Control y operación de un brazo robótico virtual mediante Robot Operating System (ROS)

Juan Sebastian Piratova Silva

Universidad Antonio Nariño

Facultad de Ingeniería Mecánica, Electrónica y Biomédica

Bogotá, Colombia

2020

Control y operación de un brazo robótico virtual mediante Robot Operating System (ROS)

Juan Sebastian Piratova Silva

Proyecto de grado presentado como requisito parcial para optar al título de:

Ingeniero Electrónico

Director:

Ing. Julián David Pareja Garzón MSc

Línea de Investigación:

Robótica

Universidad Antonio Nariño

Facultad de Ingeniería Mecánica, Electrónica y Biomédica

Bogotá, Colombia

2020

A la persona que más quise en la vida, quien lastimosamente no pudo verme alcanzar este logro, gracias por todo tu amor.

Agradecimientos

A Dios, por las segundas oportunidades concedidas, por la vida, la guía y la valentía.

A mi tutor Julián D. Pareja, por compartir sus conocimientos, por su disposición constante y por toda la ayuda prestada.

A mis padres, por su paciencia, su amor y su apoyo incondicional, que me han sido brindados a lo largo de la vida, y sé que siempre podré contar con ellos.

Resumen

El presente trabajo de grado partió del objetivo de operar un brazo robótico virtual, terminando en el desarrollo de un entorno de simulación. Esto se realizó con el fin de fortalecer el aprendizaje en el área de la robótica de los futuros ingenieros al utilizar herramientas de código abierto, facilitando así su acceso desde cualquier lugar con tener al alcance un computador.

Se presentó la creación de un brazo antropomórfico desde cero, por lo que se realizó la descripción teniendo en mente que la estructura podría construirse físicamente en el futuro para la creación de un laboratorio, Igualmente se modeló en 3D en el simulador de Gazebo, donde se determinó su comportamiento mediante el framework ROS, desarrollándose una serie de comandos básicos basados en la comunicación entre nodos para este fin. Finalmente se desarrolló una interfaz gráfica en el software Qt capaz de manipular el movimiento de las articulaciones a partir del ingreso del desplazamiento deseado en grados, lo cual al implementarse simplificó notablemente la operación del robot.

Los lenguajes de programación utilizados en el desarrollo del proyecto fueron XML para realizar la descripción del robot junto al mundo de simulación en Gazebo, y C++ para establecer el comportamiento de la estructura e igualmente para crear su interfaz gráfica de operación. Como resultado de este proyecto se obtuvo el diseño tanto en 2D como en 3D del manipulador antropomórfico, un conjunto de scripts capaces de manipular el comportamiento del robot, y una interfaz gráfica para facilitarle al usuario su maniobrabilidad.

Palabras clave: robótica, diseño, simulación, operación, ROS, Gazebo.

Abstract

The present degree work started from the objective of operating a virtual robotic arm, ending in the development of a simulation environment. This was done in order to strengthen the learning in the robotics area of future engineers by using open source tools, thus facilitating their access from anywhere with a computer within reach.

The creation of an anthropomorphic arm from scratch was presented, so the description was made keeping in mind that the structure could be physically built in the future for the creation of a laboratory, It was also modeled in 3D in the Gazebo simulator, where determined its behavior using the ROS framework, developing a series of basic commands based on communication between nodes for this purpose. Finally, a graphical interface was developed in the Qt software capable of manipulating the movement of the joints from the input of the desired displacement in degrees, which when implemented significantly simplified the operation of the robot.

The programming languages used in the development of the project were XML to make the description of the robot together with the simulation world in Gazebo, and C ++ to establish the behavior of the structure and also to create its graphical operating interface. As a result of this project, the anthropomorphic manipulator was designed in both 2D and 3D, a set of scripts capable of manipulating the robot's behavior, and a graphical interface to facilitate its maneuverability for the user.

Keywords: robotics, design, simulation, operation, ROS, Gazebo.

Contenido

Pág.

Resumen ..... **IX**

Lista de figuras ..... **XIII**

Lista de tablas ..... **XIV**

Lista de abreviaturas ..... **XV**

1. Introducción **17**
1.1 Estado del arte 18
1.2 Planteamiento del problema 22
1.3 Justificación 23
1.4 Objetivos 23
1.4.1 Objetivo general 23
1.4.2 Objetivos específicos 23

2. Marco teórico **25**
2.1 Robots manipuladores 25
2.1.1 Brazo antropomórfico 26
2.2 Ubicación espacial 27
2.2.1 Transformaciones homogéneas 29
2.2.2 Parámetros D-H 29
2.2.3 Cinemática directa 31
2.2.4 Cinemática inversa 32
2.2.5 Robotics Toolbox 34
2.3 Ubuntu 35
2.4 ROS 35
2.4.1 Estructura 36
2.4.2 Arquitectura 37
2.4.3 Distribuciones 40
2.5 Gazebo 41
2.5.1 Formato 41
2.6 Qt 42

3. Metodología **43**
3.1 Diseño del brazo 44
3.1.1 Cinemática directa del diseño 44
3.2 Diseño 3D del brazo 46
3.2.1 Diseño en Gazebo 47
3.2.2 Simulación 50

3.3 Programación del brazo 50
3.3.1 Programación en ROS 51
3.3.2 Simulación 52
3.4 Diseño de la Interfaz gráfica de operación en Qt 54

4. Resultados 55

5. Conclusiones y recomendaciones 59
5.1 Conclusiones 59
5.2 Recomendaciones 60

A. Anexo: Algoritmos esenciales de ROS 61
A1. Propiedades del paquete (package.xml) 61
A2. Compilador de CMake (CMakeLists.txt) 62
A3. Lanzador de ROS (brazo.launcher) 63

B. Anexo: Algoritmos del mundo 65
B1.1. Configuración del modelo importado (model.config) 65
B1.2. Simulación del modelo importado (model.sdf) 65
B2. Simulación del mundo (brazo.world) 66

C. Anexo: Algoritmos de funcionamiento del robot 79
C1.1. Script principal (brazo.h) 79
C1.2. Script principal (brazo.cpp) 80
C2.1. Enlace IPO (listener.h) 84
C2.2. Enlace IPO (listener.cpp) 85
C3.1. Instrucciones (comandos.h) 87
C3.2. Instrucciones (comandos.cpp) 88

D. Anexo: Algoritmos de la interfaz gráfica 91
D1. Compilador de CMake (CMakeLists.txt) 91
D2.1. Script interfaz gráfica (main.cc) 91
D2.2. Script interfaz gráfica (mainDialog.h) 92
D2.3. Script interfaz gráfica (mainDialog.cc) 94

Bibliografía 99

Lista de figuras

Pág.

Figura 1-1: Simulador robótico en uso por la población de estudio. 19
Figura 1-2: Estructura del sistema de visualización robótica. 20
Figura 1-3: Estructura del sistema de control. 21
Figura 2-1: Componentes de un sistema robótico. 25
Figura 2-2: Tipos de articulaciones. 26
Figura 2-3: Esquema de brazo antropomórfico. 27
Figura 2-4: Posición y orientación de un cuerpo rígido. 27
Figura 2-5: Extracción de matrices básicas de rotación de un cuerpo rígido. 28
Figura 2-6: Cadena cinemática de un robot antropomórfico. 31
Figura 2-7: Matriz jacobiana de un manipulador de 6 grados de libertad. 33
Figura 2-8: Columna del Jacobiano. 33
Figura 2-9: Simulación de robot antropomórfico. 34
Figura 2-10: Comunicación entre nodos (mensajes). 38
Figura 2-11: Interfaz de Gazebo. 42
Figura 3-1: Diseño del brazo robot. 44
Figura 3-2: Simulación del diseño del brazo robot. 45
Figura 3-3: Procedimiento de la creación del mundo. 47
Figura 3-4: Simulación 1 del mundo. 48
Figura 3-5: Simulación 2 del mundo. 50
Figura 3-6: Procedimiento de creación de los scripts del funcionamiento del brazo. 51
Figura 3-7: Simulación de secuencia de movimientos. 53
Figura 3-8: **I**nterfaz gráfica de operación. 54
Figura 4-1: Prueba del elemento LineEdit. 56
Figura 4-2: Prueba del elemento Slider. 56
Figura 4-3: Prueba del elemento PushButton de tareas. 57
Figura 4-4: Prueba del elemento PushButton de home. 57

Lista de tablas

Pág.

Tabla 2-1: Parámetros de la matriz D-H (información adaptada de [14]).....30
Tabla 2-2: Matriz D-H de un robot antropomórfico (información adaptada de [14]). 31
Tabla 2-3: Comandos básicos de Ubuntu. 35
Tabla 2-4: Estructura de paquetes (información adaptada de [21]). 36
Tabla 2-5: Administración de paquetes (información adaptada de [21]). 36
Tabla 2-6: Navegación de paquetes (información adaptada de [21]). 37
Tabla 2-7:: Administración de nodes (información adaptada de [21]). 38
Tabla 2-8: Administración de topics (información adaptada de [21]). 39
Tabla 2-9: Administración de messages (información adaptada de [21]). 39
Tabla 2-10: Administración de services (información adaptada de [21]). 39
Tabla 2-11: Distribuciones de ROS (información adaptada de [21]). 40
Tabla 2-12: Comandos básicos de Qt. 42
Tabla 3-1: Matriz D-H (elaborada por el autor). 45

Lista de abreviaturas

Abreviatura	Término
CAD	Computer-Aided Design
arg	argument
exe	executable
IPO	Interacción Persona-Ordenador
LHC	Large Hadron Collider
LTS	Long Term Support
msg	message
OSRF	Open Source Robotics Foundation
pkg	package
PLC	Point Cloud Library
ROS	Robot Operating System
SIAR	Sewer Inspection Autonomous Robot
srv	service
TIM	Train for RP survey and visual inspection in LHC
UDF	Simulation Description Format
URI	Uniform Resource Identifier

1. Introducción

Desde los inicios de la humanidad, el hombre siempre ha intentado mejorar su calidad de vida, llegando a crear con este fin, desde dispositivos sencillos, maquinas complejas, hasta mecanismos autónomos. Estos últimos, creados a partir de la unión de diferentes disciplinas, tales como: la mecánica, electrónica e informática, las cuales abarcan la física, ingeniería de control e inteligencia artificial, dando origen a la robótica, rama de la ingeniería donde se encuentra ubicado el presente proyecto, el cual es abordado a lo largo de la propuesta.

El desarrollo del proyecto partió de la posibilidad de predecir el comportamiento de un sistema frente posibles acontecimientos que puedan suceder, cuyo objetivo es encontrar el algoritmo que mejor se adapte al desarrollo de aplicaciones robóticas, e igualmente, detectar posibles problemas que requieran solución al simular la estructura en el ambiente a estar. De esta manera, se observa que la realización de este ejercicio ayuda a minimizar el tiempo y los costos necesarios para el desarrollo y evaluación de un sistema, siendo los simuladores robóticos la herramienta más importante a tener en cuenta. Por ello, junto con la asistencia de los frameworks se pretende que las personas no deban partir de cero a la hora de concebir un robot y puedan desarrollarlo con plataformas de bajo costo mostrándose como uno de los problemas más habituales.

En base de lo anteriormente mencionado, la metodología del proyecto fue desarrollada a partir del uso del framework ROS, el cual, con su conjunto de librerías y herramientas, facilitan la creación de aplicaciones robóticas, desde controladores hasta complejos algoritmos de manera sencilla y robusta [1]. Asimismo, la compatibilidad que tiene con la herramienta de simulación Gazebo, permitiéndole desde diseñar estructuras robóticas básicas, hasta establecer determinadas características sobre un sistema y entorno, con el fin de evaluar su comportamiento y corregir posibles fallas que se puedan presentar; convirtiendo esta combinación de herramientas en las mejores a tener en cuenta a la hora

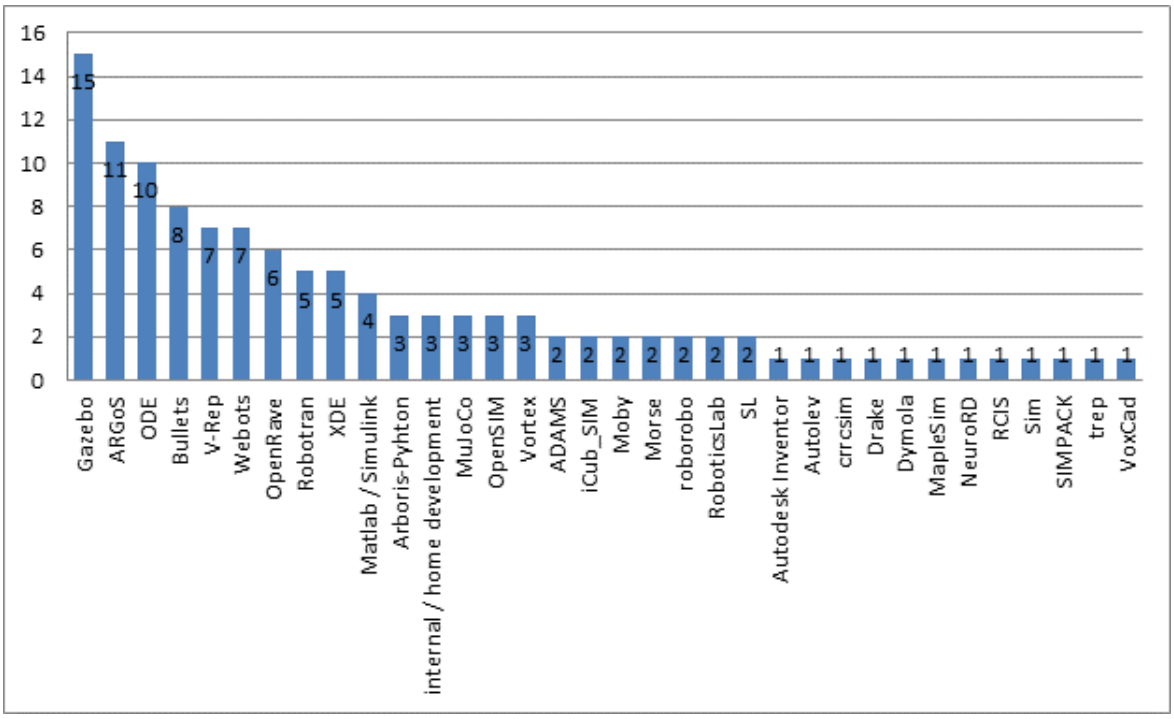
de pensar en crear un robot [2, 3]. Lo anterior en concordancia con el objetivo principal planteado para este proyecto el cual es: control y operación de un brazo robótico virtual mediante ROS, siendo esto logrado por medio del diseño de un brazo robótico en un entorno 3D, así como del uso de ROS para la programación de una serie de algoritmos y el diseño de una interfaz gráfica, con el fin de manipular el brazo virtual, para finalmente validar la integración total del sistema variando su movimiento en tiempo real en el entorno grafico por medio de sliders. Consecuentemente, se debe aclarar que en este trabajo no se abordó la implementación física del sistema desarrollado, y el proceso fue realizado enteramente en el sistema operativo Ubuntu, con lo cual los softwares necesarios son de tipo libre.

Con el fin de lograr lo establecido en el objetivo propuesto se presentan varias etapas las cuales son las siguientes: diseño del brazo, diseño 3D del brazo, programación del brazo y diseño de la interfaz gráfica de operación, además de la realización de simulaciones entre las etapas para verificar su correcto comportamiento.

1.1 Estado del arte

La Ingeniería Robótica es la ciencia interdisciplinaria que estudia el análisis, diseño, manufactura y aplicación de los mecanismos autónomos capaces de realizar tareas específicas [4]. Una herramienta de gran utilidad en el campo de la robótica son los simuladores, los cuales, con al menos el diseño de un mecanismo y las leyes de la física, puede ponerse a prueba el comportamiento de un robot en un ambiente realista. Aunque actualmente sigue en aumento la variedad al igual que su versatilidad, no existe alguno que sobresalga notablemente sobre los demás en rendimiento y aplicación, si no que estos depende de preferencias individuales como se indica en un estudio realizado [5], donde los autores comparan múltiples características de diferentes simuladores y evalúan la preferencia que tienen las personas hacia estos, arrojando el resultado de una tendencia por el uso de herramientas de código abierto, como se observa en la Figura 1-1, e igualmente se observa a Gazebo entre los más utilizados.

Figura 1-1: Simulador robótico en uso por la población de estudio.



Nombre de la fuente: Tomada de [5].

Por otra parte, se encuentran los frameworks que ayudan al desarrollo de software especializado o con características previamente definidas, que, gracias a su compilación de librerías, herramientas de desarrollo, soporte multiplataforma y multilenguaje, facilitan la integración de diferentes sistemas y minimizan el esfuerzo necesario para el desarrollo de algoritmos complejos debido a su capacidad de reutilizar fragmentos de otros fácilmente. Según un estudio realizado [6], donde se comparan estas herramientas con énfasis en la robótica, concluyó que la más utilizada era ROS, esto debido principalmente al amplio repositorio que posee apoyado por su gran comunidad.

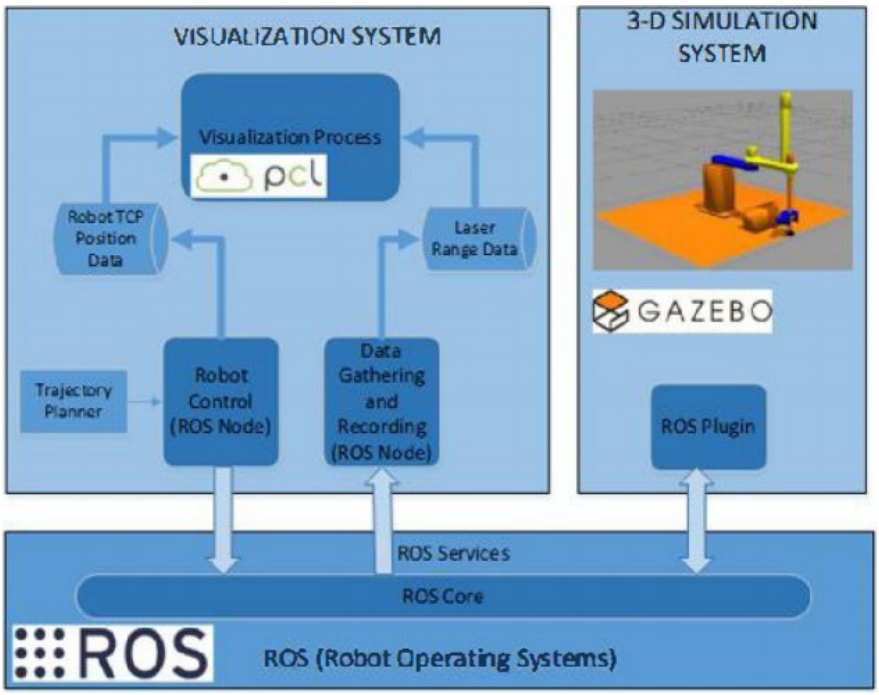
Asimismo, en el estudio [3], se detalla como actualmente la administración de los dos softwares precitados es llevada a cabo por la OSRF, logrando complementarse mutuamente, formando así una valiosa herramienta y perfilándose como una opción mundial para el avance de la robótica, gracias a su constante desarrollo y evolución.

