UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

"Implementación de detección y seguimiento de objetos en un brazo robótico UR5 mediante visión por computador para posicionamiento preciso"

TRABAJO FIN DE GRADO

Septiembre - 2023

AUTOR: Mario Sánchez Boix

DIRECTOR/ES: Arturo Gil Aparicio

Luis Miguel Jiménez García


UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE
ESCUELA POLITÉCNICA SUPERIOR DE ELCHE
GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL
TRABAJO FIN DE GRADO
SEPTIEMBRE - 2023
"IMPLEMENTACIÓN DE DETECCIÓN Y SEGUIMIENTO DE OBJETOS EN UN BRAZO ROBÓTICO UR5 MEDIANTE VISIÓN POR COMPUTADOR PARA POSICIONAMIENTO PRECISO"
AUTOR: MARIO SÁNCHEZ BOIX
DIRECTOR/ES: ARTURO GIL APARICIO
LUIS MIGUEL JIMÉNEZ GARCÍA

ÍNDICE

CAPÍTULO 1: INTRODUCCIÓN

El control y posicionamiento del extremo del robot utilizando una cámara en el robot serie puede ser una tarea compleja e imprecisa, debido a los factores tanto internos como externos que que provocan que el control no sea preciso.

En este Trabajo Fin de Grado se emplea las facilidades que nos proporciona *Robot Operating System* (ROS) para realizar dicho control y posicionamiento, utilizando en simulación el robot serie disponible en el labortaorio Robótica, Automática y Visión por Computador de la Universidad Miguel Hernández de Elche (UMH).

1.1 MOTIVACIÓN

Para el Trabajo Final de Grado, se buscaba un trabajo relacionado con la Robótica y el control de un brazo robótico, por este motivo, este trabo me brindó esa oportunidad. Mediante este trabajo he descubierto un nuevo sistema operativo que he aprendido a utilizar para la programación de robots que se emplea en muchas industrias, y para la programación de ese control he aprendido a programar en Python, y tras unas pequeñas modificaciones en la propuesta, decidí adaptarlo a lo que había estudiado.

1.2 OBJETIVOS

El presente trabajo tiene como objetivo principal el desarrollo de un sistema robótico de posicionamiento a través de ROS y guiado por visión artificial. Mediante esta herramienta permite al usuario colocar el brazo en cualquier posición y el objeto a coger en otra posición y orientación, y que éste se coloque y oriente de forma que coja el objeto correctamente. El trabajo se enmarca dentro de las líneas de investigación del Departamento de Ingeniería de Sistemas y Automática.

Además, tiene como objetivo dar a conocer como empezar a programar con ROS, como funciona su sistema y de la capacidad que presenta para controlar un robot:

- Familiarización con el entorno de trabajo de Ubuntu.
- Familiarización con el lenguaje interpretado Python y aprendizaje de programación en C++ orientada a objetos.
- Aprendizaje y desarrollo de sistemas basados en ROS.
- Familiarizarse, aprendizaje y uso avanzado de las herramientas necesarias para simular un robot con ROS (*Gazebo* y *RViz*).

- Reconocimiento de objetos 3D en base a la librería de visión *OpenCV*.
- Estudio del funcionamiento del paquete de control de trayectorias de robots que se localiza disponible en *MoveIt!*.
- Creación de un conjunto de programas que contengan nodos, que permitan la configuración de una cámara y el posicionamiento y orientación del brazo robótico.
- Conexión entre programas que permita que la detección del objeto, posicionamiento y orientación sea la correcta.

1.3 DESCRIPCIÓN

Como se ha mencionado en los objetivos de este Trabajo Fin de Grado, el principal objetivo es el de ser capaz de programar un sistema robótico para que realice una tarea de manipulación en un entorno de simulación. Para abordar ese objetivo y los objetivos secundarios, se ha redactado una memoria dividida en 8 capítulos.

En el primer capítulo, como se ha podido contemplar, se redacta una pequeña introducción del trabajo con los objetivos que se pretende alcanzar al final de éste.

En el segundo capítulo, se realiza una breve explicación de los robots colaborativos, profundizando en los robots serie, ya que, es el tipo de robot que se trabaja.

Tomando de partida el segundo capítulo, en el tercer capítulo se detalla el material que se va a utilizar en simulación para desarrollar el objetivo del trabajo.

A continuación, en el capítulo cuatro, se describe el Sistema Operativo Robótico, tanto su definición como las herramientas con las que trabaja, a modo de introducción para los siguientes capítulos.

Seguidamente, como complemento del capítulo anterior, en el capítulo cinco, se comenta las distintas herramientas matemáticas que se emplean para el cálculo de las trayectorias, como las funciones utilizadas de la biblioteca de *OpenCV*.

Una vez explicado en detalle todo el material que se va a emplear, en el sexto capítulo se explica el proceso en software que se ha llevado a cabo para simular el robot con las distintas herramientas añadidas (cámara web y ventosa al vacío (*vacuum gripper*)), junto con los programas adecuados para su funcionamiento.

En el capítulo siete, se analiza los resultados obtenidos a través de vídeos y capturas de la simulación, para explicar las distintas aplicaciones que puede presentar este funcionamiento.

El octavo capítulo contiene las conclusiones a las que se ha llegado desprendidas del trabajo, las distintas mejoras que se podría realizar sobre este trabajo.

En el capítulo 9, se comenta la bibliografía que se ha utilizado.

Por último, los capítulos 10 y 11, que son los anexos y la lista de figuras, respectivamente, contienen documentación para que el usuario no tenga problemas a la hora de utilizar ROS y del robot empleado, y tener una guía clara del proceso.

CAPÍTULO 2: INTRODUCCIÓN A LA ROBÓTICA

En este segundo capítulo se introduce al mundo de la robótica, donde se hablará de los distintos tipos de robots que existen y su morfología.

Pero antes de clasificar los distintos robots que existen, ¿qué se entiende por robot industrial? La definición de robot industrial no es algo fácil de establecer, ya que, presenta varias dificultades, ya sea por la diferencia conceptual existente entre robot y manipulador según el mercado japonés y euroamericano, como el concepto occidental. Debido a la continua evolución que sufre la robótica, su definición es compleja de establecer y como no se llegaba a una definición clara, distintas asociaciones trataron de establecer la definición de robot para que sea comúnmente aceptada sin que haya discrepancias. A continuación, se muestra las definiciones más claras y precisas establecidas por las asociaciones/organizaciones.

Según la *Asociación de Industrias Robóticas (RIA)*, la definición de robot, que es la definición comúnmente aceptada, es la siguiente:

- Un robot industrial es un manipulador multifuncional reprogramable, capaz de mover materias, piezas, herramientas o dispositivos especiales, según trayectorias variables, programadas para realizar diversas tareas.

La *Organización Internacional de Estándares (ISO)* propone la siguiente definición, que varía ligeramente de la anterior:

- Manipulador multifuncional reprogramable con varios grados de libertad, capaz de manipular materias, piezas, herramientas o dispositivos especiales según trayectorias variables programadas para realizar tareas diversas.

Para hacer la definición más completa, la *Asociación de Normalización (AFNOR)* estableció la descripción de manipulador y a partir de éste, la de robot:

- Manipulador: mecanismo formado generalmente por elementos en serie, articulados entre sí, destinados al agarre y desplazamiento de objetos. Es multifuncional y puede ser gobernado directamente por un operador humano o mediante dispositivo lógico.
- Robot: manipulador automático servocontrolado, reprogramable, polivalente, capaz de posicionar y orientar piezas, útiles o dispositivos especiales, siguiendo trayectorias variables reprogramables, para la ejecución de tareas variadas. Normalmente tiene la forma de uno o varios brazos terminados en muñeca. Su

unidad de control incluye un dispositivo de memoria y ocasionalmente de percepción del entorno. En la mayoría de las ocasiones, su uso es el de realizar una tarea de manera cíclica, pudiéndose adaptar a otra sin cambios permanentes en su materia.

Y para finalizar, según la *Federación Internacional de Robótica (IFR)*, establece la siguiente descripción:

- El robot industrial de manipulación se entiende como una máquina de manipulación automática, reprogramable y multifuncional con tres o más ejes que puedes posicionar y orientar materias, piezas, herramientas o dispositivos especiales para la ejecución de trabajos diversos en las diferentes etapas de la producción industrial, ya sea en una posición fija o en movimiento.

Si se analiza todas las definiciones se obtiene en común que un robot industrial es un brazo mecánico que tiene la capacidad de manipular objetos y que incorpora un control relativamente complejo y reprogramable.

