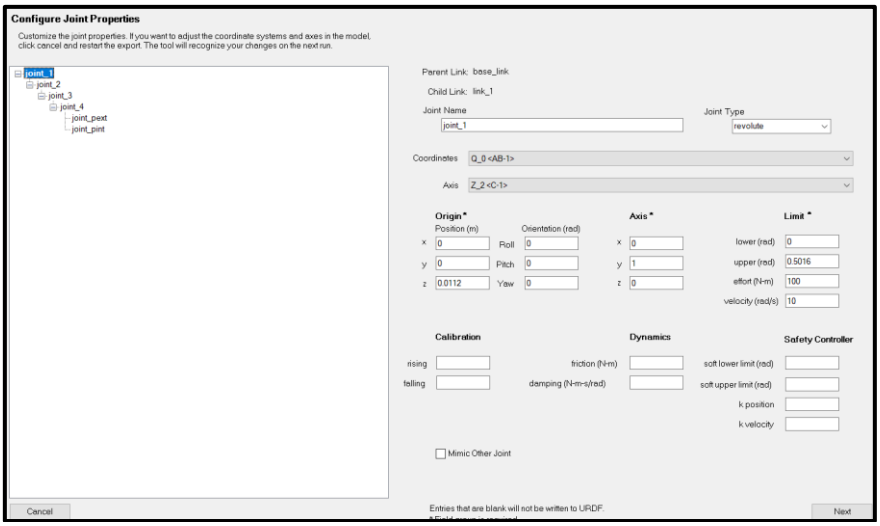


Figura 6-8. Configuración Joint Properties de SW2URDF.

6.2.3 Ajuste de los archivos de las mallas en Blender

Tal y como se comentó durante la introducción de SolidWorks en el capítulo 3, los mallados *STL* generados por la herramienta *SW2URDF* se exportan con sus orígenes ubicados en el origen global del modelo y no en el origen de su correspondiente sistema de referencia, de manera que al añadirlos al archivo *URDF* estos aparecen completamente desplazados.

Por ello, todos los mallados deben ser importados a Blender, donde se volverán a definir sus orígenes para que estos coincidan con los definidos anteriormente en SolidWorks (**Figura 6-9**). Para abrir el archivo, se va a la pestaña superior *"File>Import>Stl(.stl)"* y se selecciona el mallado a ajustar.

Con la pieza abierta, en la ventana *"Object Properties"* de la derecha, se introducen los valores *Location X*, *Y* y *Z* para que el origen deseado en la pieza coincida con el origen global de Blender. Una vez situada la pieza, en la pestaña *"Object>Snap"* se selecciona la opción *"Cursor to World Origin"* y a continuación, en *"Set Origin"* la opción *"Origin to 3D Cursor"*. De esta forma el origen de la pieza se fijará en el origen global de Blender. Por último, se reorienta la pieza y se exporta a formato *Collada* (*.dae*) y se guarda en la carpeta *meshes* de los *STL*.

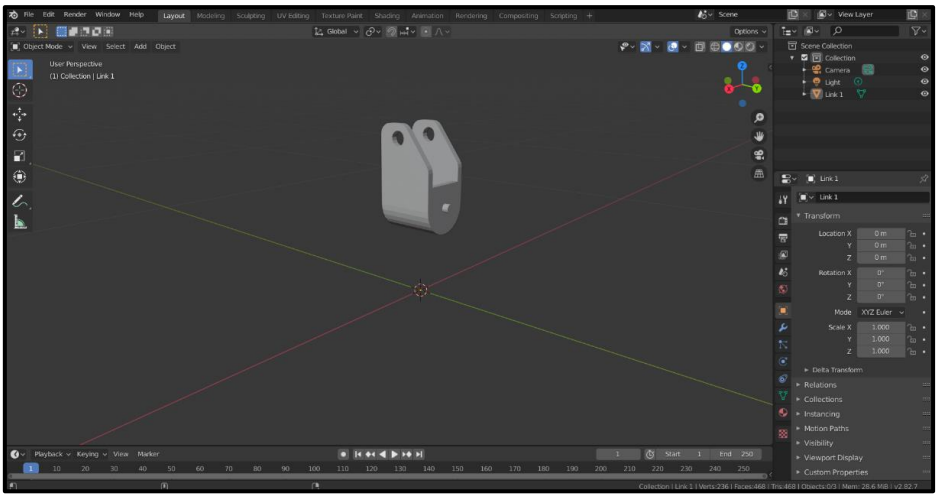


Figura 6-9. Interfaz de Blender con el mallado del eslabón 1 desubicado.

Una vez ajustadas todas las piezas, deben modificarse todos los elementos *<mesh>* del *URDF*. Basta con sustituir todas las extensiones *.stl* por *.dae*, como se ve en la **Figura 6-10**:

```
    link= base_link />
</joint>
<link
  name="link_1">
  <inertial>
    <origin
      xyz="0.024982 -5.4947E-19 0.049118"
      rpy="0 0 0" />
    <mass
      value="1.4127" />
    <inertia
      ixx="0.0031914"
      ixy="2.1342E-19"
      ixz="0.00026697"
      iyy="0.0047943"
      iyz="3.3862E-20"
      izz="0.0048415" />
  </inertial>
  <visual>
    <origin
      xyz="0 0 0"
      rpy="0 0 0" />
    <geometry>
      <mesh
        filename="package://bt_description/meshes/link_1.dae" />
    </geometry>
    <material
      name="">
      <color
        rgba="0.89804 0.91765 0.92941 1" />
    </material>
  </visual>
  <collision>
    <origin
      xyz="0 0 0"
      rpy="0 0 0" />
    <geometry>
      <mesh
        filename="package://bt_description/meshes/link_1.dae" />
    </geometry>
  </collision>
</link>
<joint
  name="joint_1"
```

Figura 6-10. Parte visual y de colisión de la base en el URDF.

6.2.4 Uso de URDF en Gazebo

Una vez listo el paquete, para añadir el modelo al repositorio de Gazebo, deberá crearse una carpeta, que se llamará *Brazo TITAN*. Dentro de esta carpeta se copiará el archivo *URDF* del brazo TITAN y la carpeta *meshes*. A mayores, es necesario añadir un

archivo de configuración del modelo. Para ello, en la misma carpeta, se crea un archivo de nombre *model.config* y se introduce el siguiente código:

Una vez lista la carpeta, se abre una terminal de Ubuntu y se introduce el comando `gazebo`. Con el programa ya abierto, se selecciona la pestaña *"Insert"* en la ventana izquierda y a continuación *"Add Path"*, se busca la ruta de la carpeta recién creada y se abre. Aparecerá el nombre del modelo, que deberá seleccionarse para añadirlo al mundo de Gazebo (**Figura 6-11**).

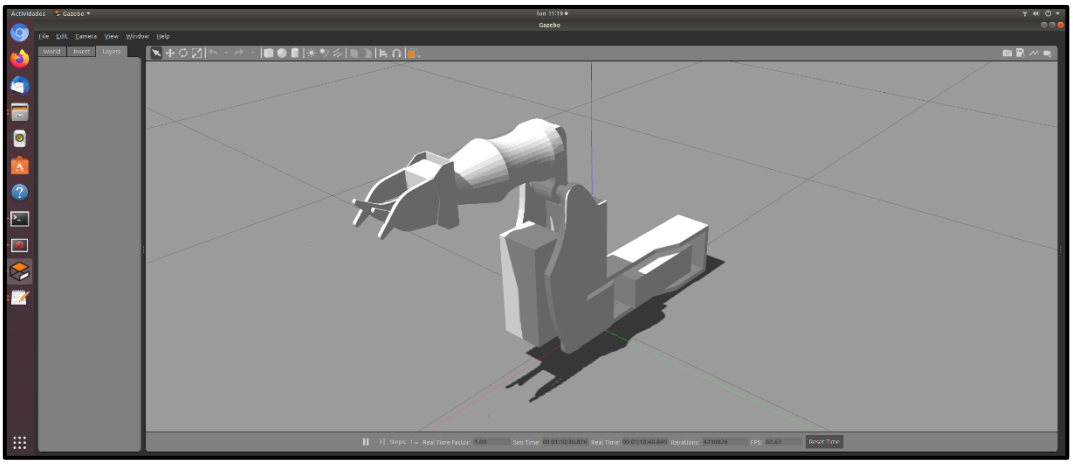


Figura 6-11. Modelo del brazo TITANROB en Gazebo.

6.3 Generación del espacio de trabajo *catkin*

Para la correcta utilización y gestión de los paquetes de ROS debe crearse un espacio de trabajo, al que se llamará *catkin_ws*. En este fichero se almacenan todos los archivos indicados en el capítulo 4, necesarios para el lanzamiento, modelado, comunicación y control del simulador. Desde el espacio de trabajo se hace llamada a todas las funciones ejecutadas durante el lanzamiento. Para crear el espacio de trabajo *catkin_ws*, se introducen los siguientes comandos en una terminal de Ubuntu:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin/ws
$ catkin_make
```

El comando *catkin_make* compilará el espacio de trabajo, creando un archivo *CMakeList.txt* en la carpeta *src*. Además, se crearán dos nuevos directorios: *build* y *devel*. Este el último almacena el archivo *setup.bash* que, al ejecutar el comando *source*, agregará la ruta de todos los paquetes del espacio *catkin_ws* al *PATH*, almacenados en la carpeta *src*, permitiendo que estos puedan ser llamados en archivos de lanzamiento:

Una vez construido el espacio de trabajo, se procede a crear los paquetes ROS del brazo, de los controles y del joystick virtual.

6.4 Creación de los paquetes ROS

Para la creación de los paquetes se hace uso de la herramientas ROS proporcionadas por *catkin*. En primer lugar, se construye el paquete de descripción del

brazo y, a continuación, el paquete con los archivos de control de las articulaciones. Finalmente, se crea un paquete para almacenar los archivos de generación del joystick virtual, construidos con *Qt Creator*, y conexión con el simulador.

6.4.1 Creación del paquete ROS del brazo

El paquete *bt_description* se construye mediante el empleo del comando *catkin_create_pkg*, que genera automáticamente los archivos de configuración de un paquete ROS. Seguidamente, se crean las tres carpetas principales de este: *urdf*, *meshes* y *launch*. Para ello, en una terminal, se introducen los siguientes comandos:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg bt_description
$ cd bt_description
$ mkdir urdf meshes launch
```

Una vez creados todos los directorios, se copian los archivos generados en SolidWorks para la descripción y los mallados editados en Blender del brazo TITANROB en las carpetas *urdf* y *meshes*, respectivamente. Posteriormente, se crean los archivos restantes del paquete, *brazo_titan.gazebo* en la carpeta urdf, para que el modelo de Gazebo pueda conectarse con ROS y ser controlado a través de su entorno, y *brazo_titna.launch* en la carpeta launch, para ejecutar el modelo en Gazebo.

```
$ touch urdf/brazo_titan.gazebo
$ touch launch/brazo_titan.launch
```

A mayores, en el archivo *brazo_titan.xacro* se incorporan los elementos *<transmission>* de cada articulación, para conectar estas con sus correspondientes controladores. El archivo se edita como si fuera un archivo de texto, ingresando la configuración manualmente.

6.4.2 Creación del paquete ROS de los controles

Igual que para el caso del paquete del brazo, para la construcción del paquete de los controles, *bt_control*, se introduce el comando *catkin_create_pkg*. A continuación, se crean los directorios *config* y *launch* y, dentro de estos, se escriben los archivos ce configuración necesarios.

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg bt_control
$ cd bt_control
$ mkdir config launch
$ touch config/bt_joint_states.yaml
$ touch config/bt_joint_controllers.yaml
$ touch launch/bt_control.launch
```

Con todos los elementos indicados ya creados, se procede a la edición de los archivos *bt_joint_states.yaml*, *bt_joint_controllers.yaml* y *bt_control.launch*, tal y como se indicó en el capítulo 4. Una vez estos estén listos, ya será posible ejecutar el simulador controlable por ROS a través de la Shell.

6.4.3 Creación del paquete ROS del joystick

Finalmente, se construye el paquete del joystick virtual, *bt_joystick*, donde se almacenarán todos los archivos necesarios para su lanzamiento y comunicación con el simulador. Mediante el uso de este joystick, es posible controlar el brazo robótico de tal forma que su comportamiento sea semejante al del brazo real, controlado mediante un joystick similar. Para ello, se empleará una vez más el comando *catkin_create_pkg*:

```
$ cd ~/catkin_make/src
$ catkin_create_pkg bt_joystick
```

Dentro desde directorio, se guardan todos los archivos de la app del joystick generados por el entorno de desarrollo *Qt Creator*, un ***IDE***[x] multi plataforma para el desarrollo de aplicaciones con Interfaces Gráficas de Usuario (GUI). Para que esta pueda comunicarse con el modelo de simulación, se crea otro archivo dentro del paquete, *transfer.py*, que publica las velocidades manejadas por el joystick en los topics correspondientes a cada una de las articulaciones.