La combinación de estas herramientas de código abierto ha demostrado tener un gran impacto en la educación como se demuestra en el proyecto [7], donde se plantea el desarrollo de un entorno virtual que permita probar robots móviles en diferente entorno de

una manera sencilla, gracias a un conjunto de scripts desarrollados en Python y C++ para controlarlos con la integración de ROS junto a Gazebo y MORSE.

Dirigidos hacia la industria también ha habido estudios con el fin de mejorar el desempeño de los robots como es el caso de la investigación [8], que presenta la estructura observada en la Figura 1-2. Esta implementación del sistema gira alrededor de un entorno de simulación, donde un manipulador industrial SCARA con la ayuda de un sensor laser genera un esquema de trayectorias y realiza la recolección de datos en función de las coordenadas del efector final y el ángulo de orientación del brazo, obteniendo de esta manera un archivo en formato PCL, el cual es necesario para la reconstrucción tridimensional del objeto.

Figura 1-2: Estructura del sistema de visualización robótica.

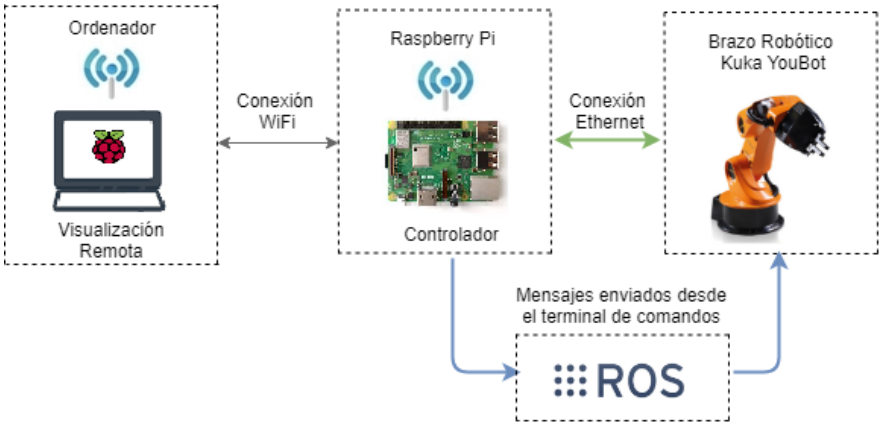


Nombre de la fuente: Tomada de [8].

De la misma manera, en el proyecto [9], donde se presenta la opción de controlar robots industriales y de investigación con sistemas embebidos utilizando la estructura de la Figura 1-3, con el fin de evitar problemas de compatibilidad de hardware gracias a que las diferentes tarjetas cuentan con una amplia variedad de librerías y paquetes. Para esto se

crea un entorno virtual con características físicas reales para un robot manipulador Kuka YouBot el cual utilizará un sistema de control desarrollado para una tarjeta Raspberry Pi.

Figura 1-3: Estructura del sistema de control.



Nombre de la fuente: Tomada de [9].

Igualmente, en el trabajo [10], en el que se pretende controlar un brazo robótico LWA 4P a través del brazo del operario utilizando un sensor Kinect de Microsoft. Para esto se crea un entorno de simulación en Gazebo con una interfaz gráfica para la operación del sistema, donde se recibirán las coordenadas generadas a partir del procesamiento de datos de un paquete de algoritmos desarrollado en ROS que permite la comunicación con el Kinect.

Por otro lado, es posible encontrar investigaciones especializadas, como es el caso de [11], en la que se propone simular un robot SIAR, considerando las características mecánicas y dinámicas reales, además de las condiciones adversas del alcantarillado. Para llevar a cabo este proyecto, se crea el entorno de simulación en Gazebo gracias a una representación del entorno real obtenida por el robot que es teleoperado, además de desarrollar plugins en ROS para la comunicación y manipulación de cada uno de los elementos, con el fin de poder controlarlos con un mando a distancia. Finalmente, considerando que los resultados obtenidos no fueron completamente fieles a la realidad debido a un movimiento anómalo de uno de los ejes del vehículo, si se consiguió una buena aproximación de base, que al mejorar la simulación sería posible el pensar en diseñar un controlador que automatice en parte o completamente las acciones del robot.

Por último, está el trabajo [12], que propone aumentar la funcionalidad del robot móvil TIM, añadiéndole la capacidad de realizar tareas de mantenimiento, las cuales, debido a la

radiación, el acceso al personal es limitado y estas funciones son realizadas por robots móviles teleoperados mientras el ambiente se va estabilizando luego de los experimentos. Para esto se analizan opciones de integración de un manipulador robótico de 6 grados de libertad en uno de sus vagones, y se establecen unas pautas deseables, como longitud, campo de movimiento y capacidad motriz, suficientes para realizar tareas de inspección y mantenimiento sencillas. Así mismo, para verificar su viabilidad se desarrolla un entorno de simulación donde se pone a prueba el sistema implementado, el cual termina cumpliendo con las expectativas iniciales, con la posibilidad de mejorarse.

Con estas investigaciones como muestra se puede observar el trabajo hecho en diferentes áreas, haciendo uso de plataformas robóticas en beneficio de generar más investigación y avances tecnológicos, con el fin de solucionar problemas que puedan surgir al momento de querer implementar o manipular un robot, así como facilitar su uso y disminuir el riesgo al teleoperarlos en un ambiente adverso, desde un lugar seguro.

1.2 Planteamiento del problema

Con el paso del tiempo el nivel tecnológico ha ido aumentando, en particular para el área de la robótica, donde poco a poco el uso de los robots se ha ido extendiendo en diferentes ámbitos, gracias a que son capaces de realizar tareas pesadas, repetitivas, delicadas, etc., sin llegar a desgastarse y manteniendo la precisión, conllevado al reemplazo de personas en la ejecución de actividades riesgosas, complejas o incluso por comodidad.

Por otra parte, a pesar de que el desarrollo de los robots puede llevarse a cabo con multitud de herramientas, por lo general tienen costos elevados las más comunes, y las que son de código abierto no son muy populares por lo que son poco conocidas, causando que se genere poca incursión en esta rama de la ingeniería.

En base a lo anterior, existe la necesidad de fortalecer el aprendizaje en el área de la robótica, de forma que se logre afianzar la teoría con la práctica, pero ya que no todas las instituciones pueden contar con laboratorios especializados, donde el costo de la adquisición de equipo y software puede no ser el más accesible, una opción viable es la creación y el fortalecimiento de laboratorios virtuales.

1.3 Justificación

Existen muchos tipos de robots, pero pocas personas conocen sobre su desarrollo en ROS, el cual es un framework que ayuda a crear algoritmos robustos de manera sencilla, que en conjunto con la integración de la herramienta de simulación Gazebo, permite diseñar modelos en 3D, establecer su comportamiento y lograr controlarlos con el fin de observar la interacción de la estructura en un ambiente deseado, revelando de esta forma posibles problemas en el diseño y la programación con el propósito de corregirlos a tiempo antes de realizar una implementación física.

Partiendo de lo anterior, se estableció este proyecto con el objetivo de diseñar un entorno de simulación con la ayuda de las herramientas previamente mencionadas, a partir de la concepción de un brazo robótico, el cual al ser creado desde cero se garantiza la operabilidad y funcionalidad, no solo de la estructura inicial, si no de cualquier tipo de robot que se desee añadir en el futuro. Esto será de ayuda a la Universidad Antonio Nariño a fortalecer la enseñanza del área de la robótica, al ofrecer conocimientos un poco más prácticos, los cuales pueden ser usados para la construcción y fortalecimiento de laboratorios virtuales.

1.4 Objetivos

1.4.1 Objetivo general

Control y operación de un brazo robótico virtual mediante el sistema operativo ROS.

1.4.2 Objetivos específicos

- Diseñar un brazo robótico en un entorno 3D.
- Programar una serie de algoritmos en ROS, para controlar el movimiento del brazo virtual.
- Diseñar e implementar una interfaz gráfica en ROS.
- Validar movimientos del brazo robot en tiempo real usando el entorno grafico con sliders.

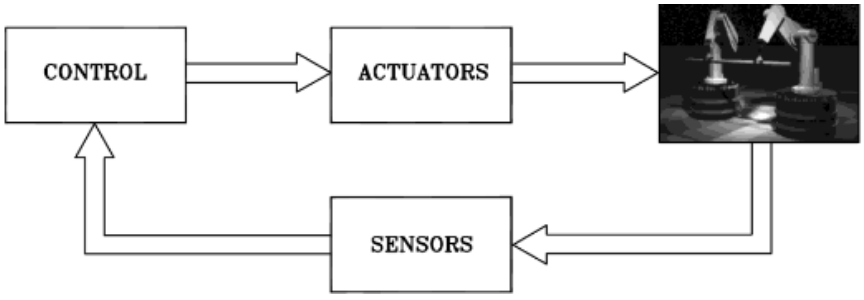
2. Marco teórico

En este capítulo se abordaron los conceptos utilizados a lo largo de la elaboración del proyecto, e igualmente se mencionaron los programas utilizados. Los conceptos de robots manipuladores y ubicación espacial fueron la base para la elaboración del diseño del robot y una posterior posible mejora del sistema e implementación física. Por otro lado, Ubuntu fue la base del proyecto, al tener la capacidad de correr Gazebo para el diseño del robot en 3D, así como ROS para la programación de su comportamiento y finalmente Qt para la elaboración de una interfaz gráfica con el fin simplificar la tarea de operación.

2.1 Robots manipuladores

Su estructura se compone de una secuencia de cuerpos rígidos (links), interconectados por articulaciones (joints), formando de esta manera un brazo que garantiza movilidad y una muñeca que confiere destreza para controlar la orientación del efector final, quien se encarga de la tarea a realizar [13]. De forma general un robot manipulador está conformado por los componentes que se observan en la Figura 2-1.

Figura 2-1: Componentes de un sistema robótico.



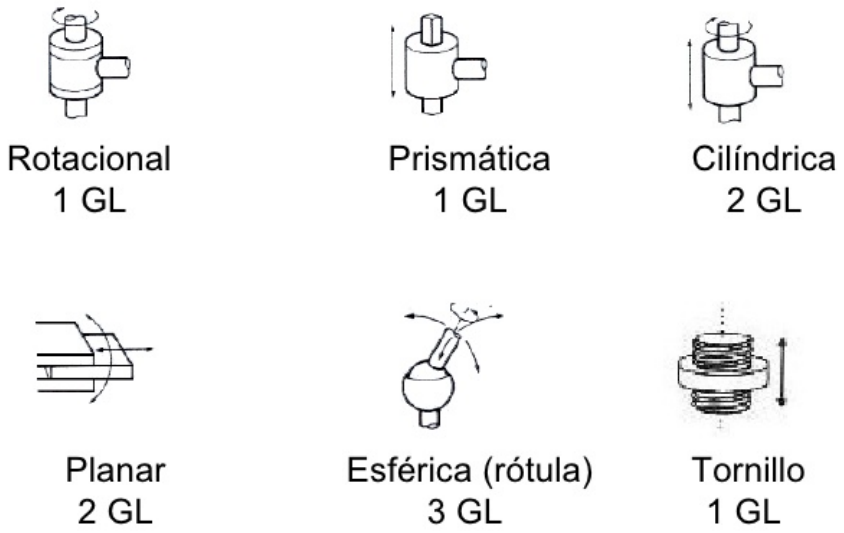
Nombre de la fuente: Tomada de [14].

- Consola de control: Se compone de un sistema electrónico con la etapa de potencia encargada de suministrar energía al robot para su movimiento, además de incluir

los algoritmos de control y una interfaz necesaria para que el usuario se comunique con el robot a través de instrucciones de programación.

- Actuadores: Suministran las señales necesarias a las articulaciones para producir movimientos.
- Manipulador: Se constituye por eslabones interconectados por articulaciones, como se observa en la Figura 2-2, las cuales están formadas por servomotores que permiten el movimiento relativo entre ellos. Igualmente cuenta con un efector final, el cual será una herramienta que puede variar según la tarea asignada.
- Sensores: Proporcionan información del estado interno el robot.

Figura 2-2: Tipos de articulaciones.

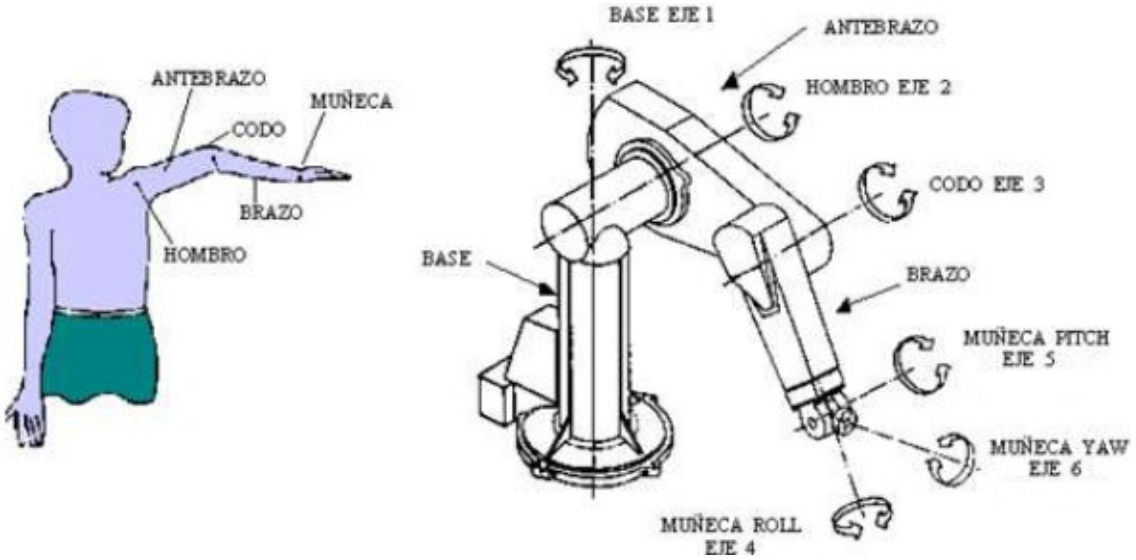


Nombre de la fuente: Tomada de [15].

2.1.1 Brazo antropomórfico

Este tipo de robots posee una configuración de tres articulaciones rotacionales para posicionamiento (primer eje perpendicular al piso, segundo y tercero perpendiculares al primero y paralelos entre sí) y por lo general otras tres de orientación para el efector final [16], como se observa en la Figura 2-3. Igualmente, hay que destacar que gracias a esta estructura el manipulador adquiriere una gran maniobrabilidad en lugares con obstáculos de manera rápida, permitiendo la ejecución de complejas trayectorias [16].

Figura 2-3: Esquema de brazo antropomórfico.

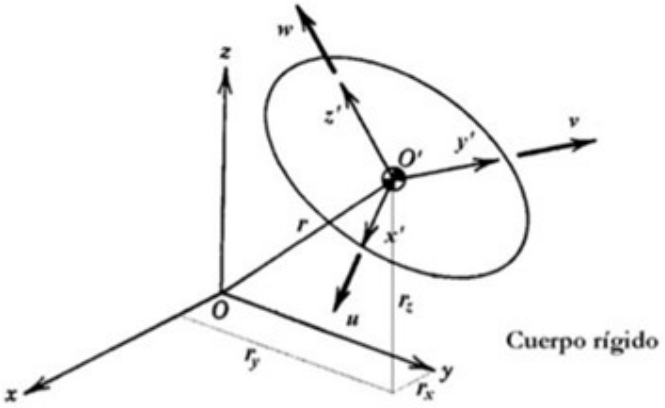


Nombre de la fuente: Tomada de [17].

2.2 Ubicación espacial

Todo movimiento realizado por un robot necesita de una ubicación espacial. Para describir la orientación de un objeto, se asigna un sistema de coordenadas ligado al cuerpo y luego se describe la rotación espacial entre dicho sistema de coordenadas (O') y el sistema de coordenadas de referencia (O). Como se observa en la Figura 2-4, O_{xyz} es el marco de referencia ortogonal y x, y, z los vectores unitarios del marco de ejes, dando como resultado la posición en el espacio de O' con respecto a O (1) y la matriz de rotación (2).

Figura 2-4: Posición y orientación de un cuerpo rígido.



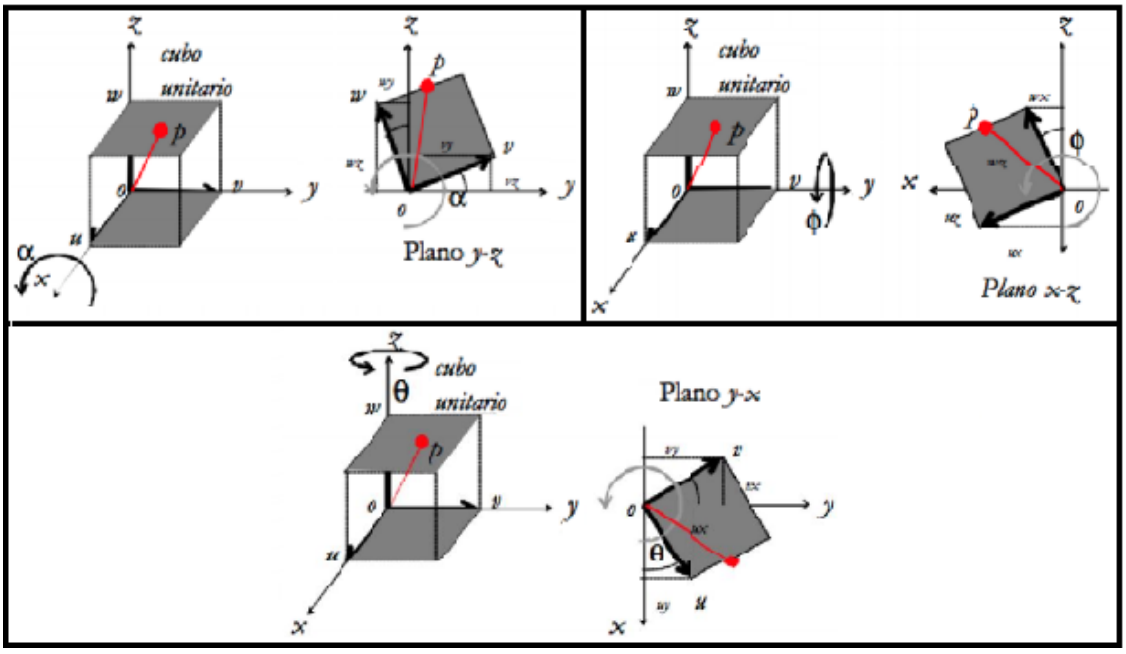
Nombre de la fuente: Tomada y adaptada de [14].

$$r = \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} \quad (1) \qquad R = \begin{bmatrix} x'_x & y'_x & z'_x \\ x'_y & y'_y & z'_y \\ x'_z & y_{,z} & z'_z \end{bmatrix} \quad (2)$$

A partir de (2) se hallan las matrices básicas de rotación de un cuerpo rígido (3) (4) (5), donde R es el sistema de coordenadas P_{uvw} girado a grados con respecto al sistema fijo de referencia P_{xyz} como se muestra en la Figura 2-5. Igualmente, si se desea realizar una rotación compuesta se deben seguir las siguientes reglas:

- Definir una matriz identidad 3x3 para representar la coincidencia entre los dos sistemas de coordenadas.
- Si el sistema de coordenadas en movimiento O_{uvw} gira respecto de uno de los ejes principales del sistema de referencia O_{xyz}, premultiplicar la matriz de rotación previa resultante por la matriz de rotación básica correspondiente a dicho giro.
- Si el sistema de coordenadas en movimiento O_{uvw} gira respecto de alguno de sus ejes, postmultiplicar la matriz de rotación previa resultante por la matriz de rotación básica correspondiente a dicho giro.