2.1 CLASIFICACIÓN DE LOS ROBOTS

Existen numerosas formas de clasificar a un robot, pero en este caso, se destacan tres de las más importantes, las cuales se pueden separar según sean sus funciones o sector al que está dedicado, según su generación y según la movilidad que presentan. Primeramente, se divide según la generación del robot, por lo tanto, se habla de su cronología:

- 1ª Generación: Son los conocidos robots manipuladores, los cuales repiten la tarea programada secuencialmente, mediante movimientos muy limitados y sin recoger información del entorno.
- 2ª Generación: Son los conocidos como robots en aprendizaje, porque adquieren información, de forma limitada, del entorno y actúan en consecuencia con movimientos más complejos. Suelen ser de mayor tamaño que los de 1º Generación
- 3ª Generación: También llamados robots reprogramables, son aquellos que están equipados con sensores y son capaces de adquirir percepciones del entorno. Se controlan con un ordenador en los que se usan lenguajes de programación para variar sus funciones.

- 4ª Generación: En esta etapa aparecen los robots inteligentes, que son los robots capaces de controlar el proceso y su entorno automáticamente en tiempo real gracias a que se ajustan y a las mejoras de los sistemas sensoriales.
- 5ª Generación: Por último, es la etapa que se encuentra en desarrollo, debido a que está relacionada con la inteligencia artificial. Gracias al alto desarrollo tecnológico que hacen que los robots sean autónomos, imitando a los seres humanos y capacitados para satisfacer las necesidades humanas.

A continuación, la otra distinción es dividir los robots según su movilidad, es decir, su capacidad de movimiento y toma de decisiones:

- Brazos robóticos: Presentan una capacidad muy reducida, pero son perfectos para la manipulación de productos, herramientas, empaquetamiento. Se tratará de los brazos robóticos en el siguiente apartado.

Figura 1. Ejemplo de brazo robótico, ABB

- Vehículos de guiado automático (AGV): Son los robots que se mueven por una pista guiada y que necesitan la supervisión de un humano.

Figura 2. Ejemplo de robot AGV

- Robots móviles autónomos: Son los llamados AMR (*Autonomous Mobile Robots*), robots que se pueden mover y tomar decisiones en tiempo real, gracias a los sensores incorporados y un equipo de procesamiento que permiten llevar a cabo sus funciones.

Figura 3. Ejemplo de robot móvil

- Humanoides: Estos robots son un tipo de AMR, pero que presentan formas y funciones humanas.

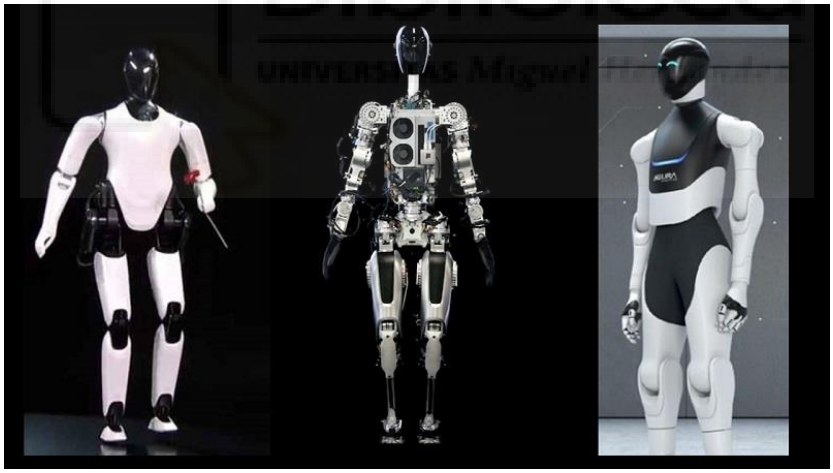


Figura 4. Ejemplo de robot humanoide

La última distinción es dependiendo de la función que realicen o del sector para el que han sido concebidos:

- Robots industriales: Se encargan de la manipulación automática en la cadena de producción y realizan actividades rutinarias y repetitivas. Son reprogramables y multifuncionales, como se ha definido previamente, y se busca reducir el tiempo de fabricación.

- Robots de servicio: Suelen ser robots móviles, que se utilizan en entornos no controlados, cuya función es la de asistir a las personas en trabajos repetitivos, sucios, etc. Dentro de esta clasificación podemos encontrar robots domésticos, de investigación y de exploración.
- Robots Militares: Desarrollados para cumplir acciones militares específicas, sobre todo para la asistir a los ejércitos en sus operaciones.
- Robots médicos: Según en que ámbito de la medicina se encuentre, pueden ser robots de alta precisión para cirugía o automatismos para personas con dependencia. Sirven de apoyo al sector sanitario. También, hay que destacar los nano robots, que se emplean en medicina para combatir enfermedades concretas.
- Robots educativos: Robots que se suelen encontrar en las escuelas cuya principal función es la de entretener a quien lo utilice, además de adquirir conocimientos de la materia y reforzar habilidades cognitivas.

Existen otras distinciones como pueden ser, según su nivel de inteligencia, el entorno de trabajo o su estructura. Pero las explicadas anteriormente, son las más importantes a la hora de clasificar un robot.

2.2 BRAZO ROBÓTICO INDUSTRIAL

En este caso, para el trabajo, se ha utilizado, según las clasificaciones que previamente se han comentado, un robot industrial, que, según su movilidad, se trata de un brazo robótico. Lo que se documenta en este apartado son todos los elementos constitutivos que componen un brazo robótico en detalle, es decir, se examina la estructura mecánica, haciendo referencia a los distintos tipos de articulaciones entre dos eslabones y se analiza los elementos terminales.

Lo primero, se define el concepto de brazo robótico, que se trata de un tipo de brazo mecánico programable, similar a un brazo humano, y puede ser el total del mecanismo o puede ser parte de un robot más complejo.

A continuación, se explica la estructura mecánica que presenta un brazo robótico, partiendo de que la mayoría de los brazos robóticos industriales presentan una similitud con las extremidades superiores del cuerpo humano, se hace esta referencia porque en ocasiones se usan términos como hombro, codo, muñeca.

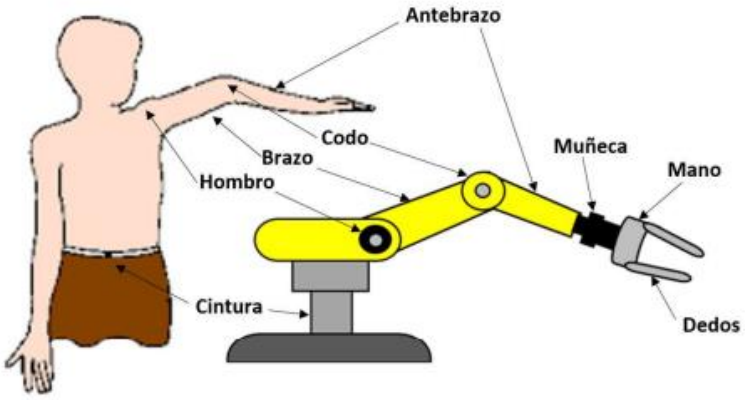


Figura 5. Relación brazo humano y brazo robótico.
Fuente: Tecnología-técnica

En el apartado, se ha mencionado el concepto de eslabón, y para el correcto entendimiento de la memoria, se puede definir el eslabón como una parte rígida del robot, que es conectado a otro mediante una junta o articulación, permiten el movimiento relativo entre estos.