```
$ cd ~/catkin_make/src/bt_joystick
$ touch transfer.py
```

Con todos los archivos creados y editados, queda listo el paquete del joystick.

6.5 Ajuste de los controladores PID

Construidos ya todos los paquetes, se procede a compilar el espacio de trabajo, habiendo así finalizado la construcción del simulador para el brazo TITANROB E500. No obstante, antes de que este pueda ser utilizado, deben ajustarse las ganancias de los controladores PID para cada una de las articulaciones del brazo robótico.

En la **Tabla 6-1** se muestran los valores de las ganancias proporcional, integral y derivativo de cada PID. Todos ellos fueron obtenidos mediante métodos de ajuste experimental, como el de Ziegler-Nichols.

Ganancia	ARTICULACIONES DEL ROBOT					
	1	2	3	4	pinza ext	pinza int
Proporcional (P)	50,0	270,0	30,5	5,0	1,2	1,2
Integral (I)	30000,0	162000,0	40666,7	540,0	720,0	720,0
Derivativa(D)	0,00000	0,00000	0,00000	0,00001	0,00030	0,00030

Tabla 6-1. Valores PID control del brazo TITANROB.

6.6 Lanzamiento y utilización del simulador

Para poder acceder a todos los archivos del proyecto, se creó un repositorio en *GitHub*, en el que se encuentran todos los archivos y directorios necesarios para el simulador almacenados en un paquete. Para su instalación, se clonará el repositorio en la carpeta *src* del espacio *catkin_ws* y se compilará:

```
$ cd ~/catkin_ws/src
$ git clone https://github.com/danirom97/brazo_titan.git
$ cd ..
$ catkin_make
```

Para ejecutar el simulador, en primer lugar, será necesario agregar las rutas de los archivos ROS y del espacio de trabajo *catkin_ws* al entorno del *PATH*. A continuación, bastará con ejecutar el archivo de lanzamiento *brazo_titan.launch*, del paquete *bt_description*:

```
$ source /opt/ros/melodic/setup.bash
$ source ~/catkin_ws/devel/setup.bash
$ roslaunch bt_description brazo_titan.launch
```

Al lanzar el archivo *launch*, se abrirá una terminal de Gazebo, donde se generará el modelo del brazo TITANROB E500, y se iniciarán los controladores (**Figura 6-12**).

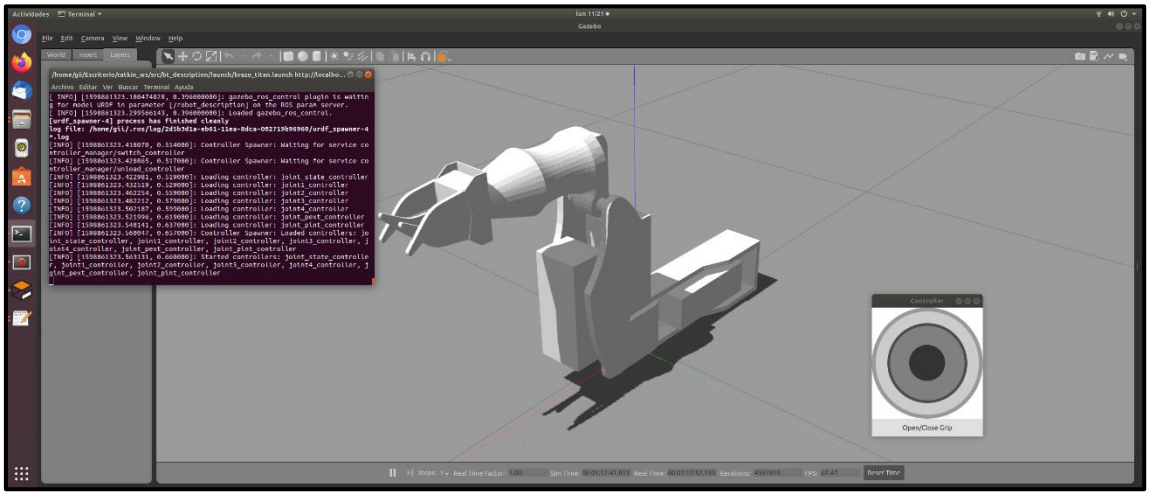


Figura 6-12. Brazo TITANROB E500 generado en Gazebo.

Por último, se abrirá la app del joystick (**Figura 6-13**), que tendrá los siguientes controles:

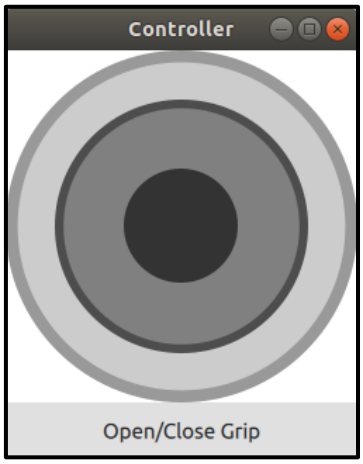


Figura 6-13. Control virtual tipo joystick construido en Qt Creator.

- Movimiento articulación 1 moviendo el joystick 1 de izquierda a derecha.
- Movimiento articulación 2 moviendo el joystick 1 de arriba abajo.
- Movimiento articulación 3 moviendo el joystick 2 de arriba abajo.
- Movimiento articulación 4 moviendo el joystick 2 de izquierda a derecha.

A modo de comprobación, se ejecuta el paquete *rqt_graph*, incluido en ROS, que proporciona una interfaz de usuario para visualizar la gráfica de computación de ROS, expuesta en la **Figura 6-14**. Así, en la , es posible ver la conexión entre los diferentes nodos ejecutados durante el proceso.

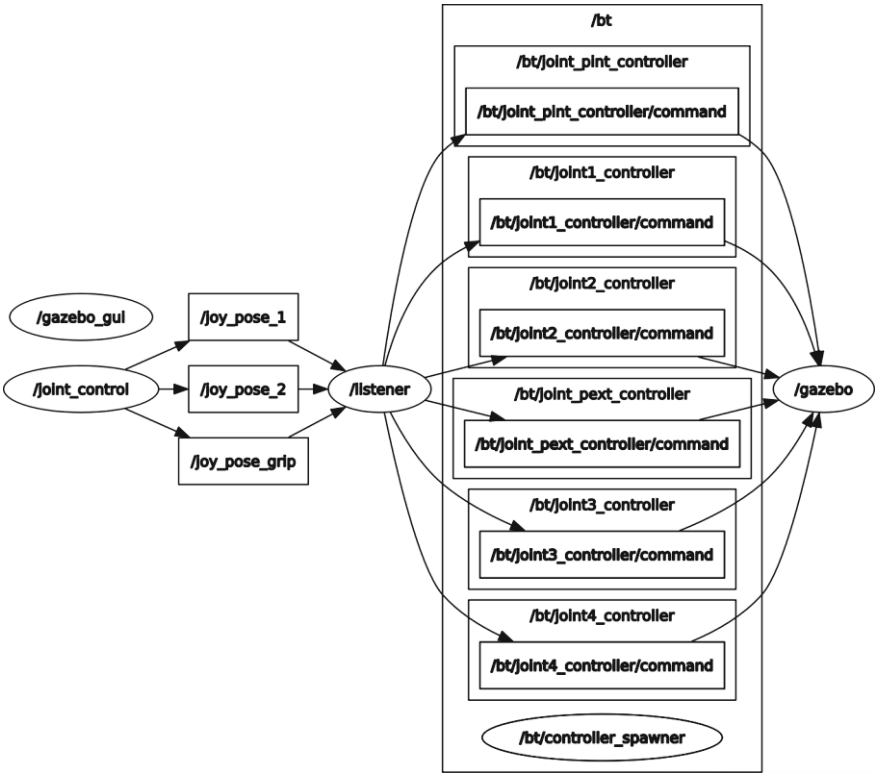


Figura 6-14. Gráfica de computación ROS para el simulador.

Como se puede comprobar, el nodo *joy_control* publica la posición relativa de los joysticks 1 y 2 y del botón de apertura de las pinzas en los *topics /joy_pose_1*, */joy_pose_2* y */joy_pose_grip*, respectivamente. Todos los mensajes publicados son recibidos por el nodo subscriptor *transfer*, que traduce la información y la publica en los correspondientes *topics* de control de las articulaciones, dentro del espacio de nombres */bt*. Todos los mensajes se envían a Gazebo para representar los movimientos.

7 PROTOCOLO DE PRUEBAS Y VALIDACIÓN

En este capítulo se describen las diferentes pruebas realizadas sobre el modelo de simulación desarrollado en este trabajo para el brazo TITANROB E500. El objetivo de estas pruebas es verificar el correcto funcionamiento de la simulación, comparando su comportamiento con el del modelo real. Para que el simulador pueda ser tomado como válido, los resultado obtenidos en las pruebas realizadas deberán, dentro de un margen aceptable, coincidir para ambos modelos, real y simulado.

En primer lugar, se realizarán una serie de pruebas de validación del modelo cinemático, comprobando las posiciones adoptadas por el efector final para unas mismas coordenadas articulares del modelo real y simulado. A continuación, se estudiará la posibilidad de eliminar los efectos de la gravedad sobre el modelo simulado, comprobando si las velocidades articulares se ven afectadas por los esfuerzos que esta ejerce sobre el brazo. Finalmente, se validará el modelo dinámico, demostrando que los movimientos articulares del brazo real y del brazo simulado son iguales.

7.1 Validación del modelo cinemático

A la hora de verificar que el modelo cinemático definido en el capítulo 4 es correcto, se realizaron varias pruebas en las que se comparó la posición adoptada por el efector final del brazo robótico real y del simulado. Para ello, se estableció en ambos modelos una misma posición de origen del espacio articular, ξ_A(0 0 0 0), un mismo sistema de referencia global para el espacio operacional, ξ_E, y un mismo sistema de referencia para la herramienta, *TCP*, respecto del punto *TP*. La posición de origen del espacio ξ_A corresponde a la mostrada en la **Figura 7-1**, donde también están señalados el sistema de referencia del espacio ξ_E, y el punto *TCP*, a 226 mm del *TP*.

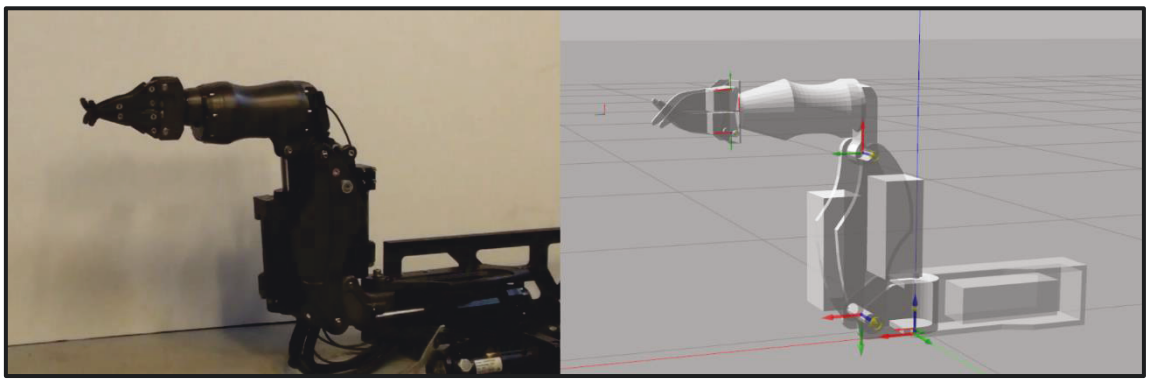


Figura 7-1. Posición articular de origen del brazo TITANROB.

La prueba realizada consistió en situar ambos brazos en unas mismas coordenadas articulares, $\boldsymbol{\theta_1}$, $\boldsymbol{\theta_2}$ y $\boldsymbol{\theta_3}$, determinando las componentes ***x***, ***y*** y ***z*** de la posición del *TCP*, respecto del espacio ξ_E, en el modelo simulado y en el modelo real. A continuación, se calcularon los errores absoluto, $\boldsymbol{E_a}$, y relativo, $\boldsymbol{E_r}$, entre las componentes de ambas posiciones. Si $\boldsymbol{p_V}$ representa la posición en el modelo simulado y $\boldsymbol{p_R}$ en el modelo real, entonces:

$$E_a = |p_V - p_R|$$

$$E_r = \frac{E_a}{R}$$

Donde ***R*** es el alcance máximo del brazo en su espacio operacional, de 916 mm. Se realizaron **9 mediciones**, aceptando solo errores relativos inferiores al **3.00%**, recogidas en la **Tabla 7-1**:

PRUEBA	q (θ_1, θ_2, θ_3) (º)			p_V (mm)			p_R (mm)		
	θ_1	θ_2	θ_3	x	y	z	x	y	z
1	0	0	0	586	0	401	594	0	404
2	0	0	-60	277	0	766	273	0	767
3	0	0	57	443	0	-32	451	0	-38
4	0	45	-60	771	0	428	766	0	431
5	0	45	0	732	0	-49	740	0	-46
6	96	0	-60	-29	275	766	-27	255	770
7	96	0	57	-46	441	-32	-43	456	-30
8	96	45	-60	-81	767	428	-64	757	443
9	96	45	0	-76	728	-49	-59	750	-47

Prueba	q (θ_1, θ_2, θ_3) (º)			E_a (mm)			E_r		
	θ_1	θ_2	θ_3	x	y	z	x	y	z
1	0	0	0	8	0	3	**0,87%**	**0,00%**	**0,33%**
2	0	0	-60	4	0	1	**0,44%**	**0,00%**	**0,11%**
3	0	0	57	8	0	6	**0,87%**	**0,00%**	**0,66%**
4	0	45	-60	5	0	3	**0,55%**	**0,00%**	**0,33%**
5	0	45	0	8	0	3	**0,87%**	**0,00%**	**0,33%**
6	96	0	-60	2	20	4	**0,22%**	**2,18%**	**0,44%**
7	96	0	57	3	15	2	**0,33%**	**1,64%**	**0,22%**
8	96	45	-60	17	10	15	**1,86%**	**1,09%**	**1,64%**
9	96	45	0	17	22	2	**1,86%**	**2,40%**	**0,22%**

Tabla 7-1. Comprobación del modelo cinemático del brazo TITAN

Como se puede comprobar, las posiciones adoptadas por el punto *TCP* en ambos modelos son muy próximas, con errores relativos inferiores al **2.50%** y un error medio del **0.72%**, muy por debajo del error máximo permitido. De esta manera queda demostrado que el modelo cinemático del brazo robótico es correcto.

7.2 Estudio de los efectos de la gravedad

Con la intención de simplificar el modelo dinámico, se estudió la posibilidad de eliminar el efecto de las cargas gravitatorias sobre las articulaciones del brazo robótico en el simulador. Por ello, se realizaron una serie de pruebas en las que se comprobó si los esfuerzas provocados por estas cargas tenían algún efecto sobre las velocidades articulares del robot, existiendo dos posibles situaciones:

I. Las velocidades son constantes y por tanto puede anularse el efecto de las cargas gravitatorias.
II. Las velocidades no son constantes y dependen de la posición de las articulaciones, no pudiendo entonces anular el efecto de las cargas gravitatorias.

Para estas pruebas se realizó una medición de los tiempos de subida, t_s, y bajada, t_b, de las articulaciones 2 y 3, correspondientes al hombro y el codo, al ser estas las más afectadas por las cargas gravitatorias, con **5** repeticiones por medición, cinco de subida y cinco de bajada. A continuación, se calcularon los errores absoluto, E_a, y relativo, E_r, a partir de los valores promedio de todas las repeticiones de los tiempos t_s y t_b:

$$E_a = |\bar{t_s} - \bar{t_b}|$$

$$E_r = \frac{E_a}{\bar{t_b}}$$

Para poder afirmar que las velocidades articulares son constantes se consideró un error relativo máximo del **2.00%**. Los resultados obtenidos se recogen en la **Tabla 7-2**:

Articulación		2	3
Prueba 1	T_{s1}	19,24	17,88
	T_{b1}	19,3	18,554
Prueba 2	T_{s2}	19,21	17,78
	T_{b2}	19,33	18,09
Prueba 3	T_{s3}	19,26	18,87
	T_{b3}	19,28	17,84
Prueba 4	T_{s4}	18,83	18,06
	T_{b4}	18,99	17,72
Prueba 5	T_{s5}	19,01	18,3
	T_{b5}	18,81	18,23
Valores medios	T_s	19,11	18,178
	T_b	19,142	18,0868
E_a		**0,032**	**0,0912**
E_r		**0,17%**	**0,50%**

Tabla 7-2. Tiempos de subida y bajada de las articulaciones del brazo TITAN.

Tal y como muestra la **Tabla 7-2**, los tiempos medidos de subida y bajada de las articulaciones 2 y 3 apenas difieren, con un error relativo de **0.17% para la articulación 2** y de **0.50% para la 3**, ambos por debajo del límite permitido. Además, en ocasiones los tiempos de subida son inferiores a los tiempos de bajada, lo que da entender que estas diferencias se deben más a las pequeñas imperfecciones en la articulación, así como en la medición, que a los efectos de la gravedad.

7.3 Validación del modelo dinámico

El principal objetivo de este trabajo es obtener un modelo de simulación del brazo TITANROB E500 que pueda ser utilizado en lugar del brazo real. Por tanto, si el comportamiento de ambos modelos difiere notablemente, no se cumplirá el requisito principal y el modelo desarrollado no será válido. Es por ello por lo que es necesario comprobar si para unas mismas órdenes de mando los movimientos del modelo simulado son similares a los del brazo real.

Con el fin de validar que el comportamiento del modelo de simulación, desarrollado en Gazebo y ROS, es el mismo que en el brazo real, se realizaron pruebas con las que comparar los tiempos requeridos por ambos modelos para mover una determinada articulación entre sus ángulos límites a máxima velocidad. Para cada articulación, se hicieron **5** repeticiones en el modelo real (t_R) y en el simulado (t_V), recogidas en la **Tabla 7-3**, calculando a continuación los errores absoluto, E_a, y relativo, E_r:

$$E_a = |\overline{t_V} - \overline{t_R}|$$

$$E_r = \frac{E_a}{\overline{t_R}}$$

Para tomar el modelo dinámico como válido, la discrepancia entre los resultados del modelo real y simulado no debe ser superior al **3.00%**.

Prueba	t_R (s)				t_V (s)			
	1	2	3	Pinza	1	2	3	Pinza
1	19,39	19,27	18,217	3,54	19,295	19,162	18,110	3,3699
2	19,21	19,27	17,935	3,32	19,295	19,165	18,110	3,3721
3	19,17	19,27	18,355	3,36	19,295	19,160	18,110	3,3708
4	19,52	18,91	17,89	3,37	19,295	19,164	18,110	3,372
5	19,30	18,91	18,265	3,46	19,295	19,162	18,110	3,371
Valor medio	19,32	19,13	18,13	3,41	19,29	19,16	18,11	3,37
E_a					**0,02304**	**0,03658**	**0,0224**	**0,03884**
E_r					**0,12%**	**0,19%**	**0,12%**	**1,14%**

Tabla 7-3. Comprobación del modelo dinámico del brazo TITAN.

A mayores de los datos de la tabla, se han gravado una serie de videos en los que se comparan los movimientos realizados por ambos modelos. En ellos se puede corroborar el correcto funcionamiento del simulador, cuyos movimientos son muy parecidos a los ejecutados por el brazo real.

8 RESULTADOS

En el presente capítulo, se hará un análisis completo de los resultados que se han conseguido en el desarrollo e implementación del simulador. Estos resultados corresponderán a cada una de las fases en las que se divide este proyecto, demostrando el correcto funcionamiento de las diferentes partes que lo conforman. Concretamente, se distinguirán tres etapas principales:

1. **Apertura del modelo en Gazebo.**
2. **Comunicación de Gazebo con los paquetes de ROS.**
3. **Validación del comportamiento del modelo de simulación.**

8.1 Apertura del modelo en Gazebo

En primer lugar, es necesario verificar, dentro del simulador, el modelo de descripción del brazo robótico. Se ha de comprobar que el modelo simulado, construido en formato URDF y posteriormente adaptado al formato XACRO, cumple con los requisitos establecidos en las especificaciones de diseño del proyecto. En otras palabras, el modelo final en formato XACRO debe representar un modelo de robot equivalente al brazo TITANROB en cuanto a diseño 3D y parámetros dimensionales se refiere. Además, hay que asegurarse de que el software del simulador de Gazebo reconoce y abre correctamente los archivos del modelo.

Para confirmar que estos criterios se cumplen, basta con abrir el modelo en Gazebo desde el repositorio de modelos como se ve en la **Figura 8-1**. Al comparar este con real, es evidente la similitud entre ambos.

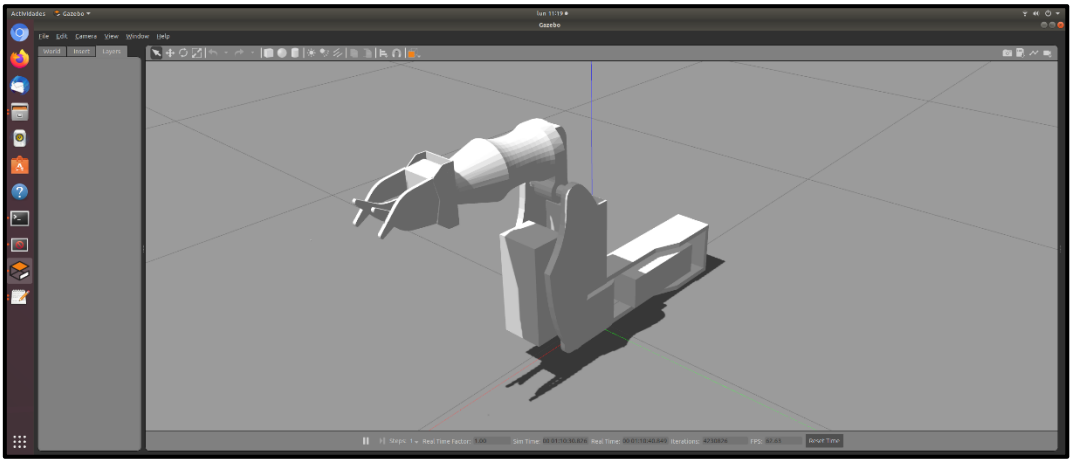


Figura 8-1. Modelo del brazo TITANROB abierto en Gazebo.

8.2 Comunicación entre ROS y Gazebo

La comunicación entre ROS y Gazebo es una de las etapas más importantes de todo el proyecto, pues de ella depende el correcto funcionamiento del simulador. Si

Gazebo no es capaz de conectarse a la gráfica de computación de ROS, no podrá interactuar con el resto de los nodos, siendo entonces imposible el envío de mensajes entre ambos y, por tanto, el control de las articulaciones a través de la terminal.

Para comprobar que ambas plataformas se comunican correctamente, basta con ejecutar el archivo de lanzamiento *brazo_titan.launch* y comprobar que todos los componentes del simulador se inician sin problemas.

Lanzamiento del modelo