Figura 2-5: Extracción de matrices básicas de rotación de un cuerpo rígido.



Nombre de la fuente: Tomada y adaptada de [14].

$$R_{x,\alpha} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} (3)$$

$$R_{y,\phi} = \begin{bmatrix} \cos\phi & 0 & \sin\phi \\ 0 & 1 & 0 \\ -\sin\phi & 0 & \cos\phi \end{bmatrix} (4)$$

$$R_{z,\theta} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} (5)$$

2.2.1Transformaciones homogéneas

Una matriz de transformación homogénea (A) relaciona un sistema de coordenadas $i-ésimo$ con el sistema de coordenadas $(i-1)-ésimo$, representando la posición y orientación de un sistema con respecto a un sistema fijo, dando como resultado la ecuación (6), la cual variará si son articulaciones rotacionales (7) o prismáticas (8), donde fueron simplificaron los términos por comodidad de visualización.

$$A_i^{i-1} = R_{z,\theta}T_{z,d}T_{x,a}R_{x,\alpha} \ (6)$$

$$A_i^{i-1} = \begin{bmatrix} c\theta_i & -c\alpha_i s\theta_i & s\alpha_i s\theta_i & a_i c\theta_i \\ s\theta_i & c\alpha_i c\theta_i & -s\alpha_i c\theta_i & a_i s\theta_i \\ 0 & s\alpha_i & c\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} (7)$$

$$A_i^{i-1} = \begin{bmatrix} c\theta_i & -c\alpha_i s\theta_i & s\alpha_i s\theta_i & 0 \\ s\theta_i & c\alpha_i c\theta_i & -s\alpha_i c\theta_i & 0 \\ 0 & s\alpha_i & c\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} (8)$$

2.2.2Parámetros D-H

Es un modelo matricial sistemático, que establece un sistema de coordenadas (ligado al eslabón) para cada elemento de una cadena articulada, obteniendo una matriz de transformación homogénea 4x4 a partir de la extracción de las características de la estructura, donde se representa cada uno de los sistemas de coordenadas con respecto

al sistema del elemento previo, indicando de esta forma las limitaciones físicas del robot. Para establecer el sistema de coordenadas se debe seguir la siguiente lista de pasos [18]:

- Localizar el eje z_i de las articulaciones (rotativa = eje de giro, prismática = eje de desplazamiento) y ubicarlo en el eje de la articulación $i+1$.
- Colocar el sistema de coordenadas 0.
- Situar el sistema en la intersección del eje z_i con la línea normal común a z_{i-1} y z_i.
  - Si ambos ejes se cortan, s_i se coloca en el punto de corte.
  - Si son paralelos, s_i se coloca en la articulación $i+1$.
- Colocar x_i en la línea normal común a z_{i-1} y z_i.
- Colocar y_i de modo que forme un sistema dextrógiro con x_i y z_i.
- Colocar el sistema s_n en el extremo del robot de modo que z_n coincida con la dirección de z_{n-1} y x_n sea normal a z_{i-1} y z_n.

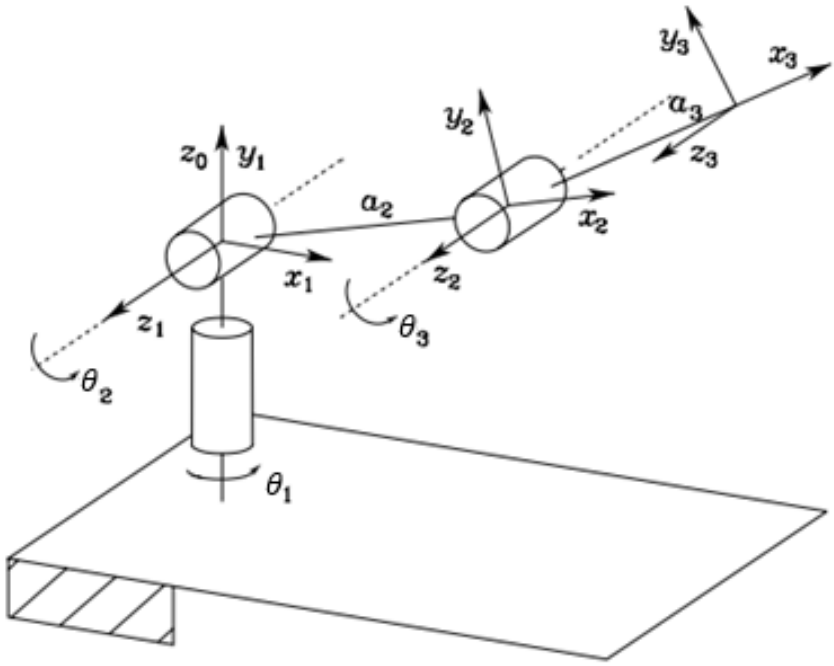
Habiendo ubicado sistemas de coordenadas se procede a construir la matriz D-H a partir de los parámetros observados en la Tabla 2-1.

Tabla 2-1: Parámetros de la matriz D-H (información adaptada de [14]).

Parámetro	Articulación	Descripción
θ_i	Rotacional	Ángulo de la articulación del eje x_{i-1} al eje x_i respecto del eje z_{i-1}
α_i		Ángulo de separación del eje z_{i-1} al eje x_i respecto del eje x_i
d_i	Rotacional	Distancia desde el origen del sistema de coordenadas $(i-1)-ésimo$ hasta la intersección del eje z_{i-1} con el eje x_i a lo largo del eje z_{i-1}
	Prismática	Parámetro variable.
a_i	Rotacional	Distancia desde la intersección del eje z_{i-1} con el eje x_i hasta el origen del sistema $i-ésimo$ a lo largo del eje x_i
	Prismática	Distancia más corta entre los ejes z_{i-1} y z_i

A continuación, se presenta el ejemplo de un manipulador antropomórfico de tres grados de libertad, donde se establece un sistema de coordenadas para cada eslabón en la Figura 2-6, y se extraen los parámetros D-H, obteniendo la Tabla 2-2.

Figura 2-6: Cadena cinemática de un robot antropomórfico.



Nombre de la fuente: Tomada de [14].

Tabla 2-2: Matriz D-H de un robot antropomórfico (información adaptada de [14]).

Link	$\boldsymbol{\theta_i}$	$\boldsymbol{d_i}$	$\boldsymbol{a_i}$	$\boldsymbol{\alpha_i}$
1	q_1	0	0	$\pi/2$
2	q_2	0	a_2	0
3	q_3	0	a_3	0

2.2.3Cinemática directa

La cinemática aborda la descripción geométrica del movimiento de sistemas mecánicos sin tomar en cuenta las fuerzas que lo producen. Por lo que la cinemática directa se refiere al estudio analítico del movimiento del robot para hallar la posición y orientación del efector

final. Para esto se reemplazan los parámetros D-H en la matriz de transformación homogénea (7) u (8) según sea el caso, generando de esta forma una por cada articulación, que al multiplicarlas todas se obtiene una transformación homogénea total (9).

$$T_n^0 = A_1^0 A_2^1 \dots A_n^{n-1} \quad (9)$$

Siguiendo el ejemplo del brazo antropomórfico, se procede a hallar su matriz homogénea total, donde *q* representa el ángulo en grados que se desea desplazar:

$$A_1^0(q_1) = \begin{bmatrix} c_1 & 0 & s_1 & 0 \\ s_1 & 0 & -c_1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A_2^1(q_2) = \begin{bmatrix} c_2 & -s_2 & 0 & a_2c_2 \\ s_2 & c_2 & 0 & a_2s_2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A_3^2(q_3) = \begin{bmatrix} c_3 & -s_3 & 0 & a_3c_3 \\ s_3 & c_3 & 0 & a_3s_3 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_3^0(q) = A_1^0 A_2^1 A_3^2 = \begin{bmatrix} c_1c_2c_3 & -c_1s_2s_3 & s_1 & c_1(a_2c_2+a_3c_2c_3) \\ s_1c_2c_3 & -s_1s_2s_3 & -c_1 & s_1(a_2c_2+a_3c_2c_3) \\ s_2s_3 & c_2c_3 & 0 & a_2s_2 + a_3s_2s_3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.2.4Cinemática inversa

La cinemática inversa determina las coordenadas de las articulaciones en función de la posición y orientación del efector final. Para esto es usada la matriz jacobiana, la cual está formada por las derivadas parciales de primer orden de una función, por lo que tiene la posibilidad de aproximar linealmente a la función en un punto. Para esto se parte de:

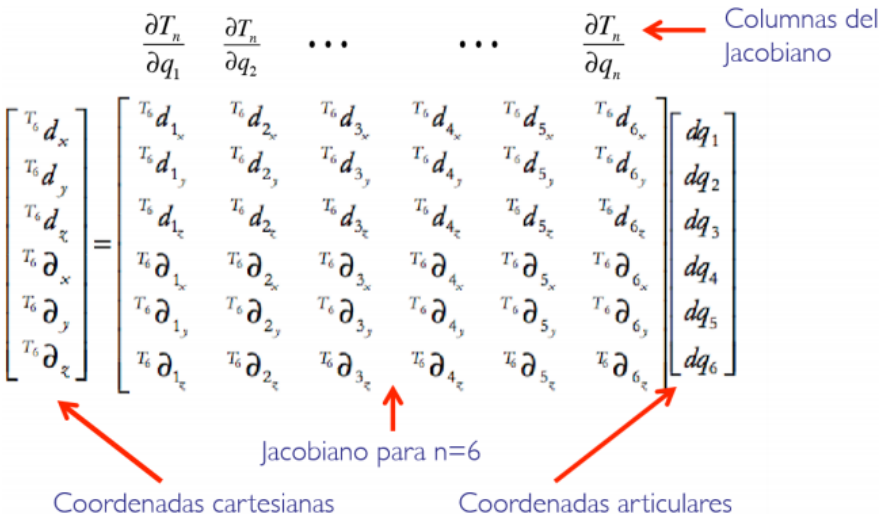
$$T + dT = \Delta T = \begin{bmatrix} 1 & -\partial_z & \partial_y & d_x \\ \partial_z & 1 & -\partial_x & d_y \\ -\partial_y & \partial_x & 1 & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} T$$

Donde $\partial = \left(\partial_x, \partial_y, \partial_z\right)^T$: rotación diferencial alrededor de los ejes principales del sistema de la base y $d = \left(d_x, d_y, d_z\right)^T$: rotación diferencial a lo largo de los ejes principales del sistema k. De esta forma se puede obtener (10) para representar un cambio diferencial dT_6 en términos de un cambio diferencial en coordenadas articulares.

$$^{m}\Delta = T^{-1}dT = T^{-1}\Delta T \quad (10)$$

Finalmente, el Jacobiano resulta en una matriz 6xn, donde n es el número de grados de libertad del robot, como se observa en la Figura 2-7, la cual se construye en base a la información de la Figura 2-8, utilizando las matrices de transformación homogénea encontradas.

Figura 2-7: Matriz jacobiana de un manipulador de 6 grados de libertad.



Nombre de la fuente: Información adaptada de [14].

Figura 2-8: Columna del Jacobiano.

$$T_n^0 = \begin{bmatrix} n_x & o_x & a_x & p_x \\ n_y & o_y & a_y & p_y \\ n_z & o_z & a_z & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Fila	Ecuación
1	$^{T_6}d_{i_x} = -n_x P_y + n_y P_x$
2	$^{T_6}d_{i_y} = -o_x P_y + o_y P_x$
3	$^{T_6}d_{i_z} = -a_x P_y + a_y P_x$
4	$^{T_6}\partial_{i_x} = n_z$
5	$^{T_6}\partial_{i_y} = o_z$
6	$^{T_6}\partial_{i_z} = a_z$

Articulación prismática
$^{T_6}d_i = n_z i + o_z j + a_z k$
$^{T_6}d_i = 0i + 0j + 0k$

Nombre de la fuente: Información adaptada de [14].

2.2.5Robotics Toolbox

Es una serie de herramientas y funciones útiles para el estudio y simulación de la robótica en Matlab. Para robots manipuladores ofrece algoritmos para comprobación de colisiones, generación de trayectorias, cinemática directa o inversa y la dinámica [19].

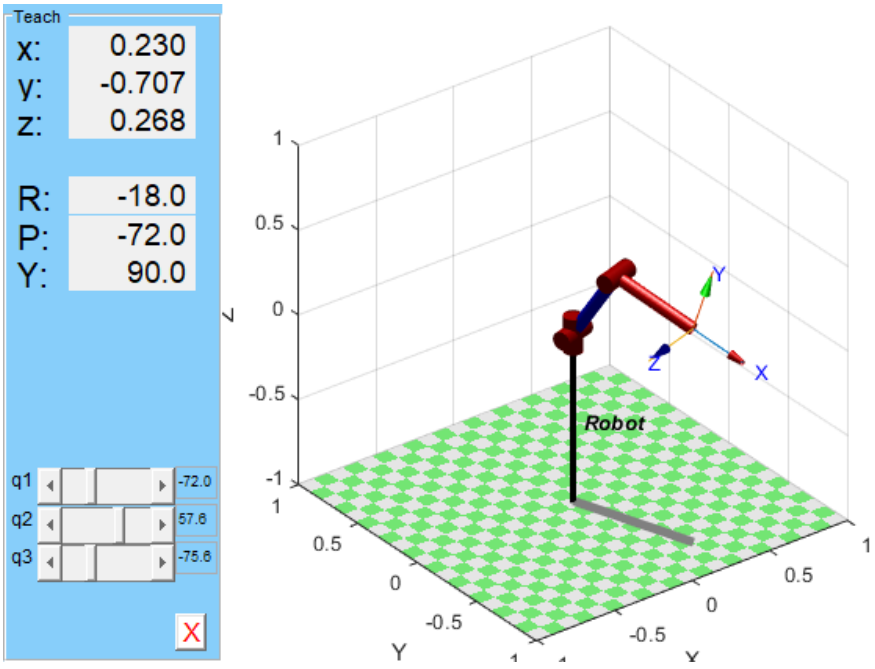
Para finalizar con el ejemplo del brazo antropomórfico, se realiza la simulación de la matriz D-H de la Tabla 2-3, a partir del siguiente algoritmo, reemplazando $q = 0$ y $a = 0.5$, obteniendo la Figura 2-9.

```
%Matriz DH [theta(rad) d(m) a(m) alpha(rad)]
L1=Link([0 0 0 pi/2]);
L2=Link([0 0 0.5 0]);
L3=Link([0 0 0.5 0]);

%Ensamble del robot
A=SerialLink([L1 L2 L3]);
A.name='Robot';

%Visualización del robot
q0=([0 0 0]);
A.plot(q0)
A.teach;
```

Figura 2-9: Simulación de robot antropomórfico.



Nombre de la fuente: Elaborada por el autor.

Luego, con el comando A.fkine(q) se puede generar la matriz de transformación homogénea total, donde *q* representa los ángulos en los que se moverá cada articulación, dando como resultado una matriz 4x4, la cual en la última columna se puede observar la posición del efector final en él espacio.

A.fkine([10 20 30])

$$T_3^0([10\ 20\ 30]^T) = \begin{bmatrix} -0.8097 & -0.2202 & -0.5440 & -0.576 \\ -0.5250 & -0.1427 & 0.8391 & -0.3735 \\ -0.2624 & 0.9650 & 0 & 0.3253 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

2.3 Ubuntu

Ubuntu es un sistema operativo de código abierto para computadoras, dispositivos inteligentes y tarjetas embebidas, que hace parte de las distribuciones de Linux. Actualmente se encuentra disponible la versión 20.04 LTS, que fue lanzada el 23 abril de 2020 [20].

A continuación, se presenta una serie de comandos básicos necesarios de la terminal de Ubuntu en la Tabla 2-3.

Tabla 2-3: Comandos básicos de Ubuntu.

Comando	Complemento	Función
cd	[/subdir]	Cambiar de carpeta.
cd ..		Volver a la carpeta anterior.
mkdir	<dir>	Crear nueva carpeta.
nano	<file-name>	Editor de texto.

2.4 ROS

Tomado y traducido de la página web principal [1], “El Sistema Operativo de Robot (ROS) es un framework flexible para escribir software de robot. Es una colección de herramientas, bibliotecas y convenciones que tienen como objetivo simplificar la tarea de crear un comportamiento robótico complejo y robusto en una amplia variedad de plataformas robóticas”.

El entorno de ROS corre principalmente en Ubuntu y experimentalmente en IOS y Windows. Puede diseñar, programar en lenguajes de Python y C++, e integrar sistemas, sensores o robots de distintos fabricantes.

2.4.1 Estructura

El software está organizado en paquetes. Estos son la unidad más pequeña que puede construirse en ROS, por lo que están en la capacidad de contener nodos, librerías, conjuntos de datos, archivos de configuración o cualquier otra cosa que constituya un módulo útil, con el objetivo de proporcionar esta funcionalidad de manera fácil para el consumo con el fin de poder reutilizar el software fácilmente.

La estructura que normalmente tiene un paquete se observa en la Tabla 2-4, y estos se administran con los comandos de la Tabla 2-5:

Tabla 2-4: Estructura de paquetes (información adaptada de [21]).

Elemento	Descripción
include/package_name/	Carpeta contenedora de las cabeceras de C++.
msg/	Carpeta contenedora de los mensajes estándar y los creados.
src/package_name/	Carpeta contenedora del código fuente del programa.
srv/	Carpeta contenedora de los tipos de servicios.
scripts/	Carpeta contenedora de scripts.
CMakeLists.txt	Archivo para compilar con CMake.
package.xml	Archivo que define propiedades sobre el paquete.

Tabla 2-5: Administración de paquetes (información adaptada de [21]).

Comando	Complemento	Función
rospack	<pkg-name>	Buscar y recuperar información sobre paquetes.
catkin_create_pkg	<pkg-name> <depends>	Crear nuevo paquete.
catkin_make	<pkg-name>	Compilar un paquete.
rosdep	install <pkg-name>	Instalar las dependencias del sistema de un paquete.
rqt		Mostrar las dependencias del paquete como un gráfico.