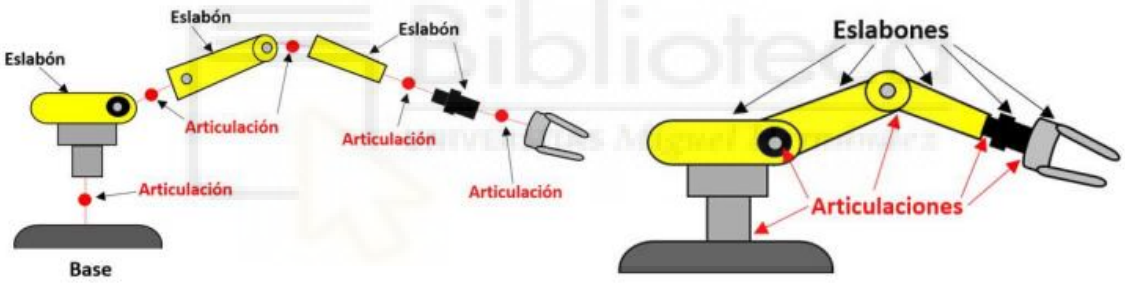


Figura 6. Descomposición del brazo robótico (eslabones y articulaciones)
Fuente: Tecnología-técnica

Dentro de la estructura mecánica, lo más importante es la articulación, ya que es la encargada de permitir el movimiento relativo entre eslabones, y dependiendo de cómo se haya construido el robot, la articulación puede ser lineal (deslizante, traslacional o prismática), donde un eslabón desliza sobre un eje solidario al anterior creando un movimiento de desplazamiento, o rotacional, si un eslabón gire en torno a un eje solidario al anterior generando un movimiento de giro, y también, se puede realizar una combinación de ambos movimientos. Cada uno de los movimientos independientes que cada articulación puede hacer con respecto a la anterior, de denomina grado de libertad (GDL), y se puede definir como cada uno de los movimientos básicos que definen la movilidad de un robot, en un espacio tridimensional. Se puede recoger esos movimientos en 6 clases básicas de grado de libertad:

- Rotacional: Es la articulación más empleada, presenta un grado de libertad y consiste en una rotación alrededor del eje de la articulación.
- Prismática: Al igual que la rotacional, tiene un grado de libertad y es una traslación a lo largo del eje de la articulación.
- Cilíndrica: En este caso, tiene dos grados de libertad, ya que, combina una rotación junto con una traslación.
- Planar: Presenta dos grados de libertad, y se caracteriza por presentar un movimiento de desplazamiento en un plano.
- Esférica o Rótula: Tiene tres grados de libertad y combina 3 giros en tres direcciones perpendiculares al espacio.
- Tornillo: Contiene un grado de libertad y consiste en la traslación a lo largo de un eje roscado.

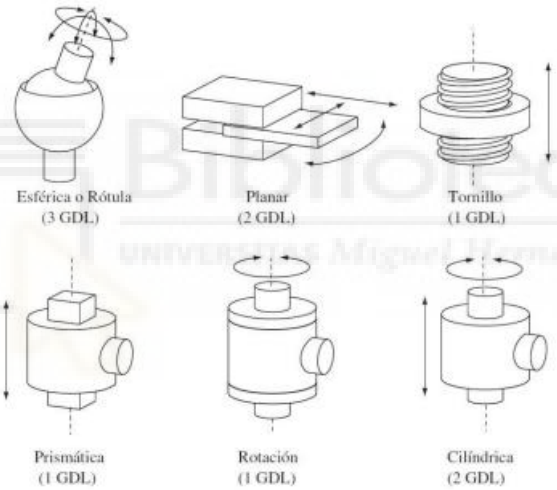


Figura 7. Las 6 clases básicas de GDL
Fuente: Tecnología-técnica

Al combinarse el conjunto de eslabones con las distintas articulaciones, se llega a la denominada cadena cinemática, que puede ser cerrada o abierta, si es cerrada significa que se puede llegar a cualquier eslabón desde cualquier otro como mínimo con dos caminos, y si es abierta significa que cada eslabón se conecta exclusivamente al anterior quedando el primero conectado al soporte y el último queda libre para conectar cualquier elemento terminal.

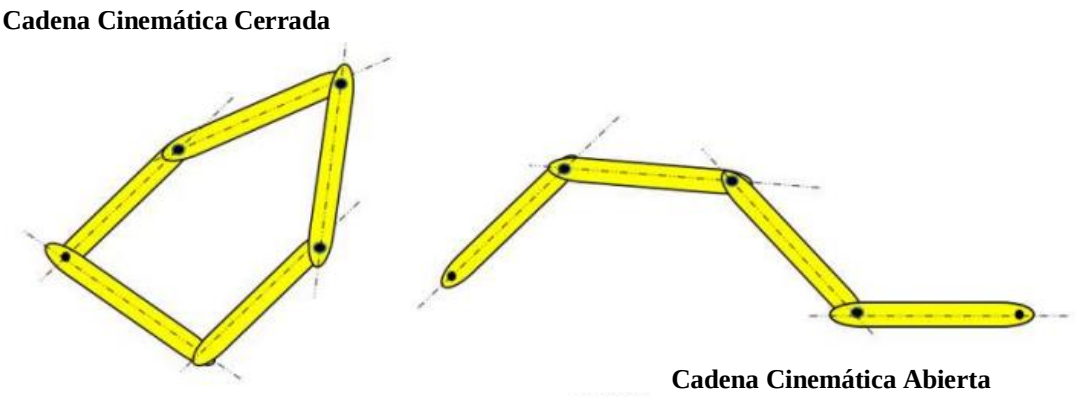


Figura 8. Descripción de cadenas cinemáticas
Fuente: Tecnología-técnica

Estas combinaciones en un robot dan lugar a diferentes configuraciones, con características que permiten adaptar al robot a distintas aplicaciones finales. Además, al realizar esas combinaciones, también se añaden los grados de libertad, por lo que, al construir un robot, el número de grados de libertad será la suma total de los grados de libertad de todas las articulaciones que lo componen. Se puede distinguir 6 tipos de robots según las articulaciones que se hayan implementado:

- Robot Cartesiano: Posee tres movimientos lineales (3 GDL) en los ejes X, Y y Z, y se emplea cuando el espacio de trabajo es grande.
- Robot Cilíndrico: Presenta un movimiento de rotación en la base, y dos movimientos lineales para la altura y radio, es decir, presenta 3 GDL.
- Robot Esférico o Polar: Cuenta con dos articulaciones rotacionales, cuyos movimientos son rotacional y angular, y una lineal, por lo que tiene 3GDL.
- Robot SCARA: Tiene dos articulaciones rotacionales y se le añade un movimiento lineal en su tercera articulación, presentando un robot de 3 GDL.
- Robot Angular o Antropomórfico: Presenta una articulación con movimiento rotacional y dos articulaciones con movimiento angular, y puede mover simultáneamente dos o tres articulaciones. Dependiendo del número de eslabones y articulaciones, el robot puede presentar desde 3 GDL hasta 7 GDL. Es el robot que más se asimila al brazo humano.
- Robot paralelo: Posee articulaciones prismáticas o rotacionales concurrentes. A diferencia de los anteriores, este robot tiene una cadena cinemática cerrada, donde una plataforma móvil se conecta a la base por medio de varias cadenas independientes.

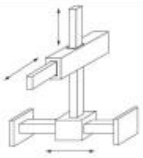
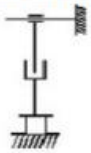
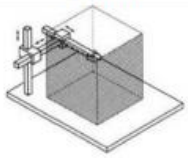
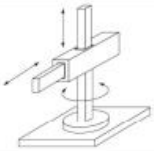
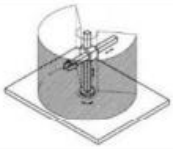
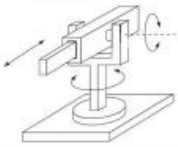
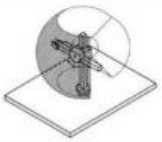
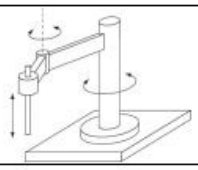
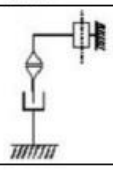
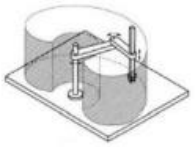
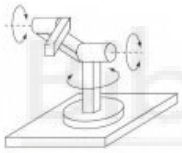
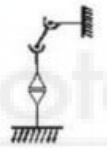
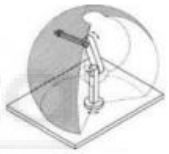
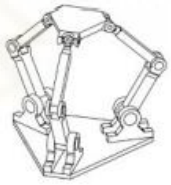
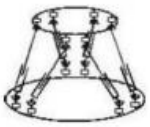
Tipo de Robot	Esquema	Estructura cinemática	Espacio de trabajo
Robot Cartesiano			
Robot Cilíndrico			
Robot Polar			
Robot SCARA			
Robot Angular o Antropomórfico			
Robot de Estructuras paralelas			

Figura 9. Los 6 tipos de robots según sus articulaciones
Fuente: Tecnología-técnica

Estos robots trabajan de manera conjunta con otras máquinas y herramientas para formar células de trabajo. Un robot industrial, en general, permite una fácil reprogramación, ahorro en costes, la reducción en tiempos de producción y la mejora del flujo de datos que presenta, entre otras cosas, y hacen que, gracias a estos factores. Los robots industriales se emplean generalmente en las siguientes aplicaciones:

- Operaciones de procesamiento.
- Operaciones de ensamblaje.
- Operaciones de empaque.
- Operaciones de soldadura.