Al ejecutar el archivo de lanzamiento en el entorno ROS, este accederá al paquete *gazebo_ros* y ejecutará el archivo *empty_world.launch*, que abre un mundo vacío de Gazebo. Simultáneamente, se cargará el modelo del brazo TITAN en el servidor de parámetros, que se generará mediante el nodo *spawn_model*. Si todo está correctamente configurado, se abrirá una terminal en Gazebo con un mundo vacío y con el modelo situado en el origen de coordenadas.

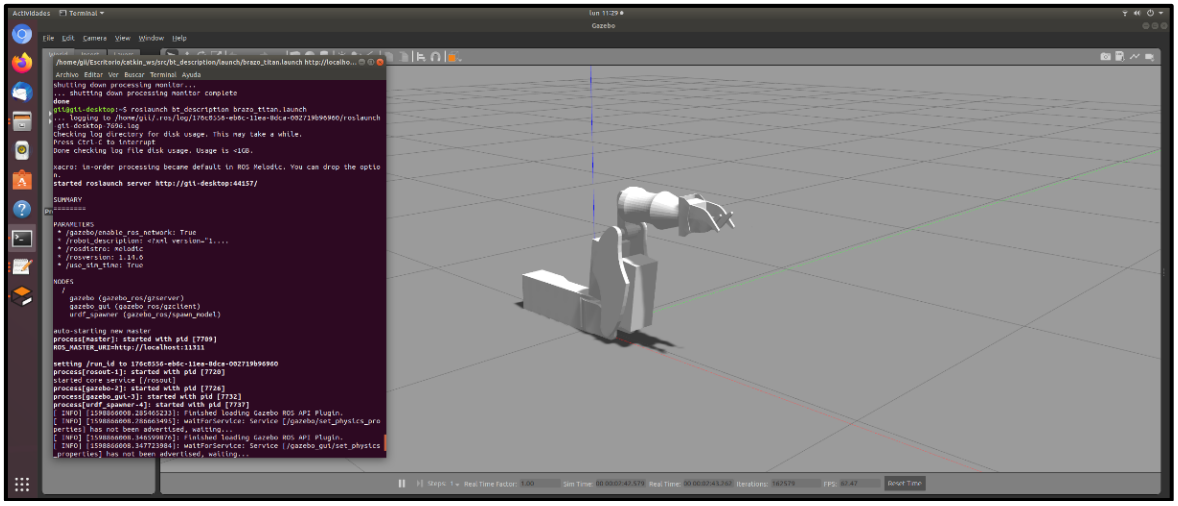


Figura 8-2. Ejecución del modelo a través de ROS.

Todos los procesos mencionados pueden observarse en el la terminal *shell* de la **Figura 8-2**, donde, al ejecutar el archivo *launch*, comienza a ejecutarse el resto de las tareas. Con esto, queda comprobada la correcta generación de los modelos en Gazebo mediante ROS.

Lanzamiento de los controles

Tras el lanzamiento del modelo, deberá iniciarse el archivo de lanzamiento de los controles, *bt_controllers.launch*, que iniciará los controladores.

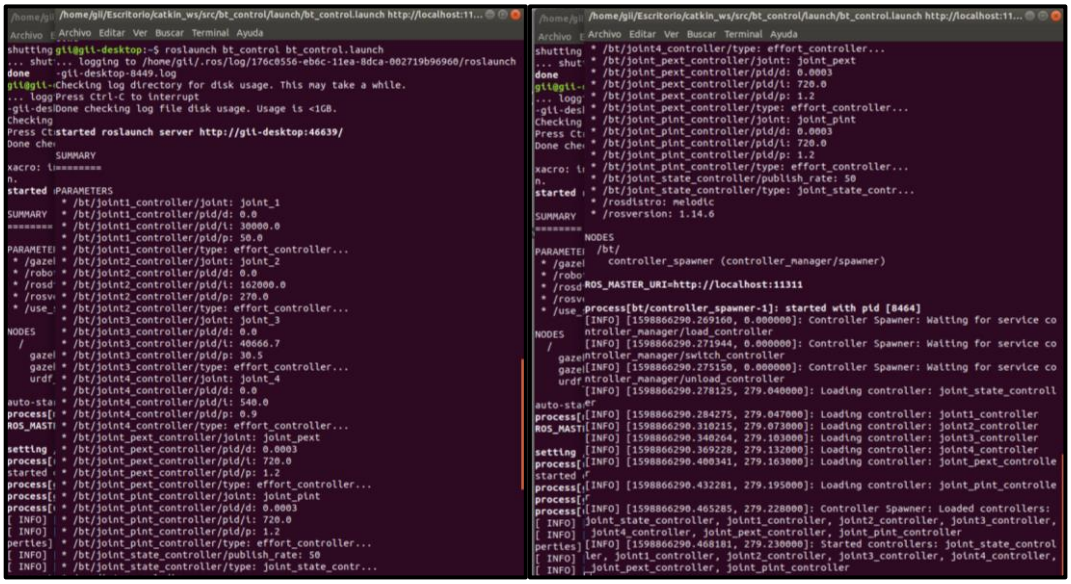


Figura 8-3. Ejecución de los controles a través de ROS.

Como puede verse en la **Figura 8-3**, primero el archivo carga todos los controladores definidos en los archivos *yaml*. A continuación, inicia el nodo *spawner* que genera todos los controladores anteriores y los conecta con las interfaces de hardware correspondientes. Finalmente, se inicia el nodo para ejecutar el control del joystick virtual.

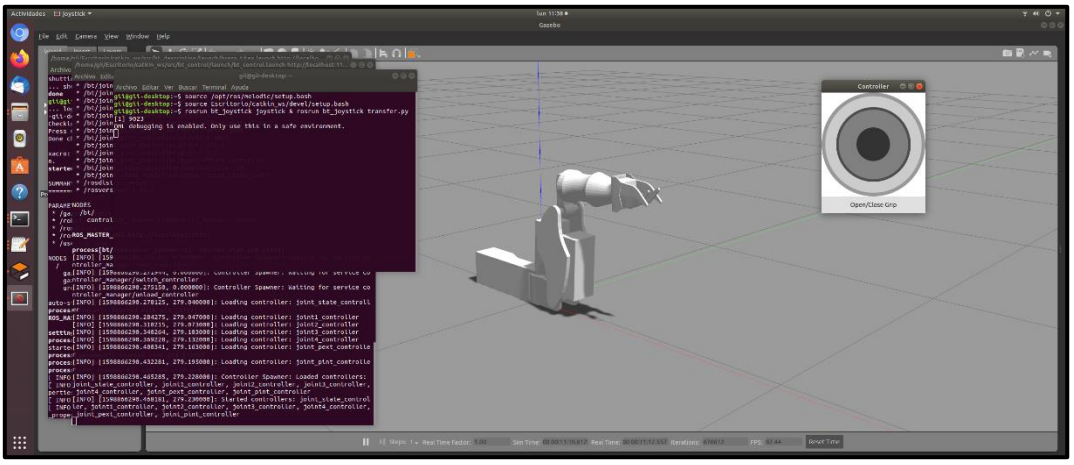


Figura 8-4. Ejecución de los nodos del joystick.

Llegados a este punto, si el simulador responde correctamente a los comandos a través de la interfaz o a los movimientos del joystick, se puede verificar el funcionamiento del simulador. Con ello, únicamente sería necesario validar sus movimientos en cuanto a su similitud con los del robot real.

8.3 Validación del simulador

En todas las pruebas descritas anteriormente en el capítulo 6, el simulador muestra unos resultados muy satisfactorios. Para cualquiera de estas pruebas, los errores

relativos de los valores del modelo de simulación respecto de los del brazo robótico real no superan el **2.50%**, muy por debajo del error máximo permitido para cada una de ellas.

Concretamente, para la validación del modelo cinemático, el error máximo producido es de un **2.40%**, con un error medio del **0.72%**, siendo el máximo permitido un **3.00%**, tal y como se recoge en la **Tabla 7-1**. Por su parte, en las pruebas de velocidades para el modelo dinámico (**Tabla 7-3**), el error más alto corresponde a un **1.14%** para las articulaciones de las pinzas de la herramienta, con un margen máximo aceptable de **3.00%**. Por su parte, la articulación 1 presenta un error del **0.12%**, la 2 de un **0.19%** y la 3 de un **0.12%**. Estos resultados permiten confirmar que los movimientos articulares realizados por el modelo del simulador coinciden, dentro de un margen más que aceptable, con los del modelo real.

Hay que destacar que este modelo dinámico finalmente fue diseño descartando los efectos de la gravedad, tras observarse en las pruebas realizadas (**Tabla 7-2**) que estos no producían un efecto notable sobre la velocidad de las articulaciones. En estas, las diferencias entre las velocidades de subida y bajada no fueron superiores al **0.50%**, con un margen de error del **2.00%**.

Además, para demostrar estos buenos resultados, se han grabado una serie de videos en los que se ponen en comparación los movimientos de ambos modelos. Todos ellos han sido subidos a un repositorio de *GitHub* junto con los paquetes de las diferentes partes que conforman el simulador, al que se puede acceder desde el siguiente enlace:

https://github.com/danirom97/brazo_titan.git

9 ESTUDIO DE APLICABILIDAD

Una vez analizados los resultados obtenidos y comprobado el correcto funcionamiento del modelo de simulación, se pueden destacar los siguientes aspectos acerca las tareas realizadas a lo largo del proyecto respecto de los distintos subobjetivos planteados para este:

- **Definición del modelo fisicomatemático del brazo:** Se ha creado un modelo para el brazo TITAN que cumple con las especificaciones del robot real, integrando tanto los elementos de visualización como los elementos de cálculo dinámico de sus componentes.
- **Integración del modelo dinámico en el simulador de Gazebo:** La descripción fisicomatemática del brazo TITAN ha sido exportada a un formato basado en *XML*, siendo así compatible con el software de Gazebo, desde donde es posible reproducir su comportamiento dinámico.
- **Implementación y ajuste de los controles de las articulaciones:** Gracias a las herramientas proporcionadas por los paquetes *ros_control* y *gazebo_ros_pkgs* se han construido los archivos para la generación de las interfaces de hardware y de los controladores que permiten comunicar Gazebo con el entorno de ROS, y así gobernar el funcionamiento de las articulaciones del brazo TITAN en el simulador.
- **Validación del modelo desarrollado:** Las pruebas realizadas con el simulador, descritas en capítulo 6, han permitido corroborar que sus movimientos se corresponden con los del brazo real con un error medio inferior al 0.50%.

A mayores de lo anterior, se incorporó un joystick virtual basado en *QML* para que el modelo simulado pudiera ser controlado de la misma forma que el brazo TITAN real, ofreciendo así un control mucho más sencillo e intuitivo, sin necesidad de escribir e introducir comandos en la interfaz de ROS para ejecutar cada movimiento del robot.

Como conclusión, con la realización de todas las tareas anteriores, se ha cumplido el objetivo principal de este Trabajo de Fin de Grado de desarrollar un modelo de simulación para el brazo robótico de un vehículo submarino, que represente fielmente su comportamiento, permitiendo así la realización de simulaciones realistas. Este modelo puede ser utilizado en sustitución del brazo real para las labores de investigación llevadas a cabo por el GII como parte del proyecto SAILOR. Para su control por joystick, no es necesario conectarlo a la laptop del operador a través de conexiones RS485, lo cual supone una notable reducción en los tiempos de montaje y puesta en marcha del brazo. Además, las continuas incidencias en la conexión entre estos componentes provocaban interrupciones en la comunicación entre la laptop y el robot, dando lugar a importantes retrasos a la hora de realizar pruebas con el brazo. Gracias al trabajo realizado en este proyecto, es posible efectuar pruebas sin necesidad de disponer de la planta física del brazo ni de encontrarse en laboratorio del canal, pudiendo así prescindir de multitud de recursos materiales y económicos y reducir los plazos de tiempo de las pruebas realizadas.