Igualmente, proporciona algunos adicionales para ayudar en la navegación de los paquetes, indicados en la Tabla 2-6:

Tabla 2-6: Navegación de paquetes (información adaptada de [21]).

Comando	Complemento	Función
roscd	<pkg-name>[/subdir]	Cambiar de carpeta.
rosd		Listar las carpetas.
rosls	<pkg-name>[/subdir]	Listar los archivos.
rosed	<pkg-name> <file-name>	Editar archivos.
roscp	<pkg-name> <file-name>	Copiar archivos.
rosrun	<pkg-name> <exe>	Correr ejecutables.

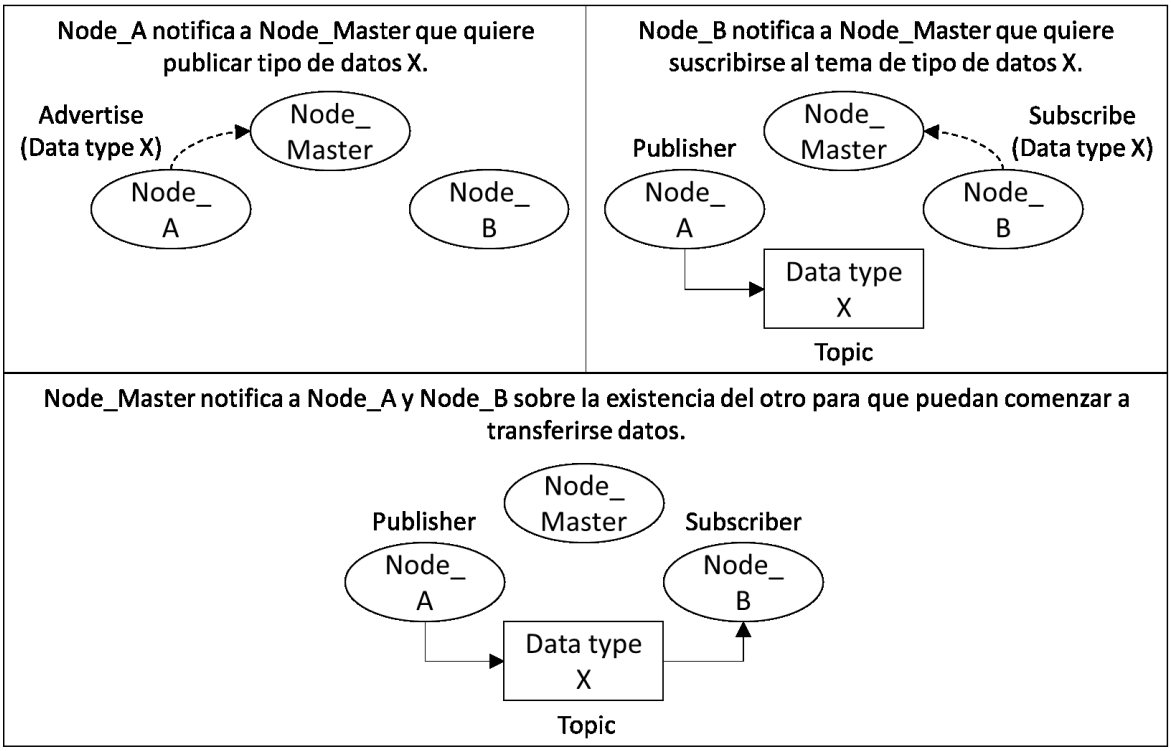
2.4.2 Arquitectura

ROS está basado en una arquitectura de grafos, donde los nodos son procesos que realizan cálculos y pueden comunicarse entre sí, publicando mensajes (estructuras de datos simples y matrices) sobre temas. Por lo que, esta arquitectura proporciona beneficios al sistema tales como: tolerancia a fallas ya que estas se aíslan en nodos individuales y reducción de la complejidad del código.

Los temas representan canales con un nombre único, que trabajan de modo unidireccional y en tiempo real, sobre los cuales los nodos intercambian mensajes mediante publicación/suscripción anónima (los nodos no saben con quien se comunican, pero si están interesados en datos se suscriben al tema relevante). Por otra parte, los nodos que necesitan comunicarse con alguno en específico (recibir respuesta a solicitud), deben usar servicios en su lugar, o el servidor de parámetros para mantener pequeñas cantidades de estado.

Finalmente, el nodo maestro tiene la función de ayudar a los demás nodos a encontrarse entre sí, realizando un seguimiento a los editores y suscriptores como se ilustra en la Figura 2-10. Igualmente, proporciona el servidor de parámetros.

Figura 2-10: Comunicación entre nodos (mensajes).



Nombre de la fuente: Tomado y adaptado de [21].

Para controlar cada elemento de la arquitectura, ROS ofrece las siguientes opciones, indicadas en las Tablas 2-7,8,9,10:

Tabla 2-7:: Administración de nodes (información adaptada de [21]).

Comando	Complemento	Función
rosnode info	<node-name>	Mostrar información acerca del nodo.
rosnode kill	<node-name>	Terminar la ejecución del nodo.
rosnode list		Listar los nodos en ejecución.
rosnode machine	<machine-name>	Listar los nodos en ejecución de la máquina.
rosnode ping	<node-name>	Probar la conectividad del nodo.
rosnode cleanup		Eliminar la información de registro de nodos inalcanzables.

Tabla 2-8: Administración de topics (información adaptada de [21]).

Comando	Complemento	Función
rostopic bw	<topic-name>	Mostrar ancho de banda usado por el tema.
rostopic delay	<topic-name>	Mostrar el retardo del tema.
rostopic echo	<topic-name>	Mostrar mensaje del tema.
rostopic find	<msg-type>	Busca tema por tipo.
rostopic hz	<topic-name>	Mostrar frecuencia de publicación del tema.
rostopic info	<topic-name>	Mostrar información acerca del tema.
rostopic list		Mostrar información de los temas activos.
rostopic pub	<topic-name> <topic-type> [data]	Publicar datos en el tema.
rostopic type	<topic-name>	Mostrar el tipo del tema.

Tabla 2-9: Administración de messages (información adaptada de [21]).

Comando	Complemento	Función
rosmsg show	<msg-type>	Mostrar información del mensaje.
rosmsg list		Listar los mensajes.
rosmsg package	<pkg-name>	Listar los mensajes del paquete.
rosmsg packages		Mostrar los paquetes que contienen mensajes.
rosmsg users	<msg-type>	Mostrar archivos que utilicen el tipo de mensaje.
Rosmsg md5	<msg-type>	Mostrar el md5sum del mensaje.

Tabla 2-10: Administración de services (información adaptada de [21]).

Comando	Complemento	Función
rossrv show	<srv-type>	Mostrar información del servicio.
rossrv list		Listar los servicios.
Rossrv md5	<srv-type>	Mostrar el md5sum del servicio.
rossrv package	<pkg-name>	Listar los servicios del paquete.
rossrv packages		Mostrar los paquetes que contienen servicios.
rosservice call	<srv-name> [srv-arg]	Llamar el servicio con los argumentos proporcionados.
rosservice find	<srv-type>	Mostrar los servicios del tipo.

Tabla 2-10: (Continuación)

Comando	Complemento	Función
rosservice list		Listar los servicios activos.
rosservice info	<srv-name>	Mostrar información acerca del servicio.
rosservice node	<srv-name>	Mostrar el nombre del nodo que proporciona el servicio.
rosservice type	<srv-name>	Mostrar el tipo del servicio.
rosservice uri	<srv-name>	Mostrar el URI del servicio.

2.4.3 Distribuciones

Una distribución ROS es un conjunto versionado de paquetes. El propósito es permitir que los desarrolladores trabajen en base de código relativamente estable gracias a las mejoras y correcciones de la última versión disponible al momento de los softwares necesarios para funcionar. La primera distribución fue lanzada en el año 2010, contando hasta la fecha con 13 diferentes, de las cuales únicamente 3 siguen recibiendo soporte como se muestra en la Tabla 2-11, las cuales fueron dirigidas a las últimas versiones LTS de Ubuntu.

Tabla 2-11: Distribuciones de ROS (información adaptada de [21]).

Distro	Release date	Poster	*Tuturtle*
ROS Noetic Ninjemys	May 23rd, 2020		
ROS Melodic Morenia	May 23rd, 2018		
ROS Kinetic Kame	May 23rd, 2016		

2.5 Gazebo

Es una herramienta de simulación de código abierto, con interfaz programática y gráfica, que permite probar algoritmos, diseñar robots, realizar pruebas de regresión y entrenar sistemas de inteligencia artificial utilizando escenarios realistas. Permite emular tanto entornos interiores como exteriores, además de poseer un motor de física robusto, que permite detección de colisiones y manipulación de objetos, además de la simulación de sensores [22].

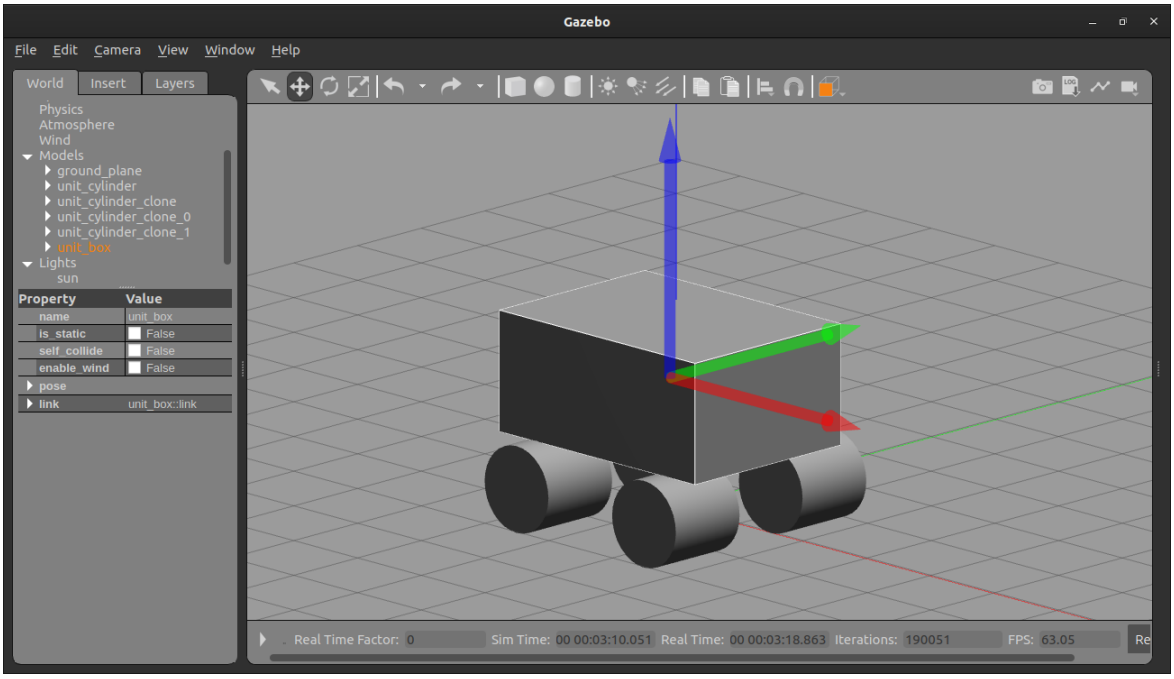
2.5.1 Formato

Gazebo maneja UDF, el cual es un archivo en lenguaje XML, donde describe el modelo del robot, contando con dos partes, una visual y una física.

- La parte visual describe los elementos que deben ser mostrados en su posición, para lo cual se cuenta con dos opciones:
  - Utilizar las figuras dispuestas (cilindro, cubo y esfera), como se observa en la Figura 2-11.
  - Establecer un mesh (descripción en CAD del elemento), para lo cual se debe exportar el plano 3D en extensión DAE o STL.
- La parte física describe dimensiones, masa, inercia, fricción, etc., que permiten establecer el comportamiento del objeto frente al ambiente.

Finalmente, la OSRF ofrece una web [23], donde se encuentran disponibles todos los parámetros posibles para cada elemento con su descripción, junto a ejemplos de estructuras básicas para no iniciar desde cero las simulaciones.

Figura 2-11: Interfaz de Gazebo.



Nombre de la fuente: Elaborada por el autor.

2.6 Qt

Qt es un framework multiplataforma de código abierto orientado a objetos, utiliza el lenguaje de C++ de forma nativa, pero pueden ser utilizados otros. Este software es utilizado para desarrollar programas que utilicen interfaz gráfica y también herramientas para introducir líneas de comandos [24].

A continuación, se presenta una serie de comandos básicos necesarios de Qt en la Tabla 2-12.

Tabla 2-12: Comandos básicos de Qt.

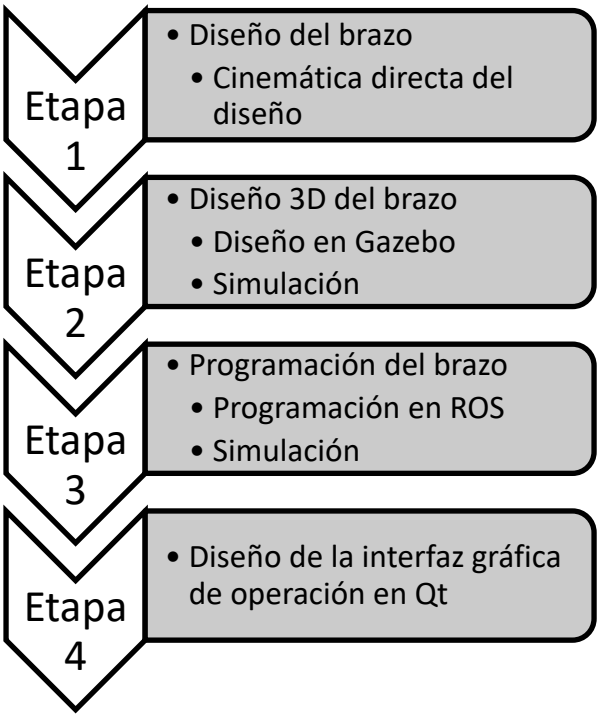
Comando	Función
make	Compilar un paquete.
./<pkg-name>	Ejecutar una interfaz gráfica especifica.

3. Metodología

Para cumplir con el desarrollo propuesto, se planeó que la estructura fuera antropomórfica, contara con 3 grados de libertad, y tuviera eslabones de alrededor 0.5 m de largo, esto con el fin de que el modelo pueda ser de utilidad a la hora de realizar algún tipo de tarea en él simulador y una posible implementación física en el futuro.

Por otra parte, la distribución del framework que se eligió fue ROS Noetic Ninjemys, por lo que las versiones de los softwares necesarios para su funcionamiento (incluyendo Ubuntu y Gazebo) son las más recientes a la fecha de su lanzamiento, debido a que sus requerimientos lo exigen, garantizando que el rendimiento y compatibilidad sean elevados.

A continuación, se describe la metodología seguida a lo largo de cuatro etapas:

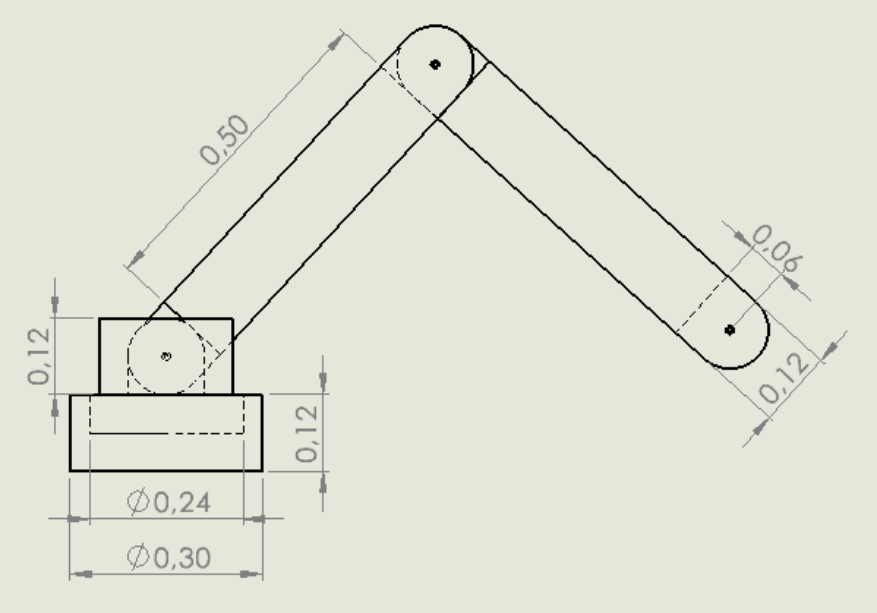


3.1 Diseño del brazo

En esta primera etapa de la metodología se realizó un plano sencillo del robot como guía, en el cual se basó el desarrollo del proyecto. Para esto se tomaron los parámetros propuestos al inicio del capítulo, por lo que no cuenta con un efector final y al ser antropomórfico tiene únicamente articulaciones rotacionales.

Finalmente, el diseño obtenido junto a sus respectivas medidas, las cuales son necesarias más adelante en esta fase y en la posterior, se presenta en la Figura 3-1.

Figura 3-1: Diseño del brazo robot.



Nombre de la fuente: Elaborada por el autor.

3.1.1 Cinemática directa del diseño

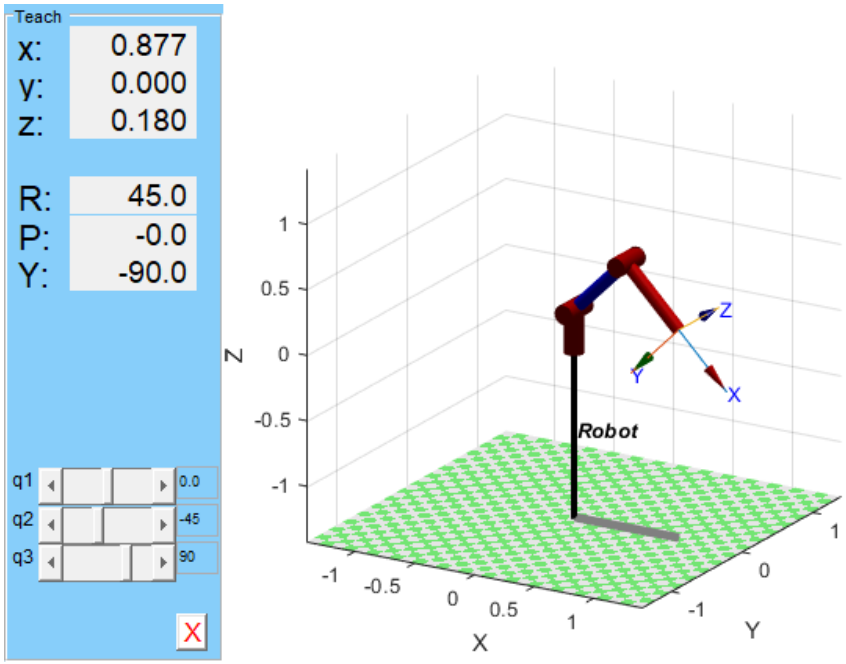
Teniendo como base los parámetros D-H en la Tabla 2-2 del ejemplo presentado en el capítulo anterior de un robot antropomórfico, se pudo deducir la matriz D-H del diseño creado. Para esto se reemplazaron las variables $a_{2,3} = 0.62$ ya que era la distancia entre los dos sistemas de coordenadas y $d_1 = 0.18$ ya que ahora el robot poseía una base, por lo que el sistema de coordenadas se ubicaba a esta altura, obteniendo de esta manera la Tabla 3-1.