- Otras operaciones como remachados, control de calidad, cortes, etc.

Explicada la parte importante de la estructura mecánica de un brazo robótico industrial, para este trabajo, se ha empleado un robot antropomórfico, al cual se ha añadido un elemento terminal para que pueda mover objetos y que se explica a continuación.

Los elementos terminales o efectores finales, son los que se encargan de interaccionar con el entorno directamente. Pueden ser elementos de aprehensión o herramientas, y la mayoría de las ocasiones se hacen diseños especificados para cada tipo de trabajo.

En este caso, el trabajo consiste en coger objetos, por lo tanto, solamente se tendrá en cuenta los elementos terminales de sujeción, los cuales son:

- Pinza de presión: Se acciona neumática o eléctricamente. Se utiliza para mover objetos que no importen ser presionados.
- Pinza de enganche: Su accionamiento es el mismo que la pinza de presión. Y son para transportar objetos grandes u objetos que no se deban ejercer presión.
- Ventosa de vacío: Se acciona neumáticamente. Para objetos con superficies lisas.
- Electroimán: Como su nombre indica, su accionamiento es eléctrico. Solamente para objetos ferromagnéticos.

2.3 ROBOTS COLABORATIVOS

El empleo de los robots en el mundo industrial presenta una larga trayectoria, pero actualmente se debe hablar de un nuevo concepto de robótica, conocida como la robótica colaborativa.

Un robot colaborativo o también llamado *cobot*, es un brazo robótico diseñado para trabajar junto a los humanos en una cadena de producción y en un espacio de trabajo colaborativo, donde durante su funcionamiento, tanto como el robot y la persona trabajan de forma simultánea. Cuya finalidad es la de automatizar procesos industriales para mejorar la eficacia y productividad y poder optimizarlos en todo lo posible.

Los *cobots* se diferencian de los robots por las características que presentan, además de que los hacen muy atractivos para implementarlos en la industria. La primera característica es el tipo de trabajo que pueden realizar, son robots perfectos para realizar tareas repetitivas por su gran precisión, evitando lesiones, y, añadir, que se pueden aplicar en puestos de trabajo donde las posturas sean incómodas o pongan en riesgo la salud de los trabajadores. Otra de las características, es que se pueden usar en todo tipo de industrias, gracias a su versatilidad, los *cobots* se pueden aplicar en cualquier tipo de

proceso industrial, desde el sector automovilístico hasta poder exponerlos a altas temperaturas, trabajar con materiales tóxicos o emplear herramientas cortantes, y, además, permite la interacción con otros equipos ya sean externos o periféricos. Y la característica más importante que hace diferenciar los *cobots* de los robots, es su

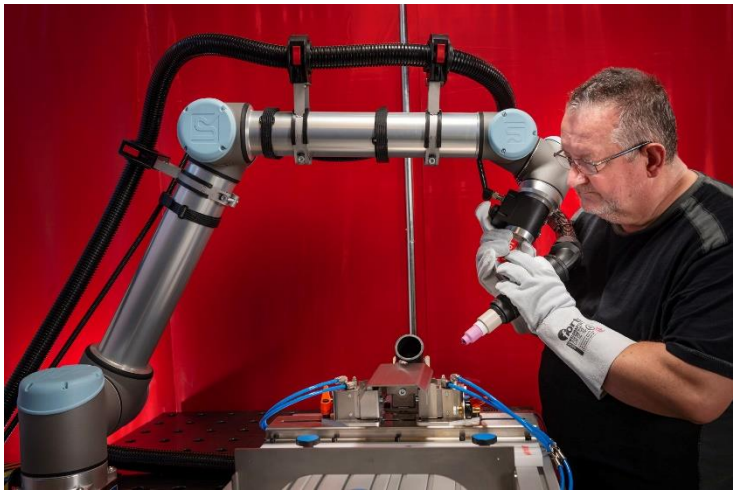


Figura 10. Ejemplo de utilización de un cobot

seguridad, permiten a los trabajadores realizar sus funciones sin tener que poner en riesgo su seguridad, debido a que tienen integrados sensores que les permite crear respuestas ante variaciones en el entorno y poder detenerse en el momento que entran en contacto para evitar posibles accidentes.

La empresa pionera en la fabricación, diseño e implementación de robots colaborativos en la industria, es la marca Universal Robots. La empresa danesa desarrollo 4 modelos de brazos robóticos, a los cuales se añaden la versión e-Series (versiones mejoradas de los modelos anteriores) y un lanzamiento de un modelo nuevo, están pensadas para simular los movimientos que realiza un brazo humano a la perfección pudiendo aplicarse en todo tipo de industria.

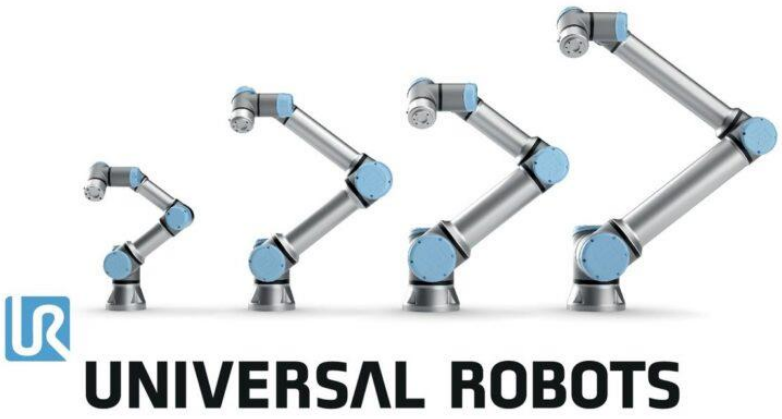


Figura 11. Modelos de robot de Universal Robots

En el capítulo siguiente se describe con detalle el robot empleado en el trabajo, el cual, se trata de uno de los modelos de Universal Robots.

CAPÍTULO 3: EQUIPO EMPLEADO

En este capítulo se expone todo el material empleado en la simulación del trabajo, se trata del brazo robótico UR5, uno de los modelos desarrollados por Universal Robots, un sensor incorporado en el extremo del brazo, el cual es una cámara y, por último, como elemento terminal, se añade en el extremo una ventosa de vacío, para la sujeción de los objetos. Se ha de añadir que los elementos son simulados, por lo tanto, en la simulación, no es un sensor ni una ventosa reales, se han creado de forma que tengan una similitud y se demuestre que se han incorporado esos elementos.

3.1 BRAZO ROBÓTICO UR5

El modelo UR5 es un robot colaborativo de la marca Universal Robots. Es el brazo robótico colaborativo más grande que ofrece Universal Robots de carga útil ligera, que no quiere decir que sea el más grande de la marca, ya que, puede ofrecer el modelo UR10, modelo con más alcance y mayor carga útil. Permite indicarle movimientos y ser programado de formas muy diversas, como se comenta en los capítulos siguientes. Se trata de un robot que consta de 6 articulaciones de rotación, es decir, es un robot con 6 GDL.

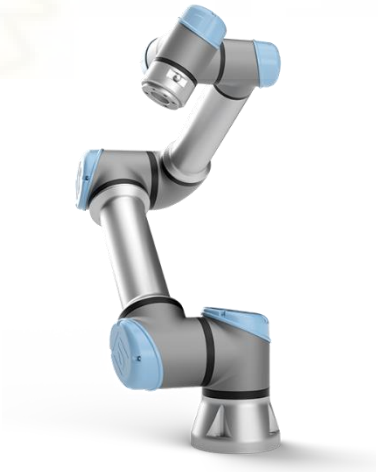


Figura 12. Modelo UR5

El UR5 es ligero, compacto, versátil y dependiendo de las circunstancias de la producción se puede adaptar en cualquier momento. Permite un radio de acción de 850 mm con una carga útil de 5 kg lo que hace que sea uno de los robots más polivalentes cuando se automatiza tareas de procesamiento de bajo peso. Se caracteriza también, por su facilidad

de programar y de manejar, y combinando estas características hace que la automatización con estos robots sea rápida, flexible y asequible. Y como se ha mencionado en capitulo anterior, la característica más representativa de un *cobot* es su seguridad, y el UR5, no es una excepción, ya que, los elementos sensoriales de su equipo hacen que el robot opere a menor velocidad cuando detecta la presencia de un trabajador y detenerse en caso de detectar alguna colisión.