9.1 Comparación con simuladores ya existentes

No es posible realizar una comparación global de este modelo de simulación con respecto a otros sistemas ya existentes, pues se trata de un modelo diseñado para unas condiciones muy concretas. No obstante, pueden destacarse dos ventajes que lo hacen preferible frente a otros simuladores de manipuladores robóticos. Por un lado, presenta una configuración más simple, sin necesidad de complementes para el ajuste y compensación de los actuadores, pues mediante la experimentación se ha podido simplificar el modelo dinámico, obviando los efectos de la gravedad. Por otro, el joystick virtual incorporado ofrece un control mucho más cómodo que, a diferencia de la mayoría de los modelos de simulación disponibles, permite controlar el simulador sin necesidad de conocer los comandos básicos para la comunicación con nodos y el envío de mensajes a través de ROS.

9.2 Trabajo futuro

A pesar de haber finalizado el trabajo, cumpliendo con su objetivo global y obteniendo unos resultados satisfactorios en todas las pruebas realizadas, el modelo de simulación elaborado aún se encuentra en etapa temprana de desarrollo.

Entre las posibles líneas de trabajo futuras se encuentra la adaptación del modelo a un entorno acuático mediante el paquete UUV Simulator, que proporciona las herramientas necesarias para simular vehículos marinos en la plataforma ROS/Gazebo, y la posterior validación de sus movimientos, para lo cual será necesario realizar pruebas con el brazo sumergido.

Otra posible línea sería la unión de este último modelo al ya desarrollado para el submarino KAI, obteniendo así un sistema completo del vehículo submarino y el brazo.

Por último, se podrían diseñar mundos de simulación que representen de manera realista entornos de trabajo del sistema submarino, como el mantenimiento de una plataforma marítima o la reparación de una estructura offshore, dando la posibilidad de realizar experimentos completos sin poner en riesgo el equipo real.

10 PRESUPUESTO

Capítulo 1: Materiales de Hardware

Nº	Concepto	Unidades	Precio unitario (€)	Importe (€)
1	Ordenador EVEN Cooler Master con Intel Core i7-7700 y RAM de 15 Gb	1	1100	700,00 €
2	Disco duro Samsung SSD 860 de 500 Gb	1	150	150,00 €
3	Tarjeta gráfica GeForce GTX 1060 NVIDIA de 4 Gb	1	500	300,00 €
Importe total del capítulo 1				1.150,00 €

Capítulo 2: Materiales de Software

Nº	Concepto	Unidades	Precio unitario (€)	Importe (€)
1	Licencia SolidWorks Eductional Edition 2018-19	1	99	99,00 €
Importe total del capítulo 2				99,00 €

Capítulo 3: Mano de obra

Nº	Concepto	Unidades (h)	Precio unitario (€)	Importe (€)
1	Diseño ingenieril	640	30	19.200,00 €
Importe total del capítulo 3				19.200,00 €

Resumen por capítulos

Capítulo 1: Materiales de Hardware	1.150,00 €
Capítulo 2: Materiales de Software	99,00 €
Capítulo 3: Mano de obra	19.200,00 €
IMPORTE DE EJECUCIÓN MATERIAL	20.449,00 €

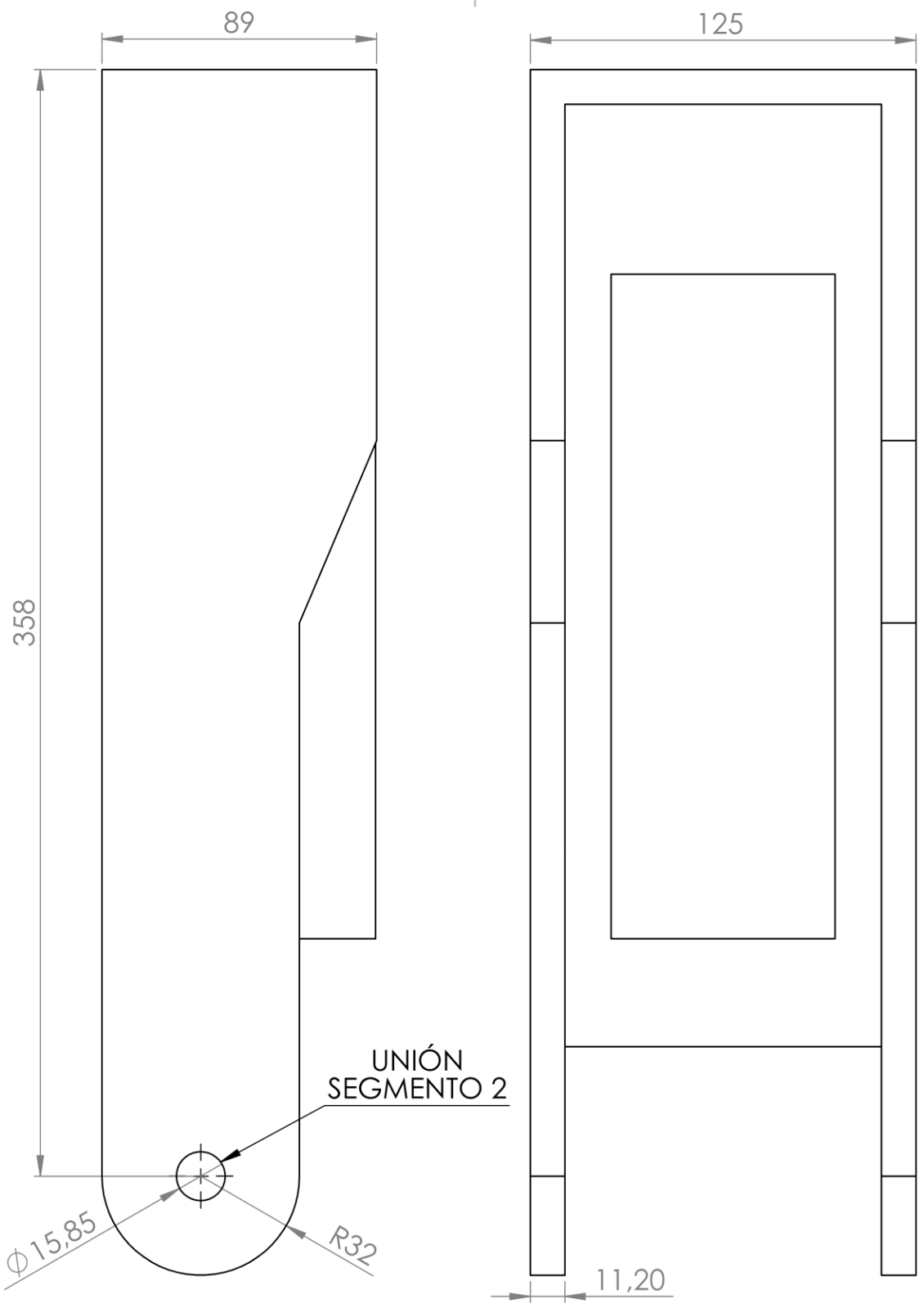
IMPORTE DE EJECUCIÓN MATERIAL	20.449,00 €
Gastos generales (13%)	2.658,37 €
Beneficio Industrial (6%)	1.226,94 €
IMPORTE DE EJECUCIÓN	24.334,31 €
IVA (21%)	5.110,21 €
IMPORTE DE CONTRATA	29.444,52 €

El importe de contrata es de **VEINTINUEVE MIL CUATROCIENTOS CUARENTA Y CUATRO EUROS CON CINCUENTA Y DOS CENTIMOS**.

11 ANEXOS

11.1 Anexo 1: Planos del Brazo TITANROB E500

En el siguiente Anexo, se incluyen los planos del brazo TITANROB E500 realizados por Guillermo Saavedra Soto en su proyecto [4]. Estos han sido utilizados, junto con los correspondientes ensamblajes y piezas en SolidWorks, para definir los modelos cinemático, dinámico y visual del brazo robótico desarrollado en este Trabajo de Fin de Grado.



PROYECTO:

MODELADO MECÁNICO DE VEHÍCULO SUBMARINO HÍBRIDO ROV/AUV

UNIVERSIDADE DA CORUÑA

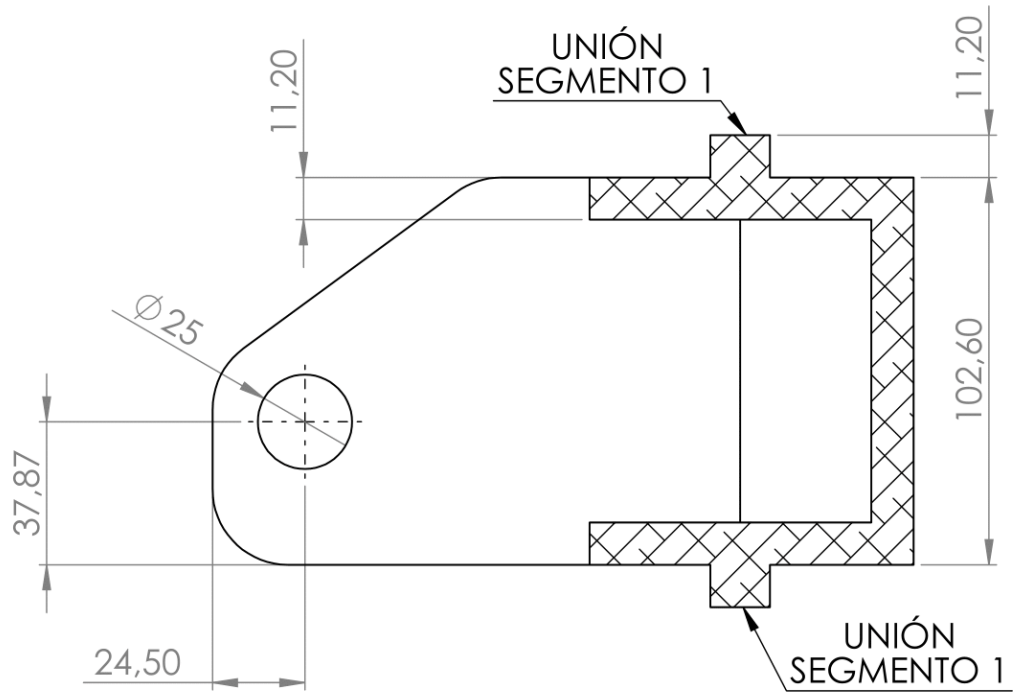
TÍTULO:

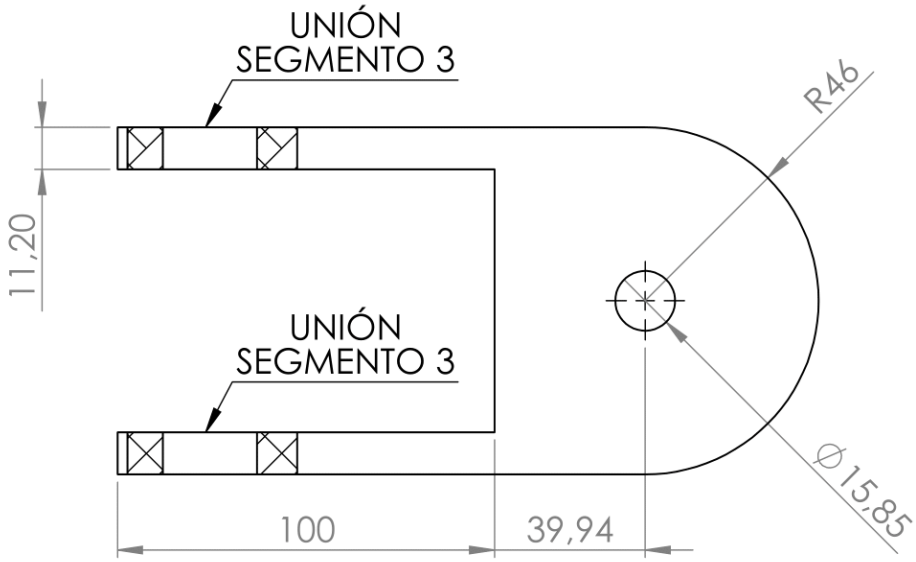
BRAZO ARTICULADO:
SEGMENTO 1

Nº PLANO:

55

ESCALA:	FECHA:	AUTOR:
1:2	1/7/19	GUILLERMO SAAVEDRA SOTO





PROYECTO:

MODELADO MECÁNICO DE VEHÍCULO SUBMARINO HÍBRIDO ROV/AUV

UNIVERSIDADE DA CORUÑA

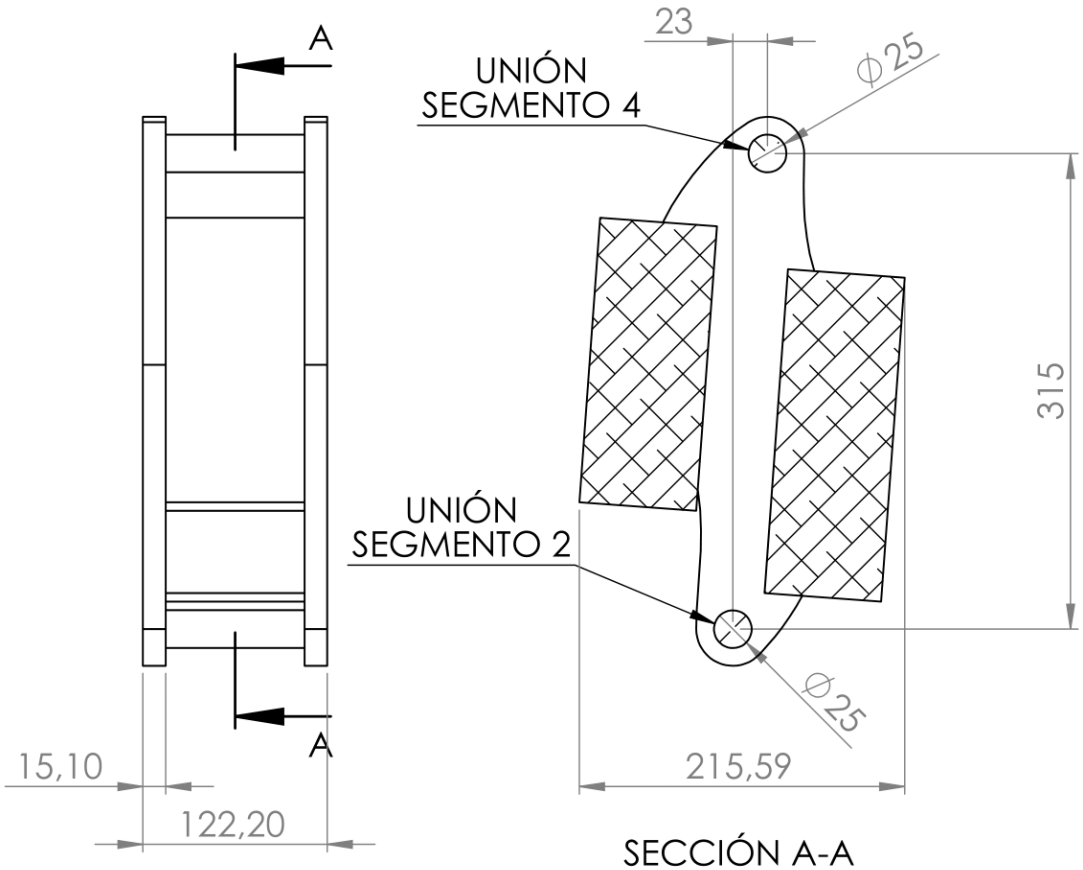
TÍTULO:

BRAZO ARTICULADO:
SEGMENTO 2

Nº PLANO:

56

ESCALA:	FECHA:	AUTOR:
1:2	1/7/19	GUILLERMO SAAVEDRA SOTO



SECCIÓN A-A

PROYECTO:

MODELADO MECÁNICO DE VEHÍCULO SUBMARINO HÍBRIDO ROV/AUV

UNIVERSIDADE DA CORUÑA

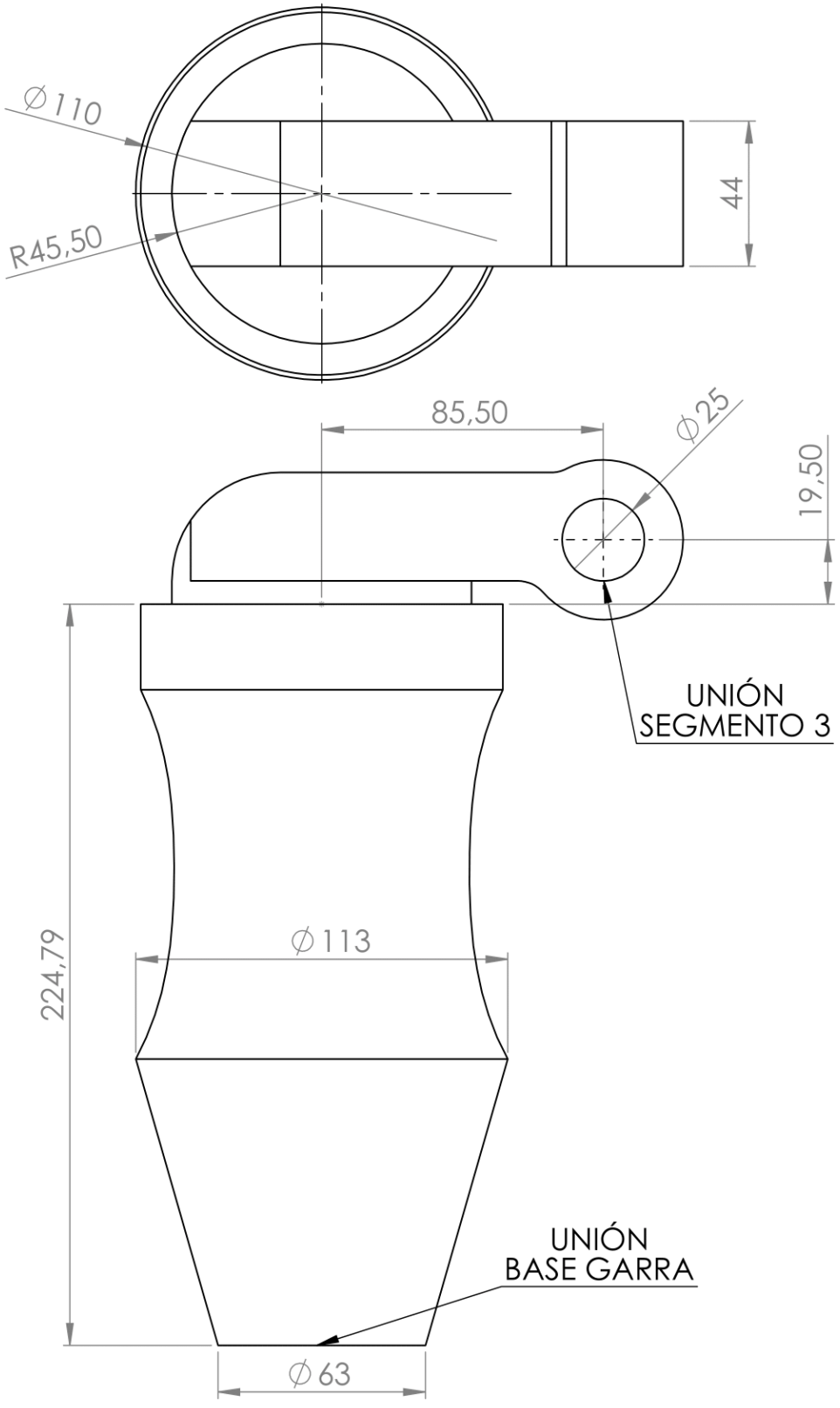
TÍTULO:

BRAZO ARTICULADO:
SEGMENTO 3

Nº PLANO:

57

ESCALA:	FECHA:	AUTOR:
1:5	1/7/19	GUILLERMO SAAVEDRA SOTO



PROYECTO:

MODELADO MECÁNICO DE VEHÍCULO SUBMARINO HÍBRIDO ROV/AUV

UNIVERSIDADE DA CORUÑA

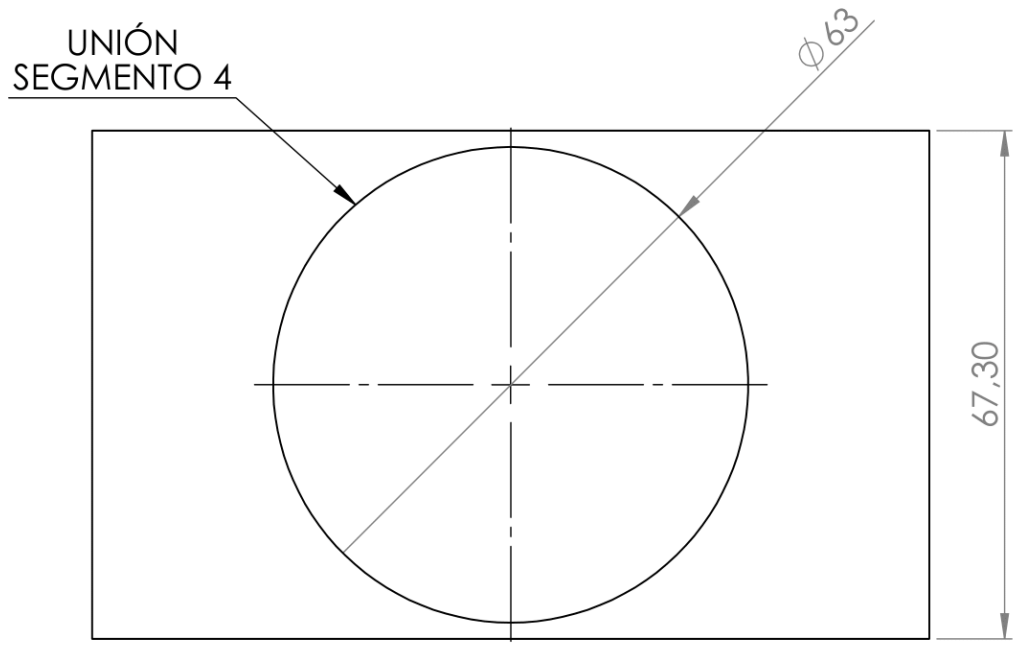
TÍTULO:

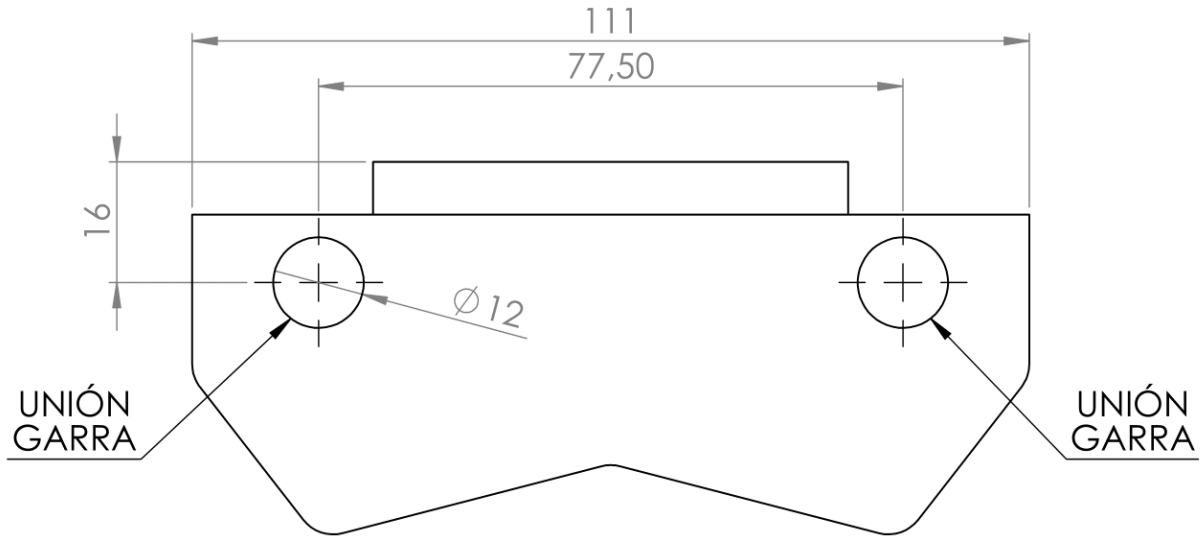
BRAZO ARTICULADO:
SEGMENTO4

Nº PLANO:

58

ESCALA:	FECHA:	AUTOR:
1:2	1/7/19	GUILLERMO SAAVEDRA SOTO





PROYECTO:

MODELADO MECÁNICO DE VEHÍCULO SUBMARINO HÍBRIDO ROV/AUV

UNIVERSIDADE DA CORUÑA

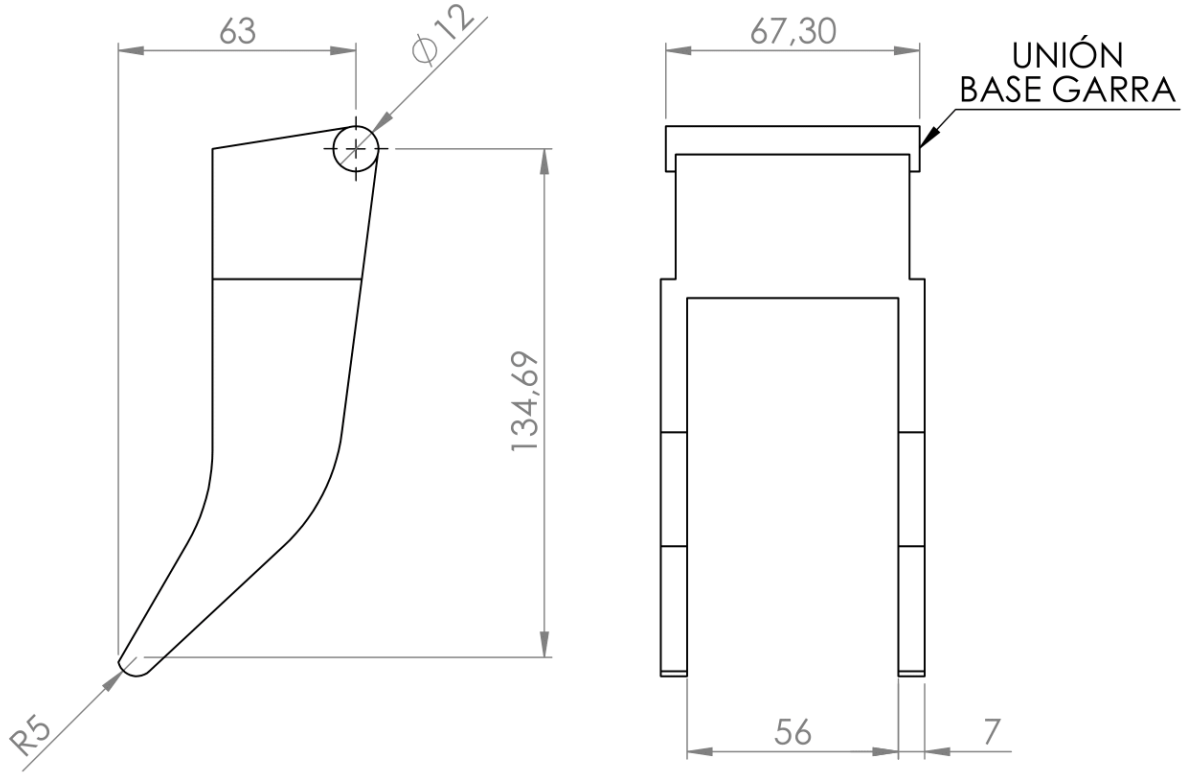
TÍTULO:

BRAZO ARTICULADO:
BASE GARRA

Nº PLANO:

59

ESCALA:	FECHA:	AUTOR:
1:1	1/7/19	GUILLERMO SAAVEDRA SOTO



PROYECTO:

MODELADO MECÁNICO DE VEHÍCULO SUBMARINO HÍBRIDO ROV/AUV

UNIVERSIDADE DA CORUÑA

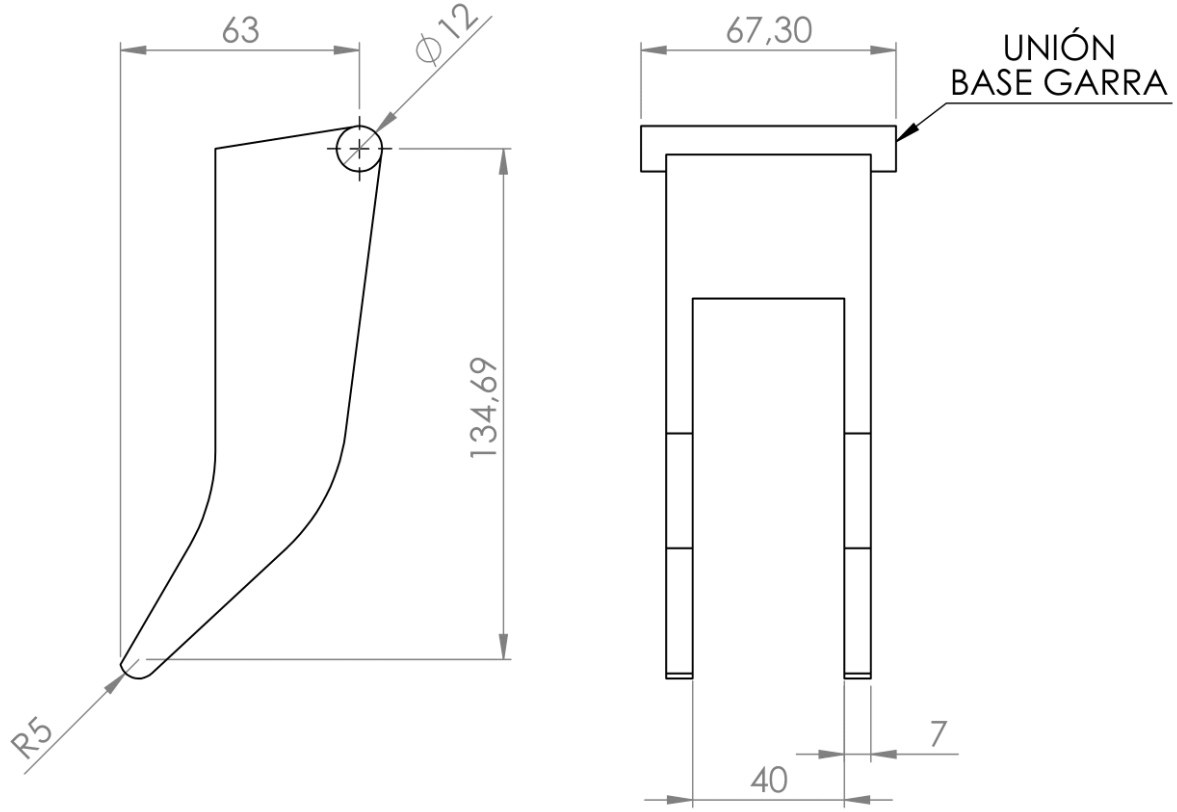
TÍTULO:

BRAZO ARTICULADO:
GARRA MAYOR

Nº PLANO:

60

ESCALA:	FECHA:	AUTOR:
1:2	1/7/19	GUILLERMO SAAVEDRA SOTO



PROYECTO:

MODELADO MECÁNICO DE VEHÍCULO SUBMARINO HÍBRIDO ROV/AUV

UNIVERSIDADE DA CORUÑA

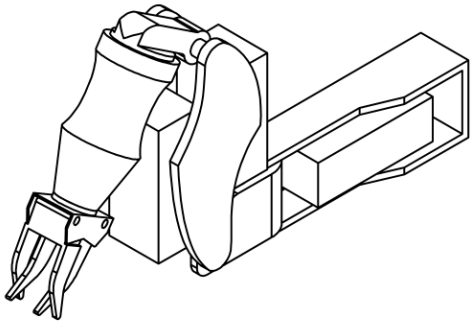
TÍTULO:

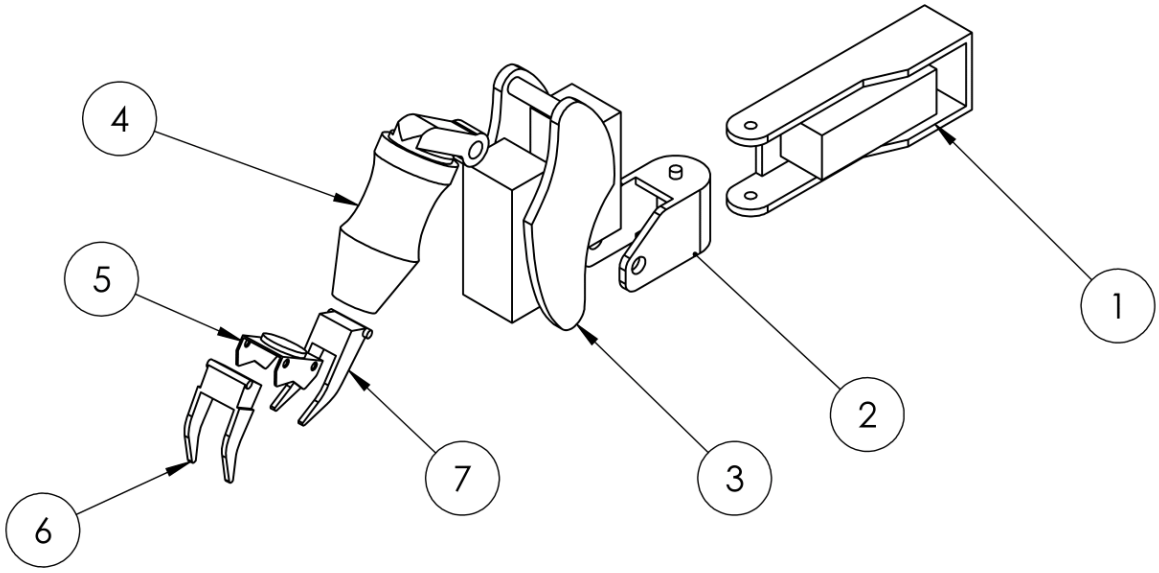
BRAZO ARTICULADO:
GARRA MENOR

Nº PLANO:

61

ESCALA:	FECHA:	AUTOR:
1:2	1/7/19	GUILLERMO SAAVEDRA SOTO





N.º DE ELEMENTO	NOMBRE DE PIEZA	CANTIDAD
1	Segmento 1	1
2	Segmento 2	1
3	Segmento 3	1
4	Segmento 4	1
5	Base garra	1
6	Garra mayor	1
7	Garra menor	1

PROYECTO:

MODELADO MECÁNICO DE VEHÍCULO SUBMARINO HÍBRIDO ROV/AUV

UNIVERSIDADE DA CORUÑA

TÍTULO:

ENSAMBLAJE BRAZO ARTICULADO

Nº PLANO:

62

ESCALA:	FECHA:	AUTOR:
1:10	1/7/19	GUILLERMO SAAVEDRA SOTO

11.2 Anexo 2: Código QML de configuración del joystick virtual