Tabla 3-1: Matriz D-H (elaborada por el autor).

Link	$\boldsymbol{\theta_i}$	$\boldsymbol{d_i}$	$\boldsymbol{a_i}$	$\boldsymbol{\alpha_i}$
1	q_1	0.18	0	$\pi/2$
2	q_2	0	0.62	0
3	q_3	0	0.62	0

Habiendo obtenido los parámetros D-H del diseño, se procedió a simular y calcular la matriz de transformación homogénea total con ayuda de Robotics Toolbox, obteniendo la Figura 3-2 y con el comando A.fkine(q), la matriz (11).

Figura 3-2: Simulación del diseño del brazo robot.



Nombre de la fuente: Elaborada por el autor.

$$T_3^0([0\ -45\ 90]^T) = \begin{bmatrix} 0.5253 & -0.8509 & 0 & 0.6514 \\ 0 & 0 & 1 & 0 \\ -0.8509 & -0.5253 & 0 & 0.18 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (11)$$

3.2 Diseño 3D del brazo

Primero que todo se preparó el entorno de trabajo en el cual se desarrolló el proyecto. Para esto, por comodidad se decidió realizar todo el proceso mediante la terminal de Ubuntu siguiendo la siguiente lista de pasos:

- Crear la carpeta para el proyecto y una dentro para código.

  ```
  >> mkdir proyecto
  >> cd proyecto
  >> mkdir src
  ```

- Inicializar el espacio de trabajo catkin y compilarlo.

  ```
  >> cd src
  >> catkin_init_workspace
  >> cd ..
  >> catkin_make
  ```

- Definir y crear el paquete de desarrollo.

  ```
  >> cd src
  >> catkin_create_pkg brazo roscpp gazebo_ros std_msgs
  ```

- Crear la carpeta para la simulación del mundo y crear uno en blanco a partir del código ejemplo de la web del SDF [23].

  ```
  >> cd ..
  >> cd src/brazo
  >> mkdir worlds
  >> cd worlds
  >> nano brazo.world
  ```

- Crear la carpeta para el lanzador y crearlo a partir de la web de Gazebo [22], Anexo A3.

  >> mkdir launcher
  >> cd launcher
  >> nano brazo.launcher

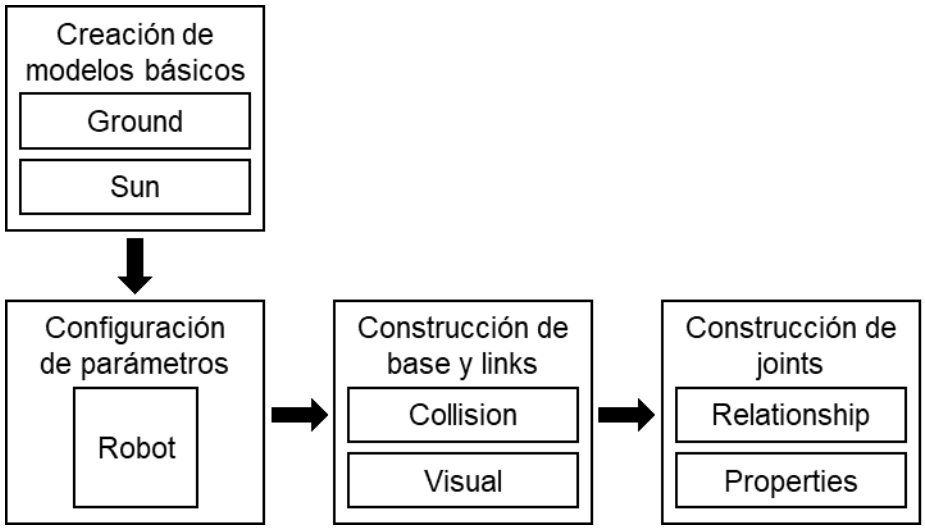
- Agregar el proyecto al bashrc [25] para que ROS reconozca el entorno en todas las terminales.

  >> cd ~/proyecto/devel
  >> echo "source ~/proyecto/devel/setup.bash" >> ~/.bashrc

3.2.1 Diseño en Gazebo

Para crear el mundo que se simuló en Gazebo se siguió el diagrama de la Figura 3-3.

Figura 3-3: Procedimiento de la creación del mundo.



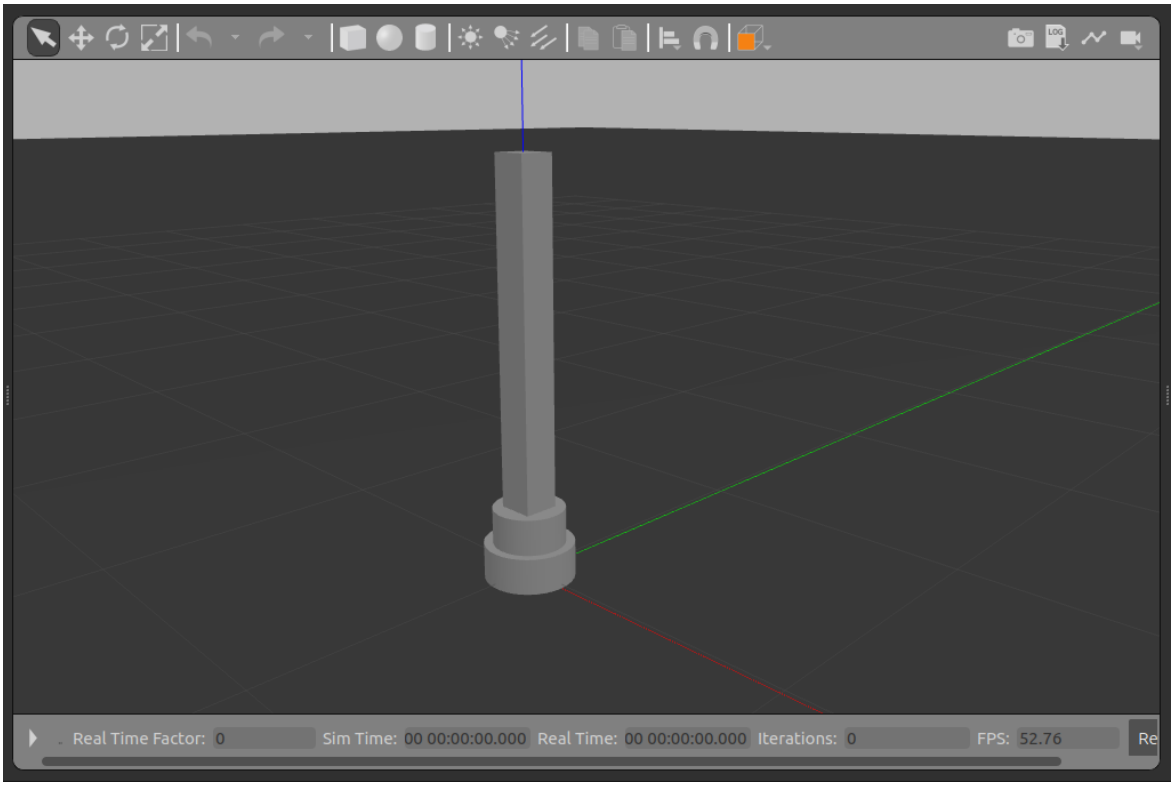
Nombre de la fuente: Elaborada por el autor.

- Creación de modelos básicos: Se agregó al mundo un modelo del plano de la tierra para evitar que los objetos cayeran al vacío y un modelo del sol para proporcionar la iluminación.

- Configuración de parámetros: Se estableció la posición inicial del robot y se habilitó la posibilidad de colisión entre las piezas.
- Construcción de base y links: Se ubicaron las piezas geométricas en el espacio, y se describieron:
  - Collision: Se establecieron las propiedades geométricas que serán afectadas por las propiedades físicas indicadas.
  - Visual: Se establecieron las propiedades geométricas que se visualizarán.
- Construcción de joints: Se estableció la ubicación de las articulaciones en los eslabones, así como la relación entre estos. También se determinaron las propiedades físicas y los límites de movimiento.

El resultado obtenido se presenta en la Figura 3-4, donde se observa que el diseño presenta problemas de obstrucción en el movimiento debido a la superposición de piezas.

Figura 3-4: Simulación 1 del mundo.



Nombre de la fuente: Elaborada por el autor.

Para solucionar esto, como se indicó en el capítulo anterior, es posible importar piezas en formato DAE o STL diseñadas en algún software CAD. Para esto, se recomienda que antes de realizar la exportación, se tenga en cuenta que las dimensiones deben estar en metros, y se debe crear un sistema de coordenadas individual como el de gazebo (Z=↑), el cual se ubicará en el centro del objeto, esto con el fin de que al colocar la pieza en el simulador se encuentre orientada como se desea.

Después de diseñar el brazo como se deseaba y tener las piezas en el formato deseado se procedió a seguir los siguientes pasos:

- Crear las carpetas y archivos necesarios, Anexos B1.1-2.

```
>> mkdir -p .gazebo/models/arm_base/meshes
>> nano .gazebo/models/arm_base/model.config
>> nano .gazebo/models/arm_base/model.sdf
```

- Mover las piezas a su nueva ubicación.

```
>> mv base.STL ../.gazebo/models/arm_base/meshes
```

- Reemplazar en el mundo el contenido de la geometría de los modelos.

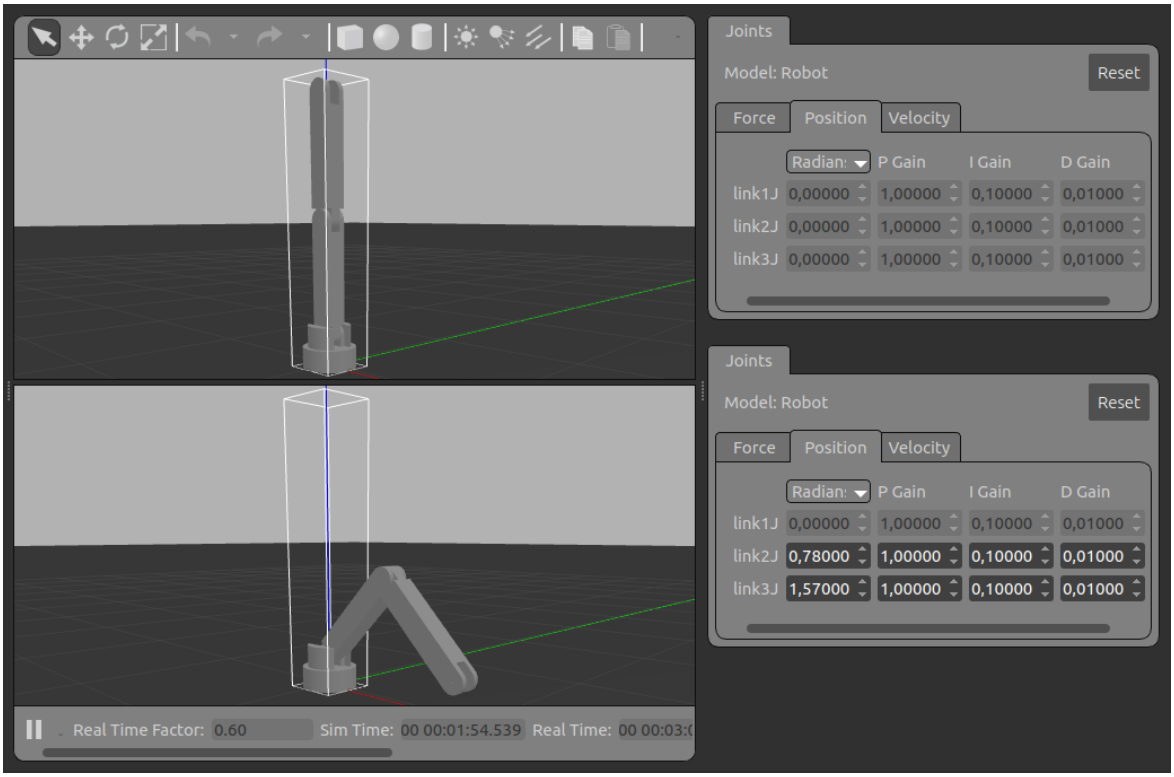
```
<geometry>
    <mesh>
        <uri>model://arm_base/meshes/base.STL</uri>
    </mesh>
</geometry>
```

Finalmente, después de haber terminado con el diseño 3D del brazo, el algoritmo resultante se encuentra en el Anexo B2.

3.2.2 Simulación

Para comprobar que el mundo creado quedó correcto, se lanzó y se hicieron pruebas de movimiento a las articulaciones desde Gazebo cómo se observa en la Figura 3-5.

Figura 3-5: Simulación 2 del mundo.



Nombre de la fuente: Elaborada por el autor.

3.3 Programación del brazo

Al igual que en la etapa anterior, se realizaron algunos preparativos antes de empezar a programar:

- En el archivo package.xml se añadió la ruta de compilado de las librerías de dependencia mencionadas al crear el paquete de desarrollo, quedando como el Anexo A1.

```
<gazebo_ros plugin_path="${prefix}/lib" gazebo_media_path="${prefix}"/>
```

- Se configuró el archivo de compilado CMakeLists.txt, añadiendo los directorios y enlaces de las dependencias utilizados tanto de ROS como de Gazebo. También se añadió un conversor de clases y objetos, junto al compilador que se utilizó, quedando como el Anexo A2.

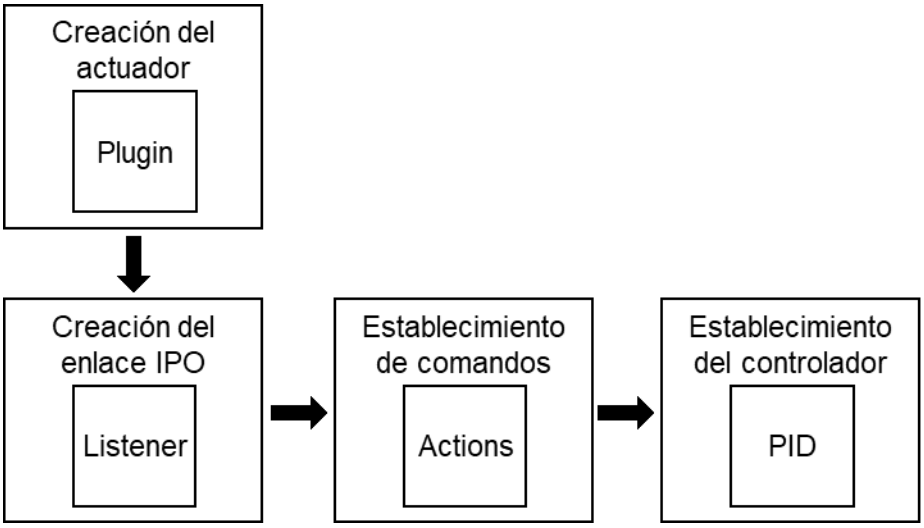
```
${Boost_INCLUDE_DIR}
add_definitions(-std=c++17)
```

Hay que aclarar que en este archivo igualmente se debieron mencionar los scripts que se utilizaron.

3.3.1 Programación en ROS

Para crear los scripts de ROS para el funcionamiento del robot se siguió el diagrama de la Figura 3-6.

Figura 3-6: Procedimiento de creación de los scripts del funcionamiento del brazo.



Nombre de la fuente: Elaborada por el autor.

- Creación del actuador: Se creó un plugin con la función de ser el actuador entre ROS y el mundo simulado, en el cual se añadió la librería resultante.

```
<plugin name="actuador" filename="libbrazo.so">  </plugin>
```

- Creación del enlace IPO: Se creó un subscriptor/publicador con la función de ser el enlace de comunicación con el usuario, el cual al recibir una cadena la publicará para su posterior uso, Anexos C2.1-2.
- Establecimiento de comandos: Se establecieron las acciones que realizará ROS al procesar las cadenas de comandos recibidas, Anexos C3.1-2:
  - d: Detectar los elementos del robot.
  - p: Aplicar fuerza de movimiento a una articulación especifica.
  - m: Mover una articulación especifica en grados.
  - e: Ejecutar archivo con secuencia de comandos.
- Establecimiento del controlador: Se estableció un bloque PID en cada articulación, para el control de fuerza aplicada al moverlas hasta el punto deseado, Anexos C1.1-2.

(p, i, d, imax, imin, cmdMax, cmdMin)

3.3.2 Simulación

Para comprobar el correcto funcionamiento de los scripts se decidió seguir una secuencia de movimientos. Para mantener el orden, se creó una carpeta contenedora de secuencias para ejecutar y se guardó una tarea. Finalmente se compiló el paquete y se lanzó el mundo.