El UR5 se puede emplear en una variedad de aplicaciones, gracias a todas las características mostradas previamente. Pero entre ellas, cabe destacar las siguientes aplicaciones:

- *Pick and place*: Donde el brazo robótico realiza una operación de recogida del objeto en una posición y colocándolo en otra.
- CNC: El robot realiza la supervisión y cuidado de la maquinaria.
- Ensamblaje colaborativo: Con la ayuda del trabajador, ambos realizan el ensamblaje.
- Otras aplicaciones son la dispensación, inspección de calidad, paletizado (sin superar la carga útil) y soldadura.

Para conocer las características técnicas que presenta el robot UR5, véase el Anexo 3, en él se muestra todas las características que presenta el robot y que se debe tener en cuenta si realiza el trabajo con el hardware real.

Figura 13. Ejemplo de UR5 realizando la tarea de embalaje

3.2 WEBCAM

Una de las herramientas que se utiliza para este trabajo es una cámara web simulada, que se encuentra acoplada en la muñeca 3 del UR5, y se mueve juntamente con éste.

Para la simulación, no hace falta especificar el tipo de cámara web que se utiliza y, por lo tanto, para mostrar que se incorpora, en la simulación se coloca un cubo de dimensiones 0.02 x 0.02 x 0.02 metros, de color verde en la muñeca 3 del UR5, y para añadir realismo a la simulación, se le asigna a la cámara un valor de masa de 0.005 kg. Sin embargo, para el correcto funcionamiento del trabajo, se especifica los parámetros de la imagen de la cámara web:

- Campo de visión horizontal (*horizontal_fov*): El campo de visión de la cámara se encuentra en radianes y el valor es de 1.3962634.
- Ancho (*width*): Se establece el ancho de la imagen, el cual es 800.
- Altura (*height*): Se establece la altura de la imagen, el cual es 800.
- Formato (*format*): Se define el formato de imagen, en este caso es "R8G8B8", es decir, cada píxel tendrá tres canales de color con 8 bits por canal.
- Distancia mínima (*near*): Establece la distancia mínima que la cámara puede ver, el cual es 0.02 metros.
- Distancia máxima (*far*): Establece la distancia máxima que la cámara puede ver, el cual es 300 metros.

Para que en la simulación pueda detectar perfectamente los objetos, se le aplica un ruido, del tipo Gaussiano, con una media de 0 y una desviación estándar de 0.007.

Por último, para poder tener una cámara corregida, se le ha aplicado una calibración para poder detectar las distorsiones que presenta, dando como resultado los siguientes valores:

- DistorsiónK1: 0.000099
- DistorsiónK2: -0.000342
- DistorsiónK3: -0.000102
- DistorsiónT1: 0.000011
- DistorsiónT2: 0.000000

Además de estos parámetros, se ha tenido en cuenta los valores de inercia:

- Ixx: 1e-6 kg*mm2
- Ixy: 0 kg*mm2
- Ixz: 0 kg*mm2
- Iyy: 1e-6 kg*mm2
- Iyz: 0 kg*mm2
- Izz: 1e-6 kg*mm2

3.3 ELEMENTO TERMINAL

Como elemento terminal para poder coger objetos, se incorpora en el centro de la muñeca 3 del UR5 una ventosa al vacío simulada, porque los objetos que se emplean son cubos lisos.

Como la cámara web, la ventosa al vacío no hace falta que se especifique que tipo se va a utilizar, y para mostrar que se incorpora, en la simulación se coloca un cilindro blanco de radio 0.005 metros y altura 0.01 metros, en la muñeca 3 del UR5, asemejándose a una ventosa, y se le da el valor de 0.001 kg de masa para darle realismo a la simulación.

Pero para que el trabajo funcione correctamente, se especifica los parámetros de fuerza que debe aplicar la ventosa para que pueda coger el objeto, además de los rangos para poder agarrar el objeto:

- Fuerza máxima (*maxForce*): Define la fuerza máxima que puede aplicar en la simulación, en este caso es de 20 unidades.
- Máxima distancia (*maxDistance*): Indica la distancia máxima que puede operar la ventosa, cuyo valor es de 0.05 unidades.
- Mínima distancia (*minDistance*): Indica la distancia mínima que puede operar la ventosa, cuyo valor es de 0.01 unidades

CAPÍTULO 4: ESTADO DEL ARTE

En el presente capítulo se presenta el núcleo del trabajo, que es el *Robot Operating System* (ROS). En él se trata tanto de su funcionamiento como la justificación de su utilización. Por último, se comenta las distintas herramientas más importantes que ofrece ROS.

4.1 MIDDLEWARES

Antes de hablar sobre ROS, se debe hacer hincapié en qué consiste un *middleware* y donde se encuentra su funcionamiento.

Se conoce como *middleware* al software que permite uno o más tipos de comunicación o conectividad entre otros componentes de softwares o aplicaciones. Recibe su nombre debido a que actúa como mediador entre una aplicación frontal, o cliente, y un recurso de fondo. En general, consiste en un conjunto de servicios que permiten que se ejecuten varios procesos en varias máquinas para interactuar.

Este software se desarrolló para fomentar la difusión de las arquitecturas distribuidas coherentes, para apoyar y simplificar el desarrollo de complejas aplicaciones distribuidas, a modo de interoperabilidad entre arquitecturas, sin tener que crear una integración personalizada cada vez que se conecte a los servicios de la aplicación, fuentes de datos o dispositivos.

Los servicios que habilitan esa comunicación entre aplicaciones y servicios se realizan mediante marcos de mensajería como JSON (*JavaScript Object Notation*), REST (*Representational State Transfer*), XML (*Extensible Markup Language*), SOAP (*Simple Object Access Protocol*) o servicios web. Además, proporciona que diferentes leguajes se comuniquen entre sí, como Java, C++, PHP y Python.

El *middleware* se establece entre las aplicaciones de software que se pueden ejecutar en diferentes sistemas operativos, es decir, “en medio”. Se asimila a la capa intermedia entre el código de la aplicación y de la infraestructura de tiempo de ejecución. Consiste en una biblioteca de funciones, y permite a las aplicaciones usar esas funciones para no tener que volver a crear esas funciones de nuevo parar cada aplicación.

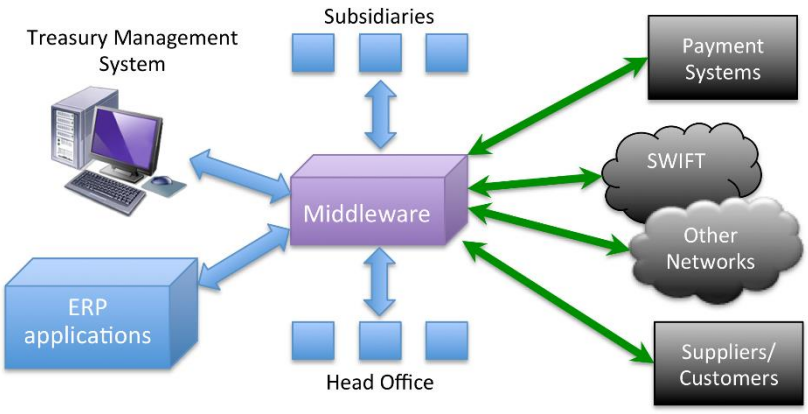


Figura 14. Arquitectura del funcionamiento del middleware

Si se centra el concepto de *middleware* en la robótica, cada robot, al ejecutarse un software, leen los datos de los sensores, extraen la información que necesitan, calculan la secuencia de acciones para controlar los actuadores y llevan a cabo una tarea determinada, es decir, hay una sola aplicación que se encarga de todas esas tareas y provoca que el mantenimiento del código sea duro y el poder reutilizar el código e intercambiar entre proyectos distintos.

En esa situación, el *middleware* ayuda a los componentes de hardware y software, que necesitan comunicarse y colaborar para llegar a una meta, mejorando la organización, mantenimiento y eficiencia de código. Se estructura el código en muchas tareas pequeñas concretas donde los componentes intercambian datos por un canal de comunicación común creado por el *middleware* y haciendo uso de interfaces compatibles entre las aplicaciones. Gracias al *middleware*, el código se puede reutilizar y compartir de manera más fácil a otros proyectos, ya que sólo se necesita mantener la misma interfaz.

4.2 DISTINTOS MIDDLEWARES

A parte de ROS, existen otros *middlewares* conocidos en la comunidad robótica que todavía son apoyados y mantenidos de forma activa. A continuación, se menciona los *middlewares* más importantes que existen actualmente, realizando una breve descripción.