```
import QtQuick 2.12
import QtQuick.Window 2.12
import QtQuick.Controls 2.5

import Ros 1.0

ApplicationWindow {
    id: background
    visible: true
    width: joystick.width
    height: joystick.height + gripButton.height
    title: qsTr("Controller")

    property int i: 1

    Image {
        id: joystick

        property real angle : 0
        property real distance : 0

        source: "background.png"
        x: 0
        y: 0

        Rectangle {
            id: rect1
            width: 80
            height: 80
            x: joystick.width *0.5
            y: joystick.height * 0.5
            color: "transparent"

        }

        ParallelAnimation {
            id: returnAnimation
            NumberAnimation { target: thumb1.anchors; property:
"horizontalCenterOffset";
                to: 0; duration: 200; easing.type: Easing.OutSine
}
            NumberAnimation { target: thumb1.anchors; property:
"verticalCenterOffset";
                to: 0; duration: 200; easing.type: Easing.OutSine
}
            onFinished: joypose1.publish()
        }

        MouseArea {
            id: mouse
```

```
            property real fingerAngle : Math.atan2(mouseX,
mouseY)
            property int mcx : mouseX - joystick.width * 0.5
            property int mcy : mouseY - joystick.height * 0.5
            property bool fingerInBounds : fingerDistance2 <
distanceBound2
            property real fingerDistance2 : mcx * mcx + mcy * mcy
            property real distanceBound : joystick.width * 0.5 -
thumb1.width * 0.5
            property real distanceBound2 : distanceBound *
distanceBound

            property double signal_x : (mouseX -
joystick.width/2) / distanceBound
            property double signal_y : -(mouseY -
joystick.height/2) / distanceBound

            anchors.centerIn: parent
            width: thumb1.width
            height: thumb1.height

            onPressed: {
                returnAnimation.stop();
            }

            onReleased: {
                returnAnimation.restart();
            }

            onPositionChanged: {
                if (i == 1){
                    if (fingerInBounds) {
                        thumb1.anchors.horizontalCenterOffset =
mcx
                        thumb1.anchors.verticalCenterOffset =
mcy
                    } else {
                        var angle = Math.atan2(mcy, mcx)
                        thumb1.anchors.horizontalCenterOffset =
Math.cos(angle) * distanceBound
                        thumb1.anchors.verticalCenterOffset =
Math.sin(angle) * distanceBound
                    }

                    // Fire the signal to indicate the joystick
has moved
                    angle = Math.atan2(signal_y, signal_x)

                    joypose1.publish();
                }

            }
        }

        Image {
            id: thumb1
```

```
            source: "joystick1.png"
            anchors.centerIn: parent

            Rectangle{
                id: joy1
                width: 40
                height: 40
                x: thumb1.width * 0.5
                y: thumb1.width * 0.5
                color: "transparent"

                RosPosePublisher{
                    id: joypose1
                    target: joy1
                    origin: rect1
                    frame: "joy_frame_1"
                    topic: "/joy_pose_1"
                }

            }

            ParallelAnimation {
                id: returnAnimation1
                NumberAnimation   {   target:   thumb2.anchors;
property: "horizontalCenterOffset";
                to:   0;   duration:   200;   easing.type:
Easing.OutSine }
                NumberAnimation   {   target:   thumb2.anchors;
property: "verticalCenterOffset";
                to:   0;   duration:   200;   easing.type:
Easing.OutSine }
                onFinished: joypose2.publish()
            }

            MouseArea {
                id: mouse2
                property  real  fingerAngle  :  Math.atan2(mouseX,
mouseY)
                property int mcx : mouseX - thumb1.width * 0.5
                property int mcy : mouseY - thumb1.height * 0.5
                property bool fingerInBounds : fingerDistance2 <
distanceBound2
                property real fingerDistance2 : mcx * mcx + mcy *
mcy
                property real distanceBound : thumb1.width * 0.5
- thumb2.width * 0.5
                property  real  distanceBound2  :  distanceBound *
distanceBound

                property   double   signal_x   :   (mouseX   -
joystick.width/2) / distanceBound
                property   double   signal_y   :   -(mouseY   -
joystick.height/2) / distanceBound

                anchors.centerIn: parent
                width: thumb2.width
```

```
                height: thumb2.height

                onPressed: {
                    returnAnimation1.stop();
                    i == 2;
                }

                onReleased: {
                    returnAnimation1.restart();
                    i == 1;
                }

                onPositionChanged: {
                    if (fingerInBounds) {
                        thumb2.anchors.horizontalCenterOffset   =
mcx
                        thumb2.anchors.verticalCenterOffset    =
mcy
                    } else {
                        var angle = Math.atan2(mcy, mcx)
                        thumb2.anchors.horizontalCenterOffset   =
Math.cos(angle) * distanceBound
                        thumb2.anchors.verticalCenterOffset    =
Math.sin(angle) * distanceBound
                    }

                    // Fire the signal to indicate the joystick
has moved
                    angle = Math.atan2(signal_y, signal_x)

                    joypose2.publish();

                }
            }

            Image {
                id: thumb2
                source: "joystick2.png"
                anchors.centerIn: thumb1

                Rectangle{
                    id: joy2
                    width: 20
                    height: 20
                    x: thumb2.width * 0.5 + 0.5
                    y: thumb2.width * 0.5 + 0.5
                    color: "transparent"