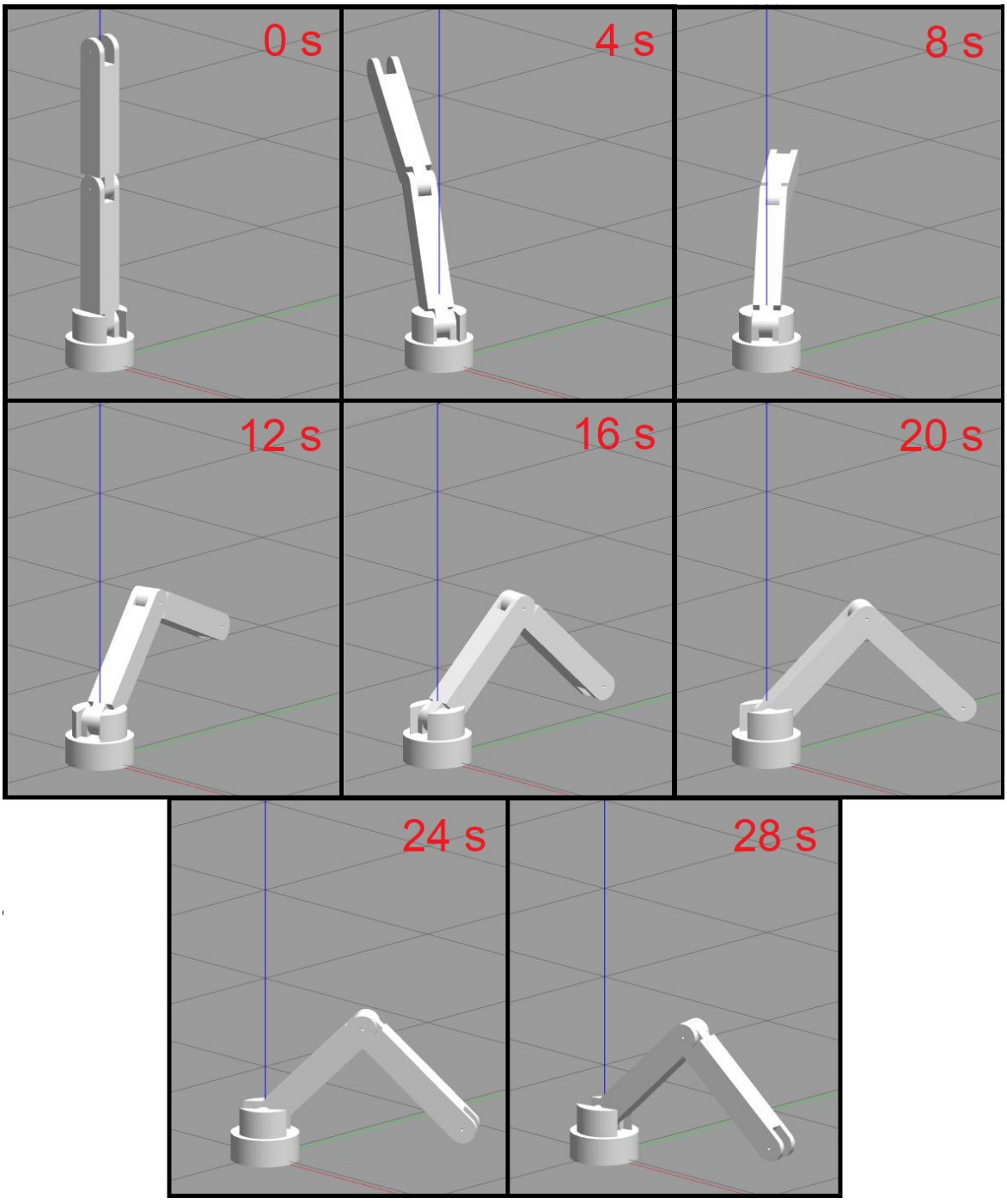
```
>> mkdir proyecto/src/brazo/programas
>> nano proyecto/src/brazo/programas/secuencia
```

```
m link1J 180
m link2J -45
m link3J -90
```

```
>> catkin_make --force-cmake
>> roslaunch brazo brazo.launcher
>> rostopic pub /brazo std_msgs/String “e ../proyecto/src/brazo/programas/tarea”
```

El resultado de la simulación de la secuencia de movimientos se observa en la Figura 3-7.

Figura 3-7: Simulación de secuencia de movimientos.



Nombre de la fuente: Elaborada por el autor.

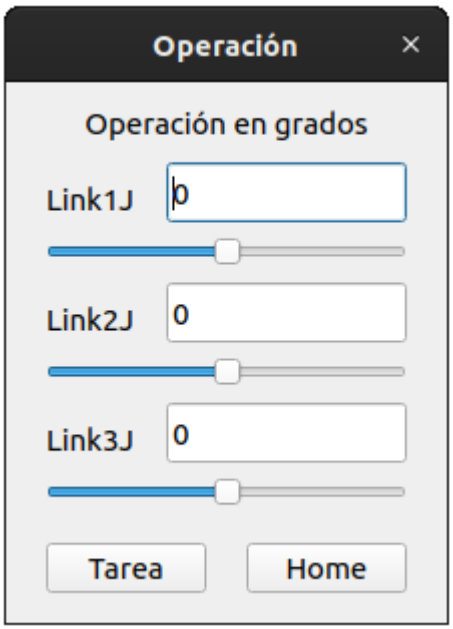
3.4 Diseño de la Interfaz gráfica de operación en Qt

El desarrollo de la interfaz gráfica se realizó utilizando el software Qt, donde al incluir las librerías de ROS se integra perfectamente con el proyecto realizado, logrando mover cada una de las articulaciones de la estructura, Anexo D.

El diseño de la interfaz gráfica cuenta con tres posibles formas de operación, las cuales son explicadas a continuación:

- LineEdit: Recibe un ángulo en grados y automáticamente envía el comando de movimiento al sistema.
- Slider: Adquiere el valor de la posición al desplazar la barra ya sea con el mouse o el teclado y envía automáticamente el comando al sistema.
- PushButton: Esta serie de botones al ser presionados envían una secuencia de movimientos guardada previamente al sistema.

Figura 3-8: Interfaz gráfica de operación.



Nombre de la fuente: Elaborada por el autor.

4. Resultados

Para desarrollar este proyecto, la base fue el diseño del brazo antropomórfico en el simulador de Gazebo, que aunque inicialmente se utilizaron las herramientas disponibles desde su interfaz gráfica, el resultado no fue muy bueno tanto visual como físicamente, por lo que mejor se optó por la posibilidad de reemplazar las piezas predeterminadas por unas diseñadas a partir de un software CAD, obteniendo exitosamente el modelo deseado a partir del plano creado inicialmente, sin causar ningún inconveniente en la simulación.

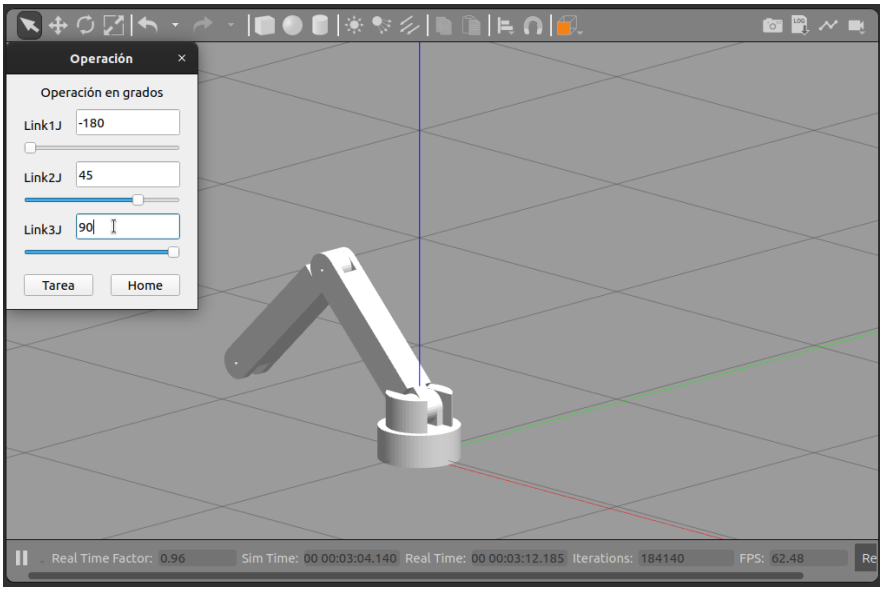
A partir del mundo diseñado se comenzaron a crear los scripts, que a pesar de que la meta inicial era crearlos en lenguaje de Python, esto no se pudo llevar a cabo debido a problemas de conversión, por lo que se terminó optando por utilizar el lenguaje de C++, ya que los softwares utilizados están desarrollados principalmente en base a este. Finalmente, se logró obtener una serie de algoritmos localizados en la sección de Anexos, capaces de manipular el comportamiento de la estructura en el simulador, desde la terminal de Ubuntu.

Teniendo en funcionamiento la totalidad del proyecto se inició el desarrollo de la interfaz gráfica en el software Qt, donde se encontraron problemas de reconocimiento entre las variables de los elementos y las de ROS, por lo que se tuvieron que realizar saltos extras en la conversión de los tipos de variables hasta lograr establecer exitosamente la comunicación en el sistema.

Como se fue indicando a través de esta sección, los resultados obtenidos en el transcurso de las diferentes etapas desarrolladas, fueron integrándose poco a poco entre sí, logrando crear un entorno de simulación completo a partir de la concepción de un robot, evidenciando mejoría en la operación entre cada una de estas etapas desarrolladas.

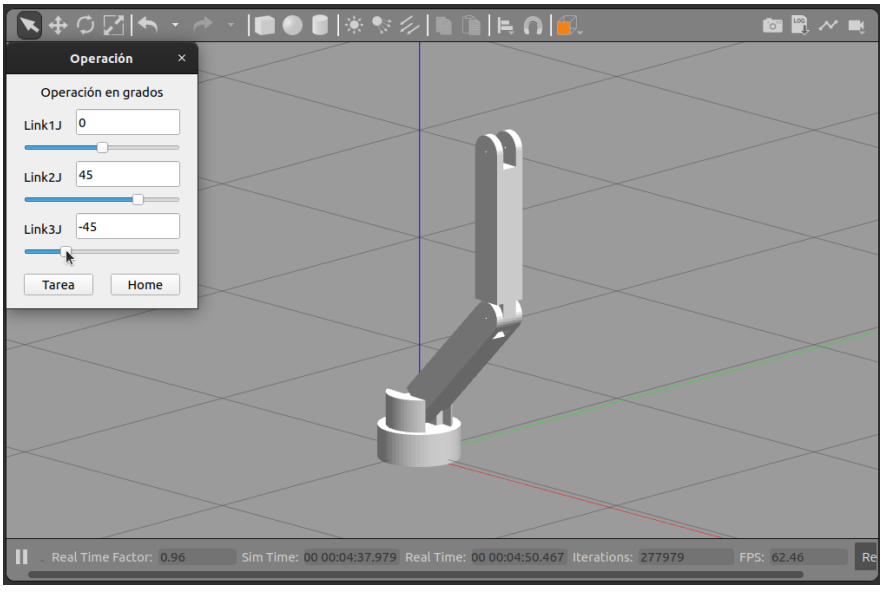
Finalmente, para comprobar la correcta integración y el funcionamiento del sistema se realizaron pruebas con los diferentes elementos de la interfaz gráfica, obteniendo los resultados de las Figuras 4-1,2 al manipular los LineEdit y Slider, que al estar interconectados los elementos se sincronizan.

Figura 4-1: Prueba del elemento LineEdit.



Nombre de la fuente: Elaborada por el autor.

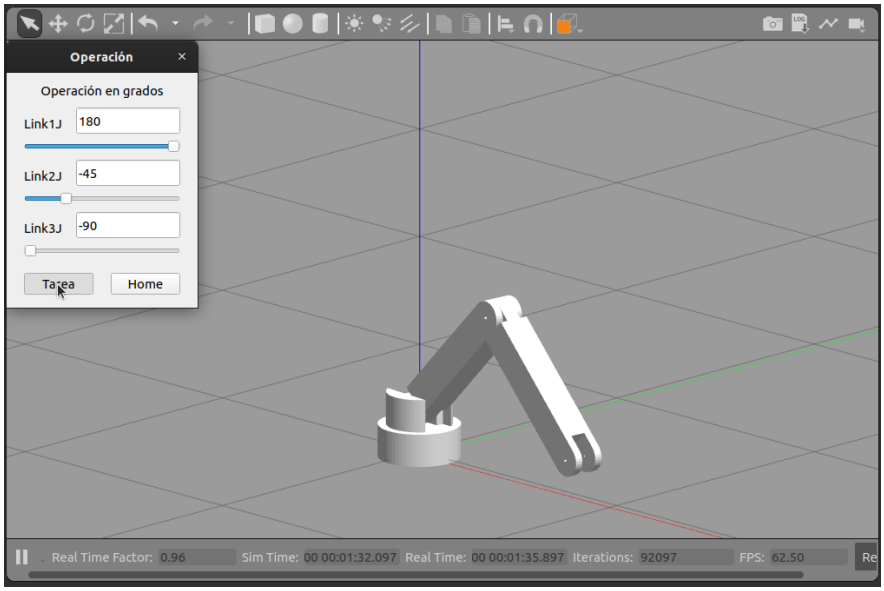
Figura 4-2: Prueba del elemento Slider.



Nombre de la fuente: Elaborada por el autor.

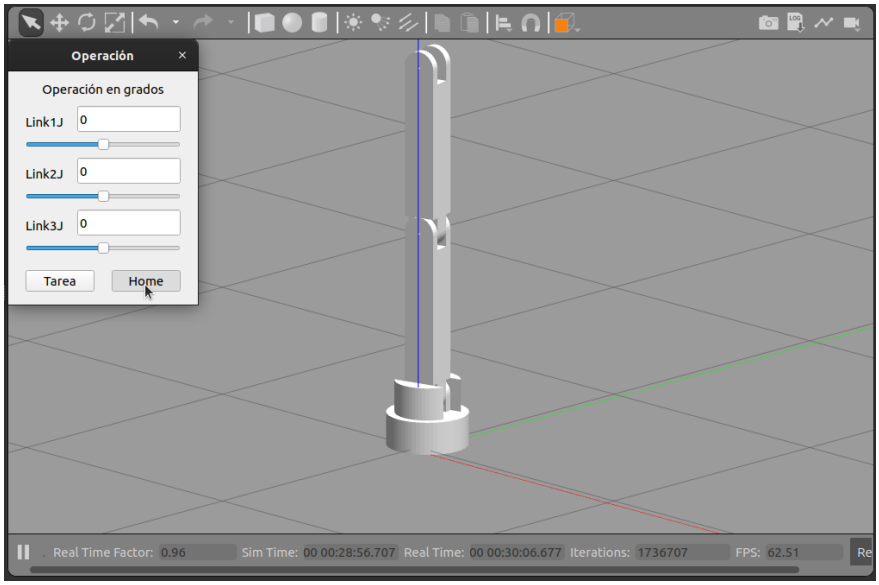
Por otra parte, los PushButton al tener la capacidad de enviar secuencias de movimientos específicas, se decidió utilizar la creada anteriormente, observando el mismo resultado obtenido en la Figura 4-3. Además, se creó una nueva secuencia donde se almacenó la posición inicial (Home), como se observa en la Figura 4-4.

Figura 4-3: Prueba del elemento PushButton de tareas.



Nombre de la fuente: Elaborada por el autor.

Figura 4-4: Prueba del elemento PushButton de home.



Nombre de la fuente: Elaborada por el autor.

5. Conclusiones y recomendaciones

5.1 Conclusiones

En el presente trabajo se abordó la concepción, el diseño, el control (básico), la operación y la implementación (virtual) de un robot manipulador antropomórfico, para proporcionar una base que permita el desarrollo de nuevas maneras de enseñanza, dirigido a la Universidad Antonio Nariño.

El simulador de Gazebo es una potente herramienta de simulación al ser capaz de aceptar diseños realizados en softwares CAD, sin tener la necesidad de modificar el código más allá que para introducir las piezas en el entorno, logrando de esta forma probar diseños propios de forma rápida con el fin de corregir problemas que puedan ocurrir.

A pesar de que ROS pueda programarse en lenguaje C++ y Python, el simulador Gazebo está basado en C++, lo que conlleva que la adaptación y conversión de código a enteramente Python sea un problema si no se poseen los suficientes conocimientos en programación, por lo que se recomienda empezar a utilizar lenguaje C++ desde el inicio de los proyectos desarrollados en ROS.

El usar una interfaz gráfica no es esencial para la operación de un robot como se evidenció en el transcurso del proyecto, pero si es de gran ayuda a la hora de facilitar su interacción, en especial para personas que no tengan mucho conocimiento en el tema, al no tener la necesidad de interactuar directamente con la consola de comandos.

Un robot de este tipo puede ser utilizado para crear un laboratorio virtual, ya que sin importar donde se encuentre el estudiante, si este tiene instalados los programas necesarios, puede interactuar con el robot con ayuda de la interfaz gráfica desde Gazebo, o si el robot es físico se podría llegar a visualizar con cámaras.

5.2 Recomendaciones

Las recomendaciones finales de este proyecto, basadas en los resultados son:

- Pasar el robot a Simscape de MATLAB, para generar las matrices ABCD y de esa manera hacer un cálculo más apropiado para un controlador especifico.
- Si se desea se puede implementar el código en un microcontrolador para luego controlar un robot real, y de esa forma hacer un aporte a los laboratorios a bajo costo, logrando a su vez incrementar el aprendizaje en el área de la robótica, desde la concepción, hasta la implementación.

A. Anexo: Algoritmos esenciales de ROS

A1. Propiedades del paquete (package.xml)

```
<?xml version="1.0"?>
## Propiedades del paquete.
<package>
    <name>brazo</name>
    <version>0.0.0</version>
    <description>The brazo package</description>
    <maintainer email="jpiratova227@uan.edu.co">Sebastian Piratova</maintainer>
    <license>TODO</license>
    ## Indicación de dependencias necesarias.
    <buildtool_depend>catkin</buildtool_depend>
    <build_depend>roscpp</build_depend>
    <build_depend>gazebo_ros</build_depend>
    <build_depend>std_msgs</build_depend>
    <run_depend>roscpp</run_depend>
    <run_depend>gazebo_ros</run_depend>
    <run_depend>std_msgs</run_depend>
    ## Indicación de la ruta de compilación de las librerías de dependencias
mencionadas.
    <export>
        <gazebo_ros plugin_path="${prefix}/lib" gazebo_media_path="${prefix}"/>
    </export>
</package>
```

A2. Compilador de CMake (CMakeLists.txt)

```
cmake_minimum_required(VERSION 3.0.2)
## Indicación de la ubicación de los algoritmos en el paquete a compilar.
project(brazo)
## Indicación de paquetes necesarios y sus dependencias.
find_package(catkin REQUIRED COMPONENTS
    roscpp
    gazebo_ros
    gazebo_plugins
    std_msgs
)
find_package(gazebo REQUIRED)
catkin_package(DEPENDS
    roscpp
    gazebo_ros
    gazebp_plugins
)
## Indicación de directorios donde se ubican los componentes necesarios.
link_directories(${GAZEBO_LIBRARY_DIRS})
include_directories(
    include
    ## Indicación de conversor de clases y objetos.
    ${Boost_INCLUDE_DIR}
    ${catkin_INCLUDE_DIRS}
    ${GAZEBO_INCLUDE_DIRS}
)
## Indicación de scripts utilizados en el proyecto.
add_library(${PROJECT_NAME}
    codigo/brazo.cpp
    codigo/listener.cpp
    codigo/comandos.cpp
)
```

```
target_link_libraries(${PROJECT_NAME}
    ${catkin_LIBRARIES}
)
## Compilador de C++ utilizado.
add_definitions(-std=c++17)
```

A3. Lanzador de ROS (brazo.launcher)

```
## Parámetros con los que iniciará Gazebo.
<launch>
    <arg name="paused" default="true"/>
    <arg name="use_sim_time" default="true"/>
    <arg name="gui" default="true"/>
    <arg name="headless" default="false"/>
    <arg name="debug" default="false"/>
    <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find brazo)/world/brazo.world"/>
        <arg name="verbose" value="true"/>
        <arg name="debug" value="$(arg debug)"/>
        <arg name="gui" value="$(arg gui)"/>
        <arg name="paused" value="$(arg paused)"/>
        <arg name="use_sim_time" value="$(arg_use_sim_time)"/>
        <arg name="headless" value="$(arg headless)"/>
    </include>
</launch>
```