4.2.1 OROCOS

En el año 2000, surge la idea de hacer un proyecto de software libre para el control de los robots, debido a las malas experiencias y fracasos al emplear softwares comerciales para investigaciones avanzadas en el campo. En ese momento surge OROCOS (*Open RObot*

Control Software) como proyecto innovador interesado por mucha gente en ese año, gracias a la participación de K. U. Leuven en Bélgica, LAAS Toulouse en Francia y KTH de Estocolmo en Suecia.

Figura 15. Logotipo de OROCOS

OROCOS proporciona las cuatro funciones principales de un middleware: abstracción del sistema operativo y soporte de tiempo real, servicios middleware, infraestructuras de comunicación y modelo basado en componentes. Está en continuo desarrollo, consolidándose como una de las mejores opciones para el control de robots y que actualmente se encuentra en diversos proyectos.

En la actualidad el middleware puede soportar tres librerías:

- *Kinematics and Dynamics* (KDL): Librería que permite el modelado y la especificación del movimiento a un problema geométrico con cálculos matemáticos. Además, proporciona librerías de clases de objetos geométricos, cadenas cinemáticas y especificación de movimientos e interpolación.
- *Bayesian Filtering* (BFL): Proporciona un marco de aplicación independiente para el procesamiento de información y algoritmos de cálculo basado en la regla de Bayes. Los algoritmos se pueden ejecutar como en tiempo real o ser utilizados para la estimación de las aplicaciones de Cinemática y Dinámica.
- *Orocos Toolchain*: Es la librería más reciente de OROCOS, donde vienen incluidas otras herramientas como *AutoProj* (descargar y compilar automáticamente), *Real Time Toolkit* (RTT) que permite crear componentes configurables, *Orocos Component Library* (crear todo tipo de componentes), y *OroGen* y *TypeGen* (generar código OROCOS).

4.2.2 Orca

Este *middleware* surgió a partir de OROCOS en KTH de Estocolmo en el año 2003. Consiste en un marco de trabajo de código abierto basado en componentes para el desarrollo de sistemas robóticos. Proporciona los medios para la construcción de sistemas robóticos complejos, desde dispositivos individuales hasta redes de sensores distribuidos.

Sus desarrolladores se centran en la reutilización del software mediante interfaces en común, simplificarla proporcionando librerías con una API y animar esa reutilización con un gran mantenimiento porque es un factor clave para el avance en la investigación robótica e industria.

Figura 16. Logotipo de Orca

Orca adopta un enfoque de Ingeniería de Software Basado en Componentes sin restricciones de arquitectónica adicional y emplear una librería de código abierto comercial.

La principal diferencia entre Orca y OROCOS es la herramienta que emplean en la construcción de la comunicación, trabajando con *Internet Communication Engine* (ICE). Gracias a ICE, Orca se compone por servicios básicos, de los cuales, los principales son:

- Registro *IceGrid*: Que ofrece un servicio de nombres, un mapeo de nombres de interfaces lógicas.
- Servicio *IceStorm*: Es un servicio de eventos, que se usa para desacoplar los publicadores de mensajes de los suscriptores.

4.2.3 YARP

Por último, el *middleware* que se trata en este apartado es YARP (*Yet Another Robotic Platform*), y al igual que los *middlewares* explicados previamente es un software gratuito y de código libre. YARP se compone por un conjunto de bibliotecas, protocolos y herramientas para mantener los módulos y dispositivos desacoplados, y lo más importante, no se trata de un sistema operativo.

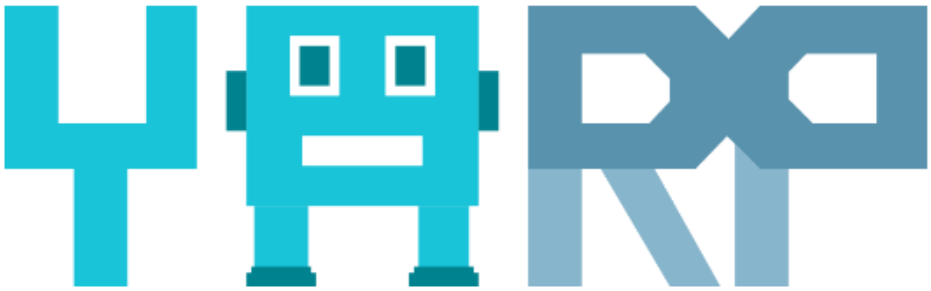


Figura 17. Logotipo de YARP

El modelo YARP utiliza una metodología para la interconexión con dispositivos que alienta la articulación flexible y puede hacer cambios menos perjudiciales en los dispositivos, además de minimizar el problema de arquitecturas incompatibles y marcos de trabajo.

Se puede resumir que YARP es un conjunto de herramientas que satisfacen las necesidades computacionales para el control de varios robots humanoides. Esas herramientas se pueden dividir en:

- *libYARP_OS*: Es la interfaz con el sistema operativo y permite el flujo de datos de diferentes hilos en diferentes máquinas.
- *libYARP_sig*: Realiza las tareas comunes de proceso de señal interconectada entre otras librerías usadas en común.
- *libYARP_dev*: Es la interfaz con dispositivos comunes usados en robótica (cámaras digitales, tarjetas de control, etc.).

4.3 ROS (ROBOTIC OPERATING SYSTEM)

Después de explicar lo que es un *middleware* y mencionar otros middlewares que se emplean en la industria, en este apartado se centra en el middleware robótico que está llamado a ser el estándar de la industria durante bastantes años.

Esta parte del proyecto se considera como trabajo de investigación, ya que, todos los conocimientos que han sido adquiridos para llevar a cabo el trabajo se han conseguido a

través del autoaprendizaje y la experiencia propia. Esto implica que se ha invertido una buena cantidad de tiempo en el aprendizaje de la herramienta ROS para el control de robots.

Para entender cómo se organiza ROS, su sistema de comunicación interna, gestión de paquetes, etc., no es algo que se domine en pocos meses o con la lectura de unos cuantos tutoriales, y puede ser caótico si no hay otra persona que lo domine y pueda resolver dudas mediante explicaciones. Para ser un experto, requiere varios años de experiencia y mucha constancia, debido a su gran velocidad de actualización.

4.4 DEFINICIÓN

ROS proviene de las siglas en inglés *Robot Operating System*. Es un meta-sistema operativo de código abierto, considerado ser más que un *middleware* porque proporciona los servicios de un sistema operativo, el cual incluye la abstracción de hardware, control de dispositivos de bajo nivel, implementación de la funcionalidad de uso común, paso de mensajes entre procesos y gestión de paquetes. Además, consta de una colección de herramientas, bibliotecas y convenios para obtener, construir, escribir y ejecutar código en varios ordenadores y poder simplificar la tarea de desarrollar un software para robots. El meta-sistema operativo describe un sistema que realiza procesos como la programación, carga, supervisión y gestión de errores utilizando capa de virtualización entre las aplicaciones y recursos informáticos distribuidos. ROS, al ser un meta-sistema operativo, no es un sistema operativo en sí, ya que, debe estar instalado sobre otro, preferentemente instalar sobre sistemas Linux, como Ubuntu o Debian.

Para mayor detalle de la instalación de ROS para este trabajo consulte el ANEXO 1.

Figura 18. Logotipo de ROS

Como objetivo principal que ROS busca es permitir la reutilización de código en la investigación y desarrollo de la robótica y poder así liberar paquetes de softwares listos para utilizarse en un gran conjunto de tareas comunes. Pero se debe añadir, que ROS se caracteriza, también, por la filosofía que propone y se puede resumir en:

- Gratuito y de código abierto: Es un software libre que está a disposición del público. ROS está distribuido bajo los términos de la licencia estándar BSD (*Berkeley Software Distribution*), permitiendo el desarrollo de proyectos tanto comerciales como no comerciales.
- Ligero: El desarrollo de controladores y algoritmos se realiza mediante bibliotecas independientes que no dependen de ROS. Para colocar toda la complejidad en las bibliotecas y crear pequeños ejecutables que expongan la funcionalidad de la biblioteca a ROS, permitiendo su reutilización en otros proyectos.
- Basado en herramientas: Se utiliza un gran número de pequeñas herramientas para construir y ejecutar los distintos componentes de ROS, para descartar un entorno monolítico de desarrollo y ejecución.
- Multilenguaje: ROS está diseñado para que sea neutral en cuanto al lenguaje. Actualmente, puede soportar el lenguaje C++, Python, Octave y LISP, y otros lenguajes en distintos estados de desarrollo.
- Gran cantidad de componentes: El software está rodeado de todo tipo de componentes que le permiten ampliar sus aplicaciones como comunicación o simulación. Dichos componentes se observan en la Figura 19.