                    RosPosePublisher{
                        id: joypose2
                        target: joy2
                        origin: joy1
                        frame: "joy_pose_2"
                        topic: "/joy_pose_2"
                    }
```

```
                }
            }

        }

    }

    Button {
        id: gripButton
        y: parent.height-gripButton.height
        width: parent.width

        text: "Open/Close Grip"
        property int order: 1

        RosStringPublisher {
            id: gripPublisher
            topic: "/joy_pose_grip"
        }

        onPressed: {
            gripPublisher.text = gripButton.order
            gripButton.order = gripButton.order * -1

        }

    }

}
```

11.3 Anexo 3: Código en Python del archivo *transfer.py*

```
#!/usr/bin/env python
import rospy
from geometry_msgs.msg import PoseStamped
from std_msgs.msg import Float64
from std_msgs.msg import String


def talker(posx,posy,posx2,posy2,v):
    pub1    =    rospy.Publisher('/bt/joint1_controller/command',
Float64, queue_size=1)
    pub2    =    rospy.Publisher('/bt/joint2_controller/command',
Float64, queue_size=1)
    pub3    =    rospy.Publisher('/bt/joint3_controller/command',
Float64, queue_size=1)
    pub4    =    rospy.Publisher('/bt/joint4_controller/command',
Float64, queue_size=1)
    pub5  =  rospy.Publisher('/bt/joint_pext_controller/command',
Float64, queue_size=1)
    pub6  =  rospy.Publisher('/bt/joint_pint_controller/command',
Float64, queue_size=1)
    rate = rospy.Rate(100) # 50hz



    pub1.publish(posx)
    pub2.publish(posy)
    pub3.publish(posy2)
    pub4.publish(posx2)
    pub5.publish(v*1.025)
    pub6.publish(v)
    rate.sleep()

def callback1(data):
    x = data.pose.position.x/35 * -0.15
    y = data.pose.position.y/35 * 0.15
    rospy.loginfo(rospy.get_caller_id() + "  I heard %f", x)

    try:
       talker(x,y,0,0,0)
    except rospy.ROSInterruptException:
        pass

def callback2(data):
    x = data.pose.position.x/49.5 * 3.86
    y = data.pose.position.y/49.5 * 0.15
    rospy.loginfo(rospy.get_caller_id() + "  I heard %f", x)

    try:
       talker(0,0,x,y,0)
    except rospy.ROSInterruptException:
        pass
```

```
def callback3(msg):

    rospy.loginfo(msg.data)

    v = float(msg.data) * 0.39
    rospy.loginfo(v)

    try:
       talker(0,0,0,0,v)
    except rospy.ROSInterruptException:
        pass

def transfer():

    # In ROS, nodes are uniquely named. If two nodes with the same
    # name are launched, the previous one is kicked off. The
    # anonymous=True flag means that rospy will choose a unique
    # name for our 'listener' node so that multiple listeners can
    # run simultaneously.
    rospy.init_node('listener', anonymous=True)
    i = 1

    rospy.Subscriber("/joy_pose_1", PoseStamped, callback1)
    rospy.Subscriber("/joy_pose_2", PoseStamped, callback2)
    rospy.Subscriber("/joy_pose_grip", String, callback3)

    # spin() simply keeps python from exiting until this node is
stopped
    rospy.spin()

if __name__ == '__main__':
    listener()
```

12 REFERENCIAS

[1] F. J. Bellas Bouza, Á. Deibe Díaz, V. Díaz Casás, R. J. Duro Fernández, M. Lema Rodríguez, F. Lopez Peña y M. Míguez Gonzalez, «"SUBMARINO AUTÓNOMO PARA LA INSPECCIÓN DE INSTALACIONES OFF-SHORE (SAILOR)",» GII - Grupo Integrado de Ingeniería, 2016. [En línea]. Available: https://gii.udc.es/proyectos/detalle/355.

[2] R. I. Luque Moraño, Á. Luis Bustamante, J. M. Molina López, M. A. Patricio Guisada, A. Berlanga de Jesús y J. García Herrero, «"SUBMARINO AUTÓNOMO PARA LA INSPECCIÓN DE INSTALACIONES OFF-SHORE (SAILOR)",» UC3M - Universidad Carlos III Madrid, 2016. [En línea]. Available: http://portal.uc3m.es/portal/page/portal/grupos_investigacion/giaa/links/SAILOR.

[3] Y. Allard y E. Shahbazian, «Unmanned Underwater Vehicle (UUV) Information,» 2014. [En línea]. Available: http://cradpdf.drdc-rddc.gc.ca/PDFS/unc199/p800838_A1b.pdf.

[4] G. Saavedra Soto, «"Modelado Mecánico de Vehículo Submarino Híbrido ROV/AUV". Escuela Politécnica Superior - UDC,» Escola Politécnica Superior - UDC, Julio 2019. [En línea]. Available: https://www.udc.es/es/tfe/traballo/?codigo=15533.

[5] E. M. Corella Solís, «"Diseño y desarrollo de un simulador de código abierto para un robot submarino de propósito general",» TEC - Instituto Tecnológico de Costa Rica, 2019.

[6] A. E. d. N. y. Certifcación, «Robots. Manipuladores industriales. Vocabulario. UNE EN ISO 8373:1998,» AENOR, Madrid, 1998.

[7] R. J. Duro Fernández, "Robótica Industrial", Ferrol: Escuela Politécnica Superior - UDC, 2019.

[8] P. Corke, Robotics, Vision & Control. Second Edition, Springer, 2017.

[9] J. R. Vieites Pardo, “Desarrollo de un modelo de un vehículo aéreo no tripulado sobre simulador de código abierto”, Ferrol: Escola Politécnica Superior - UDC, 2016.

[10] A. García del Valle, "Simulación de Procesos Industriales y Optimización". Escola Politécnica Superior - UDC, Ferrol, 2020.

[11] «IRB120 - Industrial Robots,» Robots Industriales | ABB, [En línea]. Available: https://new.abb.com/products/robotics/es/robots-industriales/irb-120.

[12] S. Edwards, «abb_experimental - ROS Wiki,» Open Source Robotics, [En línea]. Available: http://wiki.ros.org/abb_experimental.

[13] J. Moriano Moratín, «"Trajectory planning for the IRB120 robotic arm using ROS". Escuela Politécnica Superior - Universidad Alcalá,» 2014. [En línea]. Available: https://ebuah.uah.es/dspace/handle/10017/20806.

[14] I. Gracia Ruiz, «"Manipulación robótica colaborativa mediante el Sistema Operativo Robótico". Escuela de Ingeniería y Arquitectura - Universidad de Zaragoza,» 2019. [En línea]. Available: https://deposita.unizar.es/TAZ/EINA/2019/51633/TAZ-TFG-2019-4518.pdf.

[15] «OpenMANIPULATOT-X,» ROBOTIS e-Manual, [En línea]. Available: https://emanual.robotis.com/docs/en/platform/openmanipulator_x/overview/.

[16] D. Lim, «open_manipulator -ROS Wiki,» Open Source Robotics, [En línea]. Available: http://wiki.ros.org/open_manipulator.

[17] G. del Olmo Fernández, «"Simulación en Gazebo de robots móviles para tareas de transporte y manipulación". Escuela de Ingeniería y Arquitectura - Universidad de Zaragoza,» 2019. [En línea]. Available: https://deposita.unizar.es/TAZ/EINA/2019/51814/TAZ-TFG-2019-4665.pdf.

[18] «Robot colaborativo - Universal Robot (UR5),» CFZ Cobots. [En línea]. Available: https://cfzcobots.com/productos/ur5/.

[19] A. Bubeck, S. Edwars y F. Messmer, «ur_gazebo - Ros Wiki,» Open Source Robotics, [En línea]. Available: http://wiki.ros.org/ur_gazebo.

[20] H. Zhao, «"Implementation of UR5 pick and place in ROS-Gazebo with a USB cam and vacuum grippers",» GitHub, [En línea]. Available: https://github.com/lihuang3/ur5_ROS-Gazebo.

[21] C. Boado de la Fuente, «"Aplicación de técnicas de Deep Learning On-Line al aprendizaje de modelos de mundo de robótica cognitiva". Escola Politécnica Superior - UDC,» Septiembre 2019. [En línea]. Available: https://ruc.udc.es/dspace/bitstream/handle/2183/24034/BoadodelaFuente_Carmen_TFG_2019.pdf?sequence=2&isAllowed=y.

[22] Á. Fernández de la Torre, «"Estudio del aprendizaje en tiempo real de modelos de utilidad en robótica cognitiva",» Septiembre 2019. [En línea]. Available: https://ruc.udc.es/dspace/bitstream/handle/2183/24118/FernandezdelaTorre_Alvaro_TFM_2019.pdf?sequence=2&isAllowed=y.

[23] A. Ramil Rego, "Mecánica", Ferrol: Escola Politécnica Superior - UDC, 2017.

[24] J. Denavit y R. Hartenberg, A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices. Journal of Applied Mechanics, 1955.

[25] A. A. Shaban, «Computational Dynamics. Wiley-Interscience.,» 2001, p. 379.

[26] J. Legarreta y R. Martínez, "Tema 6. Dinámica de Robots y Control", OCW - Universidad del Pais Vasco.

[27] C. Renán Acosta, «"Obtención de la ecuación de Euler-Lagrange utilizando los vectores base y vectores recíprocos",» 2004.

[28] O. E. Ramos, "Cinemática Diferencial de Robots Manipuladores, UTEC, 2017.

[29] V. Díaz Casás, R. J. Duro Fernández, A. García del Valle, F. López Peña, J. D. Pena Agras, M. Naya Varela, S. Ferreño González, A. Paz López, R. Salgado Terneiro, V. Soñora Pombo, D. Souto García, P. J. Trueba Martínez, G. Varela Fernández y M. Míguez González, «"A-TEMPO: AVANCES EN TECNOLOGÍAS MARINAS - PRODUCCIÓN NAVAL Y OFFSHORE",» GII - Grupo Integrado de

Ingeniería, 2014. [En línea]. Available: https://www.gii.udc.es/proyectos/detalle/337.

[30] «Anexo II al Documento descriptivo. "Necesidades y especificaciones tecnicas". Expte. Núm. 2015/4005.,» https://udc.es/export/sites/udc/contratacionadministrativa/_galeria_down/consulta_licitacions/2015/2015_CPP_DC/2015_4005_ESPECIF_TECNICAS.pdf_2063069294.pdf, 2015.

[31] «ROS/Introduction,» Open Source Robotics Foundation, [En línea]. Available: http://wiki.ros.org/ROS/Introduction.

[32] A. Romero Montero y A. M. Mallo Casdelo, "Robótica Industrial", Ferrol: Grupo Integrado de Ingeniería - UDC, 2019.

[33] «ROS - Wiki,» Open Robotics Foundation, [En línea]. Available: http://wiki.ros.org.

[34] «Q&A answers.ros.org,» Open Source Robotics Foundation, [En línea]. Available: https://answers.ros.org/.

[35] «ROS/Concepts,» Open Robotics Foundation, [En línea]. Available: http://wiki.ros.org/ROS/Concepts.

[36] «urdf/XML,» Open Source Robotics Foundation, [En línea]. Available: http://wiki.ros.org/urdf/XML.

[37] «xacro,» Open Source Robotics Foundation, [En línea]. Available: http://wiki.ros.org/xacro#Property_and_Property_Blocks.

[38] «Gazebo,» [En línea]. Available: http://gazebosim.org/.

[39] «Gazebo : Tutorials,» [En línea]. Available: http://gazebosim.org/tutorials.

[40] «Gazebo : Tutorial : ROS Overview,» [En línea]. Available: http://gazebosim.org/tutorials?tut=ros_overview&cat=connect_ros.

[41] «Gazebo : Tutorial : URDF in Gazebo,» [En línea]. Available: http://gazebosim.org/tutorials?tut=ros_urdf&cat=connect_ros.

[42] «Alternative downloads,» Ubuntu |, [En línea]. Available: https://ubuntu.com/download/alternative-downloads.

[43] «Distributions,» Open Source Robotics Foundation, [En línea]. Available: http://wiki.ros.org/Distributions.

[44] «Gazebo : Tutorial : Which combination of ROS/Gazebo,» [En línea]. Available: http://gazebosim.org/tutorials/?tut=ros_wrapper_versions.

[45] «melodic/Installation/Ubuntu,» Open Source Robotics Foundation, [En línea]. Available: http://wiki.ros.org/melodic/Installation/Ubuntu.

[46] «sw_urdf_exporter,» Open Source Robotics Foundation, [En línea]. Available: http://wiki.ros.org/sw_urdf_exporter.