B. Anexo: Algoritmos del mundo

B1.1. Configuración del modelo importado (model.config)

```
<?xml version="1.0"?>
## Propiedades del modelo importado.
<model>
    <name>Arm</name>
    <version>1.0</version>
    <sdf version="1.7">model.sdf</sdf>
    <author>
        <name>Optional: Sebastian Piratova</name>
        <email>Optional: jpiratova227@uan.edu.co</email>
    </author>
    <description>
        A model of a Arm.
    </description>
</model>
```

B1.2. Simulación del modelo importado (model.sdf)

```
<?xml version="1.0" ?>
## Descripción del modelo importado.
<sdf version="1.7">
    <model name="arm">
        <static>true</static>
```

```
        <link name="link">
            <collision name="collision">
                <geometry>
                    <mesh>
                        <uri>model://arm_base/meshes/base.STL</uri>
                    </mesh>
                </geometry>
            </collision>
            <visual name="visual">
                <geometry>
                    <mesh>
                        <uri>model://arm_base/meshes/base.STL</uri>
                    </mesh>
                </geometry>
            </visual>
        </link>
    </model>
</sdf>
```

B2. Simulación del mundo (brazo.world)

```
<sdf version='1.7'>
    <world name='default'>
    ## Determinación de propiedades físicas del entorno de simulación.
    <gravity>0 0 -9.8</gravity>
    <magnetic_field>6e-06 2.3e-05 -4.2e-05</magnetic_field>
    <atmosphere type='adiabatic'/>
    <physics type='ode'>
        <max_step_size>0.001</max_step_size>
        <real_time_factor>1</real_time_factor>
        <real_time_update_rate>1000</real_time_update_rate>
    </physics>
```

```
<scene>
    <ambient>0.4 0.4 0.4 1</ambient>
    <background>0.7 0.7 0.7 1</background>
    <shadows>1</shadows>
</scene>
<wind/>
<spherical_coordinates>
    <surface_model>EARTH_WGS84</surface_model>
    <latitude_deg>0</latitude_deg>
    <longitude_deg>0</longitude_deg>
    <elevation>0</elevation>
    <heading_deg>0</heading_deg>
</spherical_coordinates>
## Descripción del modelo del plano de la tierra.
<model name='ground_plane'>
    <static>1</static>
    <link name='ground_plane::link'>
        <collision name='collision'>
            <geometry>
                <plane>
                    <normal>0 0 1</normal>
                    <size>100 100</size>
                </plane>
            </geometry>
            <surface>
                <contact>
                    <collide_bitmask>65535</collide_bitmask>
                    <ode/>
                </contact>
                <friction>
                    <ode>
                        <mu>100</mu>
                        <mu2>50</mu2>
                    </ode>
```

```
                    <torsional>
                        <ode/>
                    </torsional>
                </friction>
                <bounce/>
            </surface>
            <max_contacts>10</max_contacts>
        </collision>
        <visual name='visual'>
            <cast_shadows>0</cast_shadows>
            <geometry>
                <plane>
                    <normal>0 0 1</normal>
                    <size>100 100</size>
                </plane>
            </geometry>
            <material>
                <script>
                    <uri>file://media/materials/scripts/gazebo.material</uri>
                    <name>Gazebo/Grey</name>
                </script>
            </material>
        </visual>
        <self_collide>0</self_collide>
        <enable_wind>1</enable_wind>
        <kinematic>1</kinematic>
    </link>
    <pose>0 0 0 0 -0 0</pose>
</model>
## Descripción del modelo del sol.
<light name='sun' type='directional'>
    <cast_shadows>1</cast_shadows>
    <pose>0.005975 0.006008 10 0 -0 0</pose>
```

```
        <diffuse>0.8 0.8 0.8 1</diffuse>
        <specular>0.2 0.2 0.2 1</specular>
        <attenuation>
            <range>1000</range>
            <constant>0.9</constant>
            <linear>0.01</linear>
            <quadratic>0.001</quadratic>
        </attenuation>
        <direction>-0.5 0.1 -0.9</direction>
        <spot>
            <inner_angle>0</inner_angle>
            <outer_angle>0</outer_angle>
            <falloff>0</falloff>
        </spot>
    </light>
    ##  Indicación de posición y desplazamiento de los modelos básicos.
    <state world_name='default'>
        <sim_time>0 0</sim_time>
        <real_time>0 0</real_time>
        <wall_time>1600905980 555686776</wall_time>
        <iterations>0</iterations>
        <model name='ground_plane'>
            <pose>0 0 0 0 -0 0</pose>
            <scale>1 1 1</scale>
            <link name='ground_plane::link'>
                <pose>0 0 0 0 -0 0</pose>
                <velocity>0 0 0 0 -0 0</velocity>
                <acceleration>0 0 0 0 -0 0</acceleration>
                <wrench>0 0 0 0 -0 0</wrench>
            </link>
        </model>
        <light name='sun'>
            <pose>0.005975 0.006008 10 0 -0 0</pose>
        </light>
```

```
</state>
## Indicación de la posición de la visualización inicial del simulador.
<gui fullscreen='0'>
    <camera name='user_camera'>
        <pose>5.33901 -4.3366 2.81098 0 0.275643 2.35619</pose>
        <view_controller>orbit</view_controller>
        <projection_type>perspective</projection_type>
    </camera>
</gui>
## Indicación de parámetros iniciales del robot diseñado.
<model name="brazo">
    <pose frame="">0 0 0 0 0 0</pose>
    <self_collide>1</self_collide>
    ## Descripción de la base del robot diseñado.
    <link name="base">
        <pose>0 0 0 0 0 0</pose>
        ## Descripción física de la base.
        <collision name="baseC">
            <pose>0 0 0 0 0 0</pose>
            ## Indicación de la pieza importada utilizada.
            <geometry>
                <mesh>
                    <uri>model://arm_base/meshes/base.STL</uri>
                </mesh>
            </geometry>
            ## Indicación de las propiedades físicas de la pieza.
            <surface>
                <contact>
                    <ode>
                        <soft_cfm>0</soft_cfm>
                        <soft_erp>0.2</soft_erp>
                        <kp>1e+13</kp>
                        <kd>1e+11</kd>
```

```
                        <max_vel>0.125</max_vel>
                        <min_depth>0</min_depth>
                    </ode>
                </contact>
                <friction>
                    <torsional>
                        <ode/>
                    </torsional>
                    <ode/>
                </friction>
            </surface>
        </collision>
        ## Descripción visual de la base.
        <visual name="baseV">
            <pose>0 0 0 0 0 0</pose>
            <geometry>
                <mesh>
                    <uri>model://arm_base/meshes/base.STL</uri>
                </mesh>
            </geometry>
        </visual>
    </link>
    ## Descripción del primer eslabón del robot diseñado.
    <link name=" link1">
        <pose>0 0 0.06 0 0 0</pose>
        <collision name=" link1C">
            <pose>0 0 0 0 0 0</pose>
            <geometry>
                <mesh>
                    <uri>model://arm_link1/meshes/link1.STL</uri>
                </mesh>
            </geometry>
            <surface>
                <contact>
```

```
                    <ode>
                        <soft_cfm>0</soft_cfm>
                        <soft_erp>0.2</soft_erp>
                        <kp>1e+13</kp>
                        <kd>1e+11</kd>
                        <max_vel>1</max_vel>
                        <min_depth>0</min_depth>
                    </ode>
                </contact>
                <friction>
                    <torsional>
                        <ode/>
                    </torsional>
                    <ode/>
                </friction>
            </surface>
        </collision>
        <visual name=" link1V">
            <pose>0 0 0 0 0 0</pose>
            <geometry>
                <mesh>
                    <uri>model://arm_link1/meshes/link1.STL</uri>
                </mesh>
            </geometry>
        </visual>
    </link>
    ## Descripción del segundo eslabón del robot diseñado.
    <link name=" link2">
        <pose>0 0 0.12 0 0 0</pose>
        <collision name=" link2C">
            <pose>0 0 0 0 0 0</pose>
            <geometry>
                <mesh>
```

```
                    <uri>model://arm_link2/meshes/link2.STL</uri>
                </mesh>
            </geometry>
            <surface>
                <contact>
                    <ode>
                        <soft_cfm>0</soft_cfm>
                        <soft_erp>0.2</soft_erp>
                        <kp>1e+13</kp>
                        <kd>1e+11</kd>
                        <max_vel>1</max_vel>
                        <min_depth>0</min_depth>
                    </ode>
                </contact>
                <friction>
                    <torsional>
                        <ode/>
                    </torsional>
                    <ode/>
                </friction>
            </surface>
        </collision>
        <visual name=" link2V">
            <pose>0 0 0 0 0 0</pose>
            <geometry>
                <mesh>
                    <uri>model://arm_link2/meshes/link2.STL</uri>
                </mesh>
            </geometry>
        </visual>
    </link>
    ## Descripción del tercer eslabón del robot diseñado.
    <link name=" link3">
        <pose>0 0 0.74 0 0 0</pose>
```

```
<collision name=" link3C">
    <pose>0 0 0 0 0 0</pose>
    <geometry>
        <mesh>
            <uri>model://arm_link2/meshes/link2.STL</uri>
        </mesh>
    </geometry>
    <surface>
        <contact>
            <ode>
                <soft_cfm>0</soft_cfm>
                <soft_erp>0.2</soft_erp>
                <kp>1e+13</kp>
                <kd>1e+11</kd>
                <max_vel>1</max_vel>
                <min_depth>0</min_depth>
            </ode>
        </contact>
        <friction>
            <torsional>
                <ode/>
            </torsional>
            <ode/>
        </friction>
    </surface>
</collision>
<visual name=" link3V">
    <pose>0 0 0 0 0 0</pose>
    <geometry>
        <mesh>
            <uri>model://arm_link2/meshes/link2.STL</uri>
        </mesh>
    </geometry>
```

```
        </visual>
    </link>
    ## Descripción de la primera articulación del robot diseñado.
    <joint name=" link1J" type="revolute">
        <pose frame="">0 0 0 0 0 0</pose>
        ## Indicación de la relación entre los eslabones involucrados.
        <parent>base</parent>
        <child> link1</child>
        <axis>
            ## Indicación de las propiedades físicas de la primera articulación.
            <dynamics>
                <damping>1</damping>
                <friction>0</friction>
                <spring_reference>0</spring_reference>
                <spring_stiffness>0</spring_stiffness>
            </dynamics>
            ## Indicación de los límites físicos de la primera articulación.
            <limit>
                <lower>-3.14</lower>
                <upper>3.14</upper>
                <effort>10</effort>
                <velocity>0.1</velocity>
            </limit>
            ## Indicación del eje afectado por la primera articulación.
            <xyz>0 0 -1</xyz>
            <use_parent_model_frame>1</use_parent_model_frame>
        </axis>
    </joint>
    ## Descripción de la segunda articulación del robot diseñado.
    <joint name=" link2J" type="revolute">
        <pose frame="">0 0 0.06 0 0 0</pose>
        <parent> link1</parent>
        <child> link2</child>
        <axis>
```

```
            <dynamics>
                <damping>1</damping>
                <friction>0</friction>
                <spring_reference>0</spring_reference>
                <spring_stiffness>0</spring_stiffness>
            </dynamics>
            <limit>
                <lower>-1.57</lower>
                <upper>1.57</upper>
                <effort>10</effort>
                <velocity>0.1</velocity>
            </limit>
            <xyz>0 1 0</xyz>
            <use_parent_model_frame>1</use_parent_model_frame>
        </axis>
    </joint>
    ## Descripción de la tercera articulación del robot diseñado.
    <joint name=" link3J" type="revolute">
        <pose frame="">0 0 0.06 0 0 0</pose>
        <parent> link2</parent>
        <child>link3</child>
        <axis>
            <dynamics>
                <damping>1</damping>
                <friction>0</friction>
                <spring_reference>0</spring_reference>
                <spring_stiffness>0</spring_stiffness>
            </dynamics>
            <limit>
                <lower>-1.57</lower>
                <upper>1.57</upper>
                <effort>10</effort>
                <velocity>0.1</velocity>
```

```
                    </limit>
                    <xyz>0 1 0</xyz>
                    <use_parent_model_frame>1</use_parent_model_frame>
                </axis>
            </joint>
            ## Indicación del plugin creado con la función de ser el actuador del robot.
            <plugin name="actuador" filename="librobot.so"></plugin>
            </model>
        </world>
</sdf>
```

C. Anexo: Algoritmos de funcionamiento del robot

C1.1. Script principal (brazo.h)

```
// Indicación de carga única de la clase.
#ifndef BRAZO
#define BRAZO
// Indicación de las librerías necesarias.
#include "listener.h"
#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
// Contenedor de las librerías de Gazebo.
namespace gazebo{
    // Definición de estructura encargada de manipular directamente las articulaciones.
    struct Union{
        physics::JointPtr joint;
        // Controlador PID y toma de tiempo.
        common::PID pid;
        common::Time tiempoActual;
        double velocidad;
        double anguloFinal;
        bool moviendo=false;
    };
    class Listener;
    class Brazo: public ModelPlugin{
        Listener *listener;
        physics::ModelPtr modelo;
```

```
        sdf::ElementPtr sdf;
        std::map<std::string,Union> uniones;
        event::ConnectionPtr conexionUpdate;
        std::string estado;
        std::map<std::string,physics::BasePtr> buscar(physics::BasePtr contenedor,
const physics::Entity::EntityType &);
        public:
            void Load (physics::ModelPtr _model,sdf::ElementPtr _sdf);
            void OnUpdate(const common::UpdateInfo & _info);
            // Cadena contenedora del estado del brazo.
            const std::string& getEstado() const{
            return estado;
        }
        void mover(std::string laUnion, double valor);
        void parametrizar(std::string laUnion,std::string tipo, double valor);
        void pintar(std::string);
        void cargarUniones();
    };
}
#endif
```

C1.2. Script principal (brazo.cpp)

```
#include "brazo.h"
#include "listener.h"
#include <gazebo/gazebo.hh>
#include <gazebo/physics/physics.hh>
#include <map>

namespace gazebo{
```

```
// Función virtual que es llamada cuando el plugin (actuador) es creado y después de cargar el mundo.
void Brazo::Load (physics::ModelPtr _model,sdf::ElementPtr _sdf){
    this->modelo=_model;
    this->sdf=_sdf;
    cargarUniones();
    listener=new Listener();
    this->estado="iniciado";
    listener->init(this);
    this->conexionUpdate=event::Events::ConnectWorldUpdateBegin(boost::bind(&Brazo::OnUpdate,this,_1));
}
// Definición de la función encargada de actualizar constantemente el estado del brazo.
void Brazo::OnUpdate(const common::UpdateInfo & _info){
    common::Time actual=modelo->GetWorld()->SimTime();
    this->estado=" Elementos: ";
    for(auto it=uniones.begin();it!=uniones.end();it++){
        Union *_union=&it->second;
        this->estado+=it->first+": A:"+std::to_string(_union->joint->Position(0));
        this->estado+=" V:"+std::to_string(_union->joint->GetVelocity(0));
        this->estado+=": ";
        if(_union->moviendo){
            common::Time ticActual=actual-_union->tiempoActual;
            double error=_union->joint->Position(0)-_union->anguloFinal;
            double absError=(error>0)?error:-1*error;
            double velocidad=0;
            velocidad=_union->pid.Update(error, ticActual);
            _union->velocidad=velocidad;
        }
        _union->joint->SetVelocity(0,_union->velocidad);
    }
}
```

```
    // Definición de la función encargada de mover las articulaciones según el angulo que se indique.
    void Brazo::mover(std::string laUnion, double valor){
        Union *_union=&uniones[laUnion];
        if(_union){
            _union->anguloFinal=valor*0.0174;
            _union->tiempoActual=0;
            _union->moviendo=true;
        }
    }
    // Definición de la función encargada de actualizar la velocidad de las articulaciones según sea necesario.
    void Brazo::parametrizar(std::string laUnion,std::string tipo, double valor){
        Union *_union=&uniones[laUnion];
        if(_union){
            if(tipo=="v"){
                _union->velocidad=valor;
                _union->moviendo=false;
            }
            else if(tipo=="VM"){
                _union->pid.SetCmdMax(valor);
                _union->pid.SetCmdMin(-1*valor);
            }
        }
    }
// Definición de la función encargada de detectar los elementos del robot.
    void Brazo::pintar(std::string tipo){
        if(tipo=="uniones"){
            this->estado=" Elementos: ";
            std::map<std::string,physics::BasePtr> uniones=buscar(this->modelo,physics::Entity::EntityType::JOINT);
            for(auto it=uniones.begin();it!=uniones.end();it++){
                this->estado+=it->first+" ";
```

```
            }
        }
    }
    std::map<std::string,physics::BasePtr> Brazo::buscar(physics::BasePtr contenedor,
const physics::Entity::EntityType & t){
        std::map<std::string,physics::BasePtr> resultado;
        unsigned int n=contenedor->GetChildCount();
        for(int i=0;i<n;i++){
            if(contenedor->GetChild(i)->HasType(t)){
                resultado[contenedor->GetChild(i)->GetName()]=contenedor-
>GetChild(i);
            }
            std::map<std::string,physics::BasePtr> resultado2=buscar(contenedor-
>GetChild(i),t);
            resultado.insert(resultado2.begin(),resultado2.end());
        }
        return resultado;
    }
    // Definición de la función encargada de aplicar el controlador a cada articulación
después que se detecten.
    void Brazo::cargarUniones(){
        std::map<std::string,physics::BasePtr> uniones=buscar(this-
>modelo,physics::Entity::EntityType::JOINT);
        for(auto it=uniones.begin();it!=uniones.end();it++){
            Union _union;
            _union.joint=boost::static_pointer_cast<physics::Joint>(it->second);
            // Parámetros del controlador PID (p, i, d, imax, imin, cmdMax, cmdMin).
            _union.pid.Init(2, 1, 1, 0, 0,_union.joint->GetVelocityLimit(0),-1*_union.joint-
>GetVelocityLimit(0));
            _union.tiempoActual=0;
            this->uniones[it->first]=_union;
        }
    }
```

```
    // Registro del plugin (actuador).
    GZ_REGISTER_MODEL_PLUGIN(Brazo);
}
```

C2.1. Enlace IPO (listener.h)

```
#ifndef LISTENER
#define LISTENER

#include "ros/ros.h"
#include "ros/callback_queue.h"
#include "ros/subscribe_options.h"
#include "std_msgs/String.h"
#include <stdio.h>
#include <thread>
#include "brazo.h"

namespace gazebo{
    class Brazo;
    class Listener{
    private:
        // Puntero manejador de nodos.
        std::unique_ptr<ros::NodeHandle> nodo;
        // Definición de suscriptor.
        ros::Subscriber subscriber;
        // Definición de publicador.
        ros::Publisher publisher;
        // Definición de colas.
        ros::CallbackQueue cola;
        ros::CallbackQueue cola2;
        // Hilo manejador de colas.
        std::thread threadColas;
```

```
        Brazo * bot;
    public:
        void init(Brazo *);
        void listener(const std_msgs::String::ConstPtr& msg);
        // Funciones invocadas al detectar cambios en los topics.
        static void conexion(const ros::SingleSubscriberPublisher&);
        static void desconexion(const ros::SingleSubscriberPublisher&);
        // Función de hilo verificador.
        void thread();
    };
}
#endif
```