Figura 19. Tabla de componentes que contiene ROS.
Fuente: ROS Robot Programming

- Proceso distribuido: Su programación se realiza en forma de unidades mínimas de procesos, llamados nodos, y cada nodo se ejecuta de forma independiente e intercambia datos sistemáticamente.

Además, ROS ofrece una comunidad web para poder compartir y colaborar, y repositorios para distribuir paquetes de software.

4.5 JUSTIFICACIÓN DEL EMPLEO ROS

Como se ha explicado con anterioridad ROS es un software de código abierto, que implica cero gastos, y cuenta con una amplia cantidad de desarrolladores que contribuyen a mejorar el sistema, lo que permite que esté actualizado en todo momento, lo que hace que sea un buen software para realizar un trabajo de esta envergadura. Gracias a su gama variada de herramientas, permite observar el comportamiento del robot en un laboratorio o en la industria, permitiendo que se pruebe en una simulación para prever fallos antes de llevarlo a la realidad.

Otra de las razones que motivó a continuar el uso de la herramienta ROS, es investigar un nuevo software, que no se ha tratado en el curso y que se emplea en la industria.

Por último, otro aspecto para tener en cuenta es que ROS permite diferentes lenguajes de programación, siendo en este caso C++ y Python, los cuales se ha utilizado los dos en el presente trabajo.

4.6 ESTRUCTURA Y FUNCIONAMIENTO

ROS presenta una estructura que consta de tres niveles de conceptos donde en cada uno se describe el funcionamiento del software:

- *ROS Filesystem Level*: En este nivel se encuentra la estructura que deben tener las carpetas y los archivos mínimos necesarios para que funcione el software.
- *ROS Computation Graph Level*: Se crea una red con el objetivo de que todos los procesos se conecten, y los nodos envíen o reciban datos de esta red.
- *ROS Community Level*: Es el conjunto de herramientas para compartir paquetes, documentación y dudas entre los desarrolladores.

En cada nivel se divide en una serie de conceptos que deben ser explicados para el correcto entendimiento del funcionamiento de ROS, pero los más importantes son los dos primeros:

-En el nivel *ROS Filesystem Level* se encuentran los siguientes conceptos:

- Paquetes: Son la unidad principal para organizar el software en ROS. Contienen nodos, bibliotecas, conjuntos de datos y archivos de configuración. También son los elementos de compilación y lanzamiento más simples.
- Metapaquetes: Sirven para representar un grupo de paquetes relacionados.
- Manifiesto de paquete: Son archivos *.xml* con metadatos sobre un paquete.
- Repositorios: Es una colección de paquetes que compartes un sistema VCS (*Version Control System*) común.
- Tipos de mensajes (*msg*): Carpeta con archivos *.msg* que definen las estructuras de datos para los mensajes enviados en ROS.
- Tipos de servicio (*srv*): Carpeta con archivos *.srv* que definen las estructuras de datos de solicitud y respuesta para los servicios en ROS.

-Dentro del nivel *ROS Computation Graph Level* están los siguientes conceptos:

- Nodos: Son las unidades computacionales de ROS, donde se produce toda la computación además de encargarse de gestionar la información entrante y saliente. Cada nodo tiene asociado un script en C++ o Python para dictar su configuración y su comportamiento.
- Master: Proporciona el registro de nombres y busca al resto del gráfico de cálculo. Sin el master, los nodos no se podrían encontrar, intercambiar mensajes o invocar servicios.
- Servidor de parámetros: Forma parte del master y permite que se almacene los datos por clave en una ubicación central.
- Mensajes: Para que los nodos se comuniquen entre sí, se envían mensajes. Son una estructura de datos, que comprenden campos escritos y admiten tipos primitivos estándar (enteros, coma flotante, booleanos, etc.) o matrices tipo primitivos.
- *Topics*: En un nombre que se emplea para identificar el contenido del mensaje, porque los mensajes se enrutan por un sistema de transporte con semántica de publicación/suscripción. De esta forma, un nodo envía un mensaje publicándolo en un *topic* determinado y lo mismo ocurre, cuando un nodo recibe mensajes suscribiéndose al *topic* correspondiente. Si el nodo publica mensajes se considera *Publisher* y si recibe se considera *Subscriber*.

- Servicios: En muchas ocasiones, cuando se publica en un *topic*, su transporte unidireccional de muchas a muchas no es apropiado para las interacciones de solicitud y respuesta. Para ello están los servicios, que se define como un par de estructuras de mensajes: una para la solicitud y otro para la respuesta.
- *Bags*: Son un formato para guardar y reproducir datos de mensajes ROS. Se encargan de almacenar datos difíciles de recopilar, pero son necesarios para desarrollar y probar algoritmos.

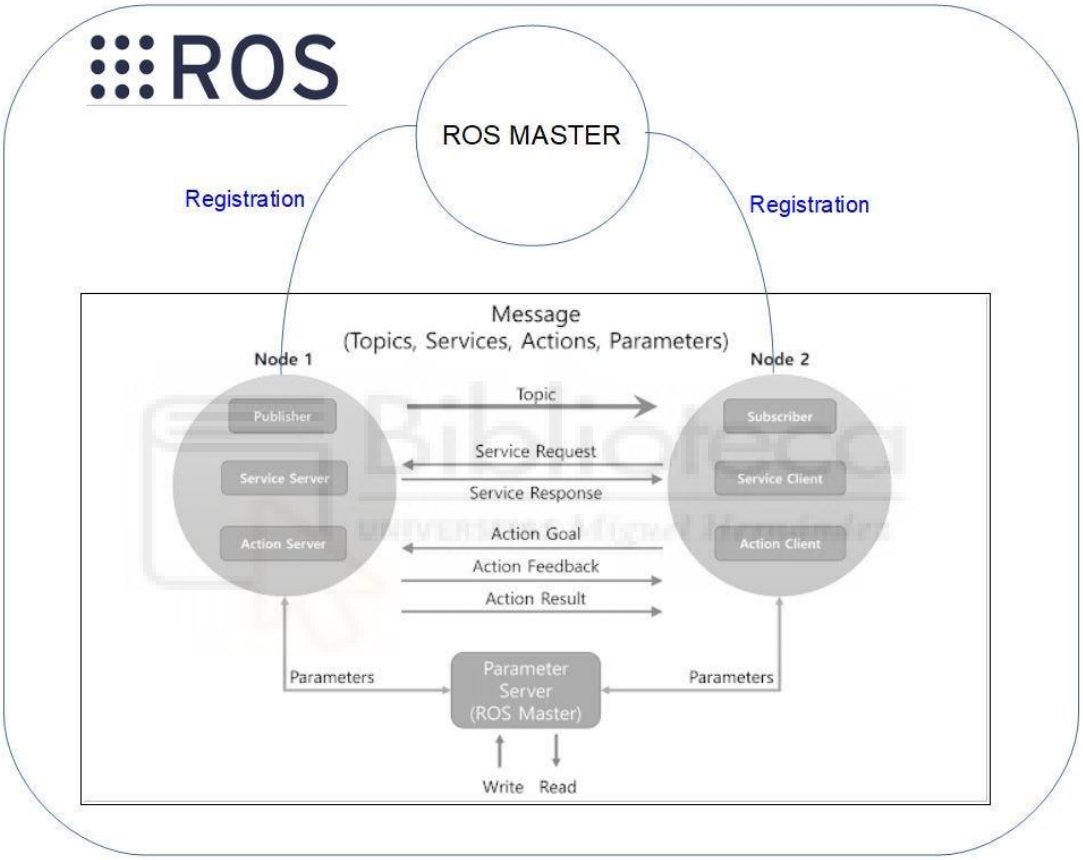


Figura 20. Funcionamiento del ROS Computation Graph Level
Fuente: ROS Robot Programming

4.7 HERRAMIENTAS DE ROS

ROS ofrece una serie de herramientas dedicadas a la simulación de entornos y robots, para poder visualizar como seria el robot para detectar posibles errores y comprobar el comportamiento del robot cuando se ejecutan los programas deseados en los distintos entornos antes de llevarlo a la realidad.