C2.2. Enlace IPO (listener.cpp)

```
#include "listener.h"
#include "ros/ros.h"
#include "ros/callback_queue.h"
#include "ros/subscribe_options.h"
#include "std_msgs/String.h"
#include "Comandos.h"
#include <stdio.h>
#include <gazebo/gazebo.hh>

namespace gazebo{
    void Listener::init(Brazo * bot){
        this->bot=bot;
        // Indicación de inicializar ROS con un nodo estándar, si no lo está,
        if(!ros::isInitialized()){
            int argc=0;
            char **argv=NULL;
            ros::init(argc,argv, "gazebo_client", ros::init_options::NoSigintHandler);
        }
```

```
        // Reseteo del nuevo nodo.
        this->nodo.reset(new ros::NodeHandle("gazebo_client"));
        // Definición de suscriptor que recibirá los mensajes tipo String de afuera.
        ros::SubscribeOptions so=ros::SubscribeOptions::create<std_msgs::String>(
            // Topic de publicación.
            "/brazo",
            // Numero de parámetros.
            1,
            // Enlace con la función Listener.
            boost::bind(&Listener::listener, this, _1),
            ros::VoidPtr(),
            // Envío de mensaje a cola para su proceso.
            &this->cola
            );
        // El nodo inicializado anteriormente se convierte en el suscriptor.
        this->subscriber=this->nodo->subscribe(so);
        // Definición de publicador encargado de informar el estado de conexión de
nodos al tema y enviará su petición a la cola.
        ros::AdvertiseOptions ad=ros::AdvertiseOptions::create<std_msgs::String>(
            "/brazo_m",
            1,
            &this->conexion,
            &this->desconexion,
            ros::VoidPtr(),
            &this->cola2
            );
        this->publisher=this->nodo->advertise(ad);
        this->threadColas=std::thread(std::bind(&Listener::thread, this));
    }
    void Listener::listener(const std_msgs::String::ConstPtr& msg){
        std::string m=msg->data.c_str();
        Comandos::procesar(m, this->bot);
    }
```

```
    // Funciones informadoras del estado de conexión de los nodos.
    void Listener::conexion(const ros::SingleSubscriberPublisher&){
        ROS_INFO("Conectado");
    }
    void Listener::desconexion(const ros::SingleSubscriberPublisher&){
        ROS_INFO("Desconectado");
    }
    // Definición de la función del hilo verificador del estado de los nodos y llamará el
primer mensaje de la cola para inicializarlo.
    void Listener::thread(){
        static const double timeout=0.01;
        while(this->nodo->ok()){
            this->cola.callAvailable(ros::WallDuration(timeout));
                std_msgs::String m;
                std::stringstream ms;
                ms<<this->bot->getEstado()<<" ";
                m.data=ms.str();
                this->publisher.publish(m);
        }
    }
}
```

C3.1. Instrucciones (comandos.h)

```
#ifndef COMANDOS_H_
#define COMANDOS_H_

#include <stdio.h>
#include "std_msgs/String.h"
#include "brazo.h"
// Función encargada de dividir en partes las líneas comandos recibidas.
std::vector<std::string> split(const std::string &c, char d);
namespace gazebo{
```

```
    class Comandos{
        public:
            static bool procesar(std::string &comando, Brazo *);
    };
}
#endif
```

C3.2. Instrucciones (comandos.cpp)

```
#include "comandos.h"
#include <stdio.h>
#include "std_msgs/String.h"
#include <vector>
#include "brazo.h"

namespace gazebo{
    bool Comandos::procesar(std::string &comando, Brazo * bot){
        // Vector de cadenas.
        std::vector<std::string> partes=split(comando, ' ');
        // Definición de las funciones de los comandos recibidos.
        switch(partes[0][0]){
            // Detectar los elementos del robot.
            case 'd':
                if(partes.size()>1){
                    bot->pintar(partes[1]);
                }
                break;
            // Aplicar fuerza de movimiento a una articulación especifica.
            case 'p':
                if(partes.size()>3){
                    bot->parametrizar(partes[1],partes[2],std::stod(partes[3]));
                }
```

```
                    break;
                // Mover una articulación especifica en grados.
                case 'm':
                    if(partes.size()>2){
                        bot->mover(partes[1],std::stod(partes[2]));
                    }
                    break;
                // Ejecutar archivo con secuencia de comandos.
                case 'e':
                    std::ifstream stream(partes[1]);
                    if(stream.good()){
                        std::string linea;
                        while(!stream.eof()){
                            linea.clear();
                            std::getline(stream, linea);
                            Comandos::procesar(linea, bot);
                        }
                        stream.close();
                    }
                    break;
            }
            return false;
        }
}
// Definición de la función encargada de dividir en partes las líneas comandos recibidas.
std::vector<std::string> split(const std::string &c, char d){
    std::vector<std::string> resultado;
    std::stringstream cs(c);
    std::string parte;
    while(std::getline(cs, parte, d)){
        resultado.push_back(parte);
    }
    return resultado;
}
```

D. Anexo: Algoritmos de la interfaz gráfica

D1. Compilador de CMake (CMakeLists.txt)

```
cmake_minimum_required(VERSION 3.0.2)
project(interfaz)
set(CMAKE_AUTOMOC ON)
find_package(Qt5Core REQUIRED)
find_package(Qt5Widgets)
include_directories(
      src
      ${Qt5Core_INCLUDE_DIRS}
      /opt/ros/noetic/include
)
set(CMAKE_AUTOMOC ON)
add_definitions(${Qt5Core_DEFINITIONS})
add_executable(interfaz src/main.cc src/mainDialog.cc )
target_link_libraries(  interfaz    ${Qt5Widgets_LIBRARIES}  /opt/ros/noetic/lib/libroscpp.so
/opt/ros/noetic/lib/libroslib.so
/opt/ros/noetic/lib/librosconsole.so /opt/ros/noetic/lib/libroscpp_serialization.so)
```

D2.1. Script interfaz gráfica (main.cc)

```
#include <iostream>
#include <QtWidgets/QApplication>
#include <QtWidgets/QDialog>
```

```
#include "mainDialog.h"
// Determinación de la función que creará la aplicación interna de Qt.
QCoreApplication* createApplication(int &argc, char *argv[]){
   for (int i = 1; i < argc; ++i)
      if (!qstrcmp(argv[i], "-no-gui"))
         return new QCoreApplication(argc, argv);
   return new QApplication(argc, argv);
}
MainDialog *dialog;
// Arranque de la aplicación de Qt.
int main(int argc, char* argv[]){
   QScopedPointer<QCoreApplication> app(createApplication(argc, argv));
   dialog = new MainDialog;
   dialog->show();
   int res=app->exec();
   return res;
}
```

D2.2. Script interfaz gráfica (mainDialog.h)

```
#ifndef MAINDIALOG_H
#define MAINDIALOG_H

#include <iostream>
#include <QtWidgets/QDialog>
#include <QtWidgets/QLabel>
#include <QtWidgets/QLineEdit>
#include <QtWidgets/QPushButton>
#include <QtWidgets/QSlider>
#include "ros/ros.h"
#include "std_msgs/String.h"
```

```
class MainDialog : public QDialog{
    Q_OBJECT
    public:
        MainDialog(QWidget *parent = 0);
    public slots:
        void mover1J();
        void mover2J();
        void mover3J();
        void smover1J();
        void smover2J();
        void smover3J();
        void tarea();
        void home();
    private:
        QLabel *lDescripcion;
        QLabel *lLink1J;
        QLabel *lLink2J;
        QLabel *lLink3J;
        QLineEdit *tLink1J;
        QLineEdit *tLink2J;
        QLineEdit *tLink3J;
        QSlider *sLink1J;
        QSlider *sLink2J;
        QSlider *sLink3J;
        QPushButton *bTarea;
        QPushButton *bHome;
        ros::Publisher publicador;
        void enviarRos(std::string);
};
#endif
```

D2.3. Script interfaz gráfica (mainDialog.cc)

```
#include "mainDialog.h"
// Determinación del método constructor.
MainDialog::MainDialog(QWidget *parent): QDialog(parent){
    // Determinación del objeto QLabel (Etiquetas de texto).
    this->lDescripcion = new QLabel(this);
    this->lDescripcion->setGeometry(QRect(0, 10, 220, 20));
    this->lDescripcion->setText("Operación en grados");
    this->lDescripcion->setAlignment(Qt::AlignCenter);
    this->lLink1J = new QLabel(this);
    this->lLink1J->move(20, 50);
    this->lLink1J->setText("Link1J");
    this->lLink2J = new QLabel(this);
    this->lLink2J->move(20, 110);
    this->lLink2J->setText("Link2J");
    this->lLink3J = new QLabel(this);
    this->lLink3J->move(20, 170);
    this->lLink3J->setText("Link3J");
    // Determinación del objeto QLineEdit (Recibe valor en grados ingresado, tipo de
variable QString).
    this->tLink1J = new QLineEdit(this);
    connect(tLink1J, SIGNAL(textChanged(QString)), this, SLOT(mover1J()));
    this->tLink1J->setGeometry(QRect(80, 40, 120, 30));
    this->tLink2J = new QLineEdit(this);
    connect(tLink2J, SIGNAL(textChanged(QString)), this, SLOT(mover2J()));
    this->tLink2J->setGeometry(QRect(80, 100, 120, 30));
    this->tLink3J = new QLineEdit(this);
    connect(tLink3J, SIGNAL(textChanged(QString)), this, SLOT(mover3J()));
    this->tLink3J->setGeometry(QRect(80, 160, 120, 30));
    // Determinación del objeto QSlider (Recibe posición del Slider, tipo de variable int).
    this->sLink1J = new QSlider(this);
    connect(sLink1J, SIGNAL(valueChanged(int)), this, SLOT(smover1J()));
```

```
this->sLink1J->setGeometry(QRect(20, 70, 180, 30));
this->sLink1J->setMinimum(-180);
this->sLink1J->setMaximum(180);
this->sLink1J->setValue(0);
this->sLink1J->setOrientation(Qt::Horizontal);
this->sLink2J = new QSlider(this);
connect(sLink2J, SIGNAL(valueChanged(int)), this, SLOT(smover2J()));
this->sLink2J->setGeometry(QRect(20, 130, 180, 30));
this->sLink2J->setMinimum(-90);
this->sLink2J->setMaximum(90);
this->sLink2J->setValue(0);
this->sLink2J->setOrientation(Qt::Horizontal);
this->sLink3J = new QSlider(this);
connect(sLink3J, SIGNAL(valueChanged(int)), this, SLOT(smover3J()));
this->sLink3J->setGeometry(QRect(20, 190, 180, 30));
this->sLink3J->setMinimum(-90);
this->sLink3J->setMaximum(90);
this->sLink3J->setValue(0);
this->sLink3J->setOrientation(Qt::Horizontal);
// Determinación del objeto QPushButton.
this->bTarea = new QPushButton(this);
connect(bTarea, SIGNAL(clicked()), this, SLOT(tarea()));
this->bTarea->setText(tr("Tarea"));
this->bTarea->move(20, 230);
this->bTarea->setAutoDefault(false);
this->bTarea->setDefault(false);
this->bHome = new QPushButton(this);
connect(bHome, SIGNAL(clicked()), this, SLOT(home()));
this->bHome->setText(tr("Home"));
this->bHome->move(120, 230);
this->bHome->setAutoDefault(false);
this->bHome->setDefault(false);
// Definición del título de la ventana de la interfaz gráfica.
setWindowTitle(tr("Operación"));
```

```
    if(!ros::isInitialized()){
        int argc=0;
        char **argv=NULL;
        ros::init(argc,argv, "MI_APP", ros::init_options::NoSigintHandler);
    };
    ros::NodeHandle n;
    publicador = n.advertise<std_msgs::String>("brazo", 1000);
}
// Definición de función encargada de sincronizar los objetos, convertir las variables al tipo requerido y unir el valor recibido a la cadena.
void MainDialog::mover1J(){
    QString v1q=tLink1J->text();
    int v1i=v1q.toInt();
    sLink1J->setValue(v1i);
    std::string v1s = std::to_string(v1i);
    enviarRos("m link1J "+v1s);
}
void MainDialog::mover2J(){
    QString v2q=tLink2J->text();
    int v2i=v2q.toInt();
    sLink2J->setValue(v2i);
    std::string v2s = std::to_string(v2i);
    enviarRos("m link2J "+v2s);
}
void MainDialog::mover3J(){
    QString v3q=tLink3J->text();
    int v3i=v3q.toInt();
    sLink3J->setValue(v3i);
    std::string v3s = std::to_string(v3i);
    enviarRos("m link3J "+v3s);
}
void MainDialog::smover1J(){
    int v1i=sLink1J->value();
```

```
	QString v1q=v1q.number(v1i);
	tLink1J->setText(v1q);
}
void MainDialog::smover2J(){
	int v2i=sLink2J->value();
	QString v2q=v2q.number(v2i);
	tLink2J->setText(v2q);
}
void MainDialog::smover3J(){
	int v3i=sLink3J->value();
	QString v3q=v3q.number(v3i);
	tLink3J->setText(v3q);
}
// Definición de función encargada de enviar la cadena de comandos guardada previamente.
void MainDialog::tarea(){
	enviarRos("e /home/sebastian/proyecto /src/brazo/programas/tarea");
}
void MainDialog::home(){
	enviarRos("e /home/sebastian/proyecto/src/brazo/programas/home");
}
// Definición de función encargada de publicar el mensaje en el topic (Recibe la cadena de comandos, tipo de variable std::string).
void MainDialog::enviarRos(std::string mensaje){
	std_msgs::String msg;
	msg.data = mensaje;
	publicador.publish(msg);
}
```

Bibliografía

[1] ROS. [Sitio web]. [Consultado: 1 de septiembre de 2020]. Disponible en: https://www.ros.org

[2] ROS & Gazebo at the DRC Finals. [Sitio web]. [Consultado: 1 de septiembre de 2020]. Disponible en: https://www.osrfoundation.org/ros-gazebo-at-the-drc-finals/

[3] Cuevas Castañeda, C. C. (2016). Ros-gazebo. una valiosa Herramienta de Vanguardia para el Desarrollo de la Robótica. Publicaciones E Investigación, 10, 145-160. https://doi.org/10.22490/25394088.1593

[4] ¿Qué es la robótica? (Introducción a la robótica y microcontroladores). (2017, 20 noviembre). Hacia el Espacio. http://haciaelespacio.aem.gob.mx/revistadigital/articul.php?interior=733

[5] S. Ivaldi, J. Peters, V. Padois and F. Nori, "Tools for simulating humanoid robot dynamics: A survey based on user feedback," 2014 IEEE-RAS International Conference on Humanoid Robots, Madrid, 2014, pp. 842-849, doi: 10.1109/HUMANOIDS.2014.7041462.

[6] Araújo, A., Portugal, D., Couceiro, M., Sales, J., & Rocha, R. (2014). Desarrollo de un robot móvil compacto integrado en el middleware ROS. Revista Iberoamericana de Automática e Informática industrial, 11(3), 315-326. https://doi.org/10.1016/j.riai.2014.02.009

[7] Rojas Bustos, Juan Pablo. “ENTORNO VIRTUAL PARA LA SIMULACIÓN DE UN ROBOT MÓVIL SOBRE EL FRAMEWORK ROS”, Universidad Piloto de Colombia (Colombia), 2015.

[8] S. Ergur and M. Ozkan, "Trajectory planning of industrial robots for 3-D visualization a ROS-based simulation framework," 2014 IEEE International Symposium on Robotics and Manufacturing Automation (ROMA), Kuala Lumpur, 2014, pp. 206-211, doi: 10.1109/ROMA.2014.7295889.

[9] Escobar Naranjo, Juan Camilo. "DISEÑO DE SISTEMAS DE CONTROL INDUSTRIAL DE ROBOTS BASADOS EN INDUSTRIA 4.0", Universidad Técnica de Ambato (Ecuador), 2019.

[10] Melero Cazorla, Pablo. "Teleoperación de un brazo robot mediante el sensor Kinect", Universidad de Almería (España), 2014.

[11] Martínez Rozas, Simón Ernesto. “Modelado y simulación de robots terrestres para la inspección del alcantarillado”, Escuela Técnica Superior de Ingeniería (España), 2018.

[12] Samper Escudero, José Luis. “ANALISIS DE UN BRAZO ROBÓTICO CON GAZEBO Y ROS PARA TAREAS DE INSPECCIÓN REMOTA EN EL CERN”, Escuela Técnica Superior de Ingenieros Industriales (España), 2016.

[13] Reyes Cortés, F. (2011). *Robótica: control de robots manipuladores* («Revisado», «1» ed.). Alfaomega Grupo Editor.

[14] Siciliano, B., Sciavicco, L., Villani, L., & Oriolo, G. (2009). Robotics Modelling Planning. Springer.

[15] Gudiño-Lau, J. A., Alcalá-Rodríguez, J., Narrarro, H., Velez-Díaz, D., & Charre-Ibarra, S. (2018). Diseño y modelo cinemático de un robot delta para el diagnóstico y rehabilitación. *XIKUA Boletín Científico De La Escuela Superior De Tlahuelilpan*, *6*(11). https://doi.org/10.29057/xikua.v6i11.2764

[16] Araujo, F., Ayala, L., & Bermeo, C. Universidad Politécnica Salesiana, (2014). *ROBOTS ANTROPOMÓRFICOS*. DOCPLAYER. https://docplayer.es/13485497-Robots-antropomorficos.html

[17] Amat Verdú, Octavio, "Diseño de un manipulador controlado conel microcontrolador Arduino", Universidad Politécnica de Valencia (España), 2017.

[18] Ramírez Leyva, F. (2012, marzo). Robótica: Modelado cinemática de Robots. Universidad Tecnológica de la Mixteca. http://www.utm.mx/~hugo/robot/Robot2.pdf

[19] Robotics Toolbox. (2020, 27 abril). Peter Corke. https://petercorke.com/toolboxes/robotics-toolbox/

[20] Enterprise Open Source and Linux. (s. f.). Ubuntu. Recuperado 23 de septiembre de 2020, de https://ubuntu.com

[21] Documentation - ROS Wiki. (s. f.). wiki.ROS.org. Recuperado 1 de septiembre de 2020, de http://wiki.ros.org

[22] O. (s. f.). Gazebo. Gazebo. Recuperado 1 de septiembre de 2020, de http://gazebosim.org

[23] O. (s. f.-b). SDFormat Home. SDFormat. Recuperado 1 de septiembre de 2020, de http://sdformat.org

[24] T.Q. Company. (s. f.). Qt | Cross-platform software development for embedded & desktop. Qt Company. Recuperado 1 de septiembre de 2020, de https://www.qt.io/

[25] J., C. (2020, 28 julio). Descripción y uso de los archivos ~/.bashrc y /etc/bashrc. zeppelinux. https://www.zeppelinux.es/descripcion-y-uso-de-los-archivos-bashrc-y-etc-bashrc/