4.7.1 Gazebo

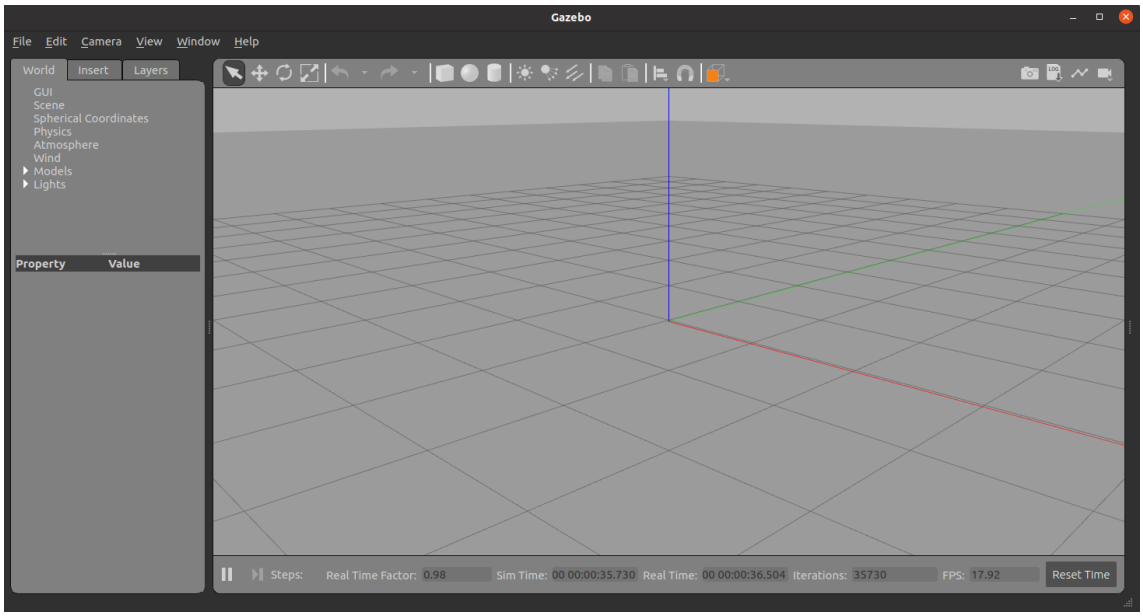
Gazebo es una potente herramienta de simulación 3D, que permite probar de manera inmediata robots diseñados, algoritmos en robots y realizar pruebas de regresión utilizando escenarios realistas. También puede controlar los entornos donde los robots van a funcionar, para comprobar su funcionamiento en un espacio controlado antes de llevarlo a la realidad.

Gazebo es una colección de bibliotecas de software de código abierto diseñadas para simplificar el desarrollo de aplicaciones de alto rendimiento, que ha cogido popularidad en estos años, también debido a su compatibilidad con ROS. Cada biblioteca tiene dependencias mínimas que les permite resolver transformaciones matemáticas, codificación de vídeo y gestión de procesos entre otras tareas.

Figura 21. Logotipo de Gazebo

Para mostrar el entorno de *Gazebo*, se puede lanzar directamente o mediante un archivo *.launch* donde se especifique qué se desee mostrar.

Figura 22. Mundo vacío e interfaz de Gazebo



A continuación, se describen algunas de las características más importantes del simulador 3D:

- Simulación realista: Una gran variedad de sensores y modeles de ruidos están disponibles, como cámaras monoculares, LIDAR, IMU y más en camino, a los que se le puede añadir ruido gaussiano o personalizado. Presenta gráficos 3D avanzados gracias a Ogre (Motores de renderizado de gráficos de código abierto) con una física precisa por los motores físicos que emplean, como Bullet, Simbody y DART para conseguir una simulación más dinámica.
- Rendimiento: La mejora de rendimiento es debido a que Gazebo admite el uso de múltiples servidores, además de una carga dinámica de activos permitiendo la descarga y carga automática de activos.
- Plataformas e integraciones: Las bibliotecas de Gazebo se pueden usar tanto en Linux como en MacOS y Windows. En la nube de Gazebo se puede descargar y cargar distintos modelos y modelos ya creados.

4.7.2 RViz

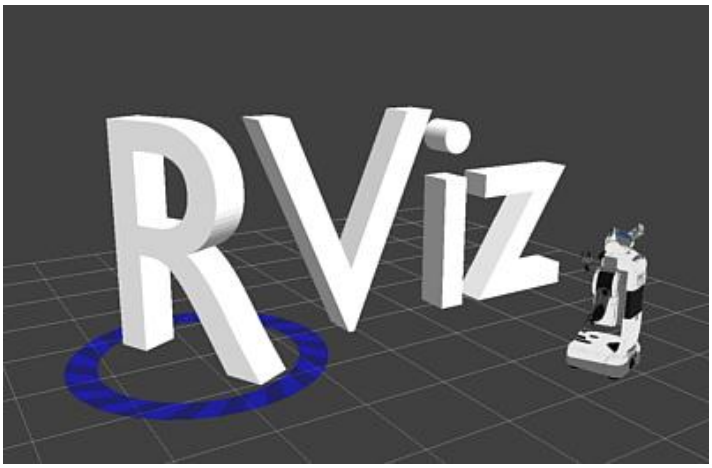


Figura 23. Logotipo de RViz

RViz es la abreviación de *ROS Visualization* y es otra potente herramienta útil y completa de visualización 3D que ofrece ROS, ya que, permite mostrar de forma intuitiva y en tiempo real, el estado y la información de un sistema en ROS. En *RViz* se puede integrar todo tipo de elementos (visión, control, navegación…) y controlarlos de forma interactiva. Se debe añadir, que ofrece una multitud de opciones parar variar el entorno que se visualiza, como la adición o extracción de elementos, marcadores, trayectorias, etc. Y poder añadir sensores o ejecutar movimientos.

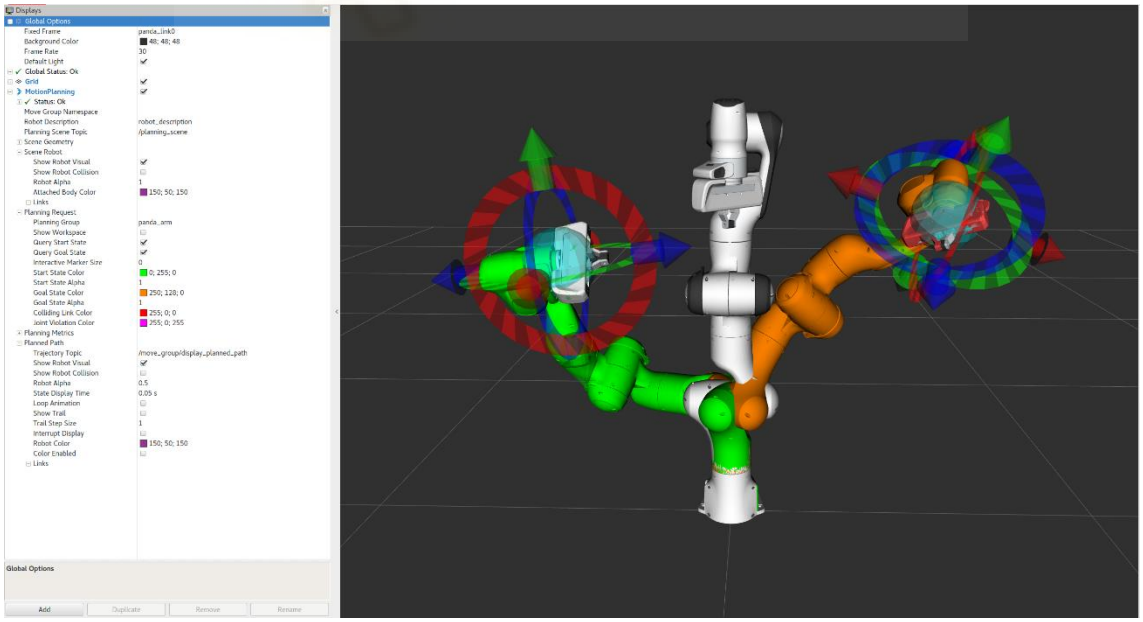


Figura 24. Interfaz de RViz junto ejemplo del robot Panda

La característica más importante que tiene *RViz*, aparte de poder visualizar cualquier tipo de robot en simulación, es poderse conectar a un robot real y reproducirlo en el ordenador en tiempo real a sus movimientos y sensores.

4.7.3 URDF

El archivo URDF (*Unified Robot Description Format*) es un formato de lenguaje que se utiliza en la descripción de robots y donde se especifica sus propiedades (dimensiones, articulaciones, parámetros físicos) en el marco de gramática XML. Mediante estos archivos podemos generar el modelo de un robot y simularlo tanto en *Gazebo* como en *RViz*, las dos herramientas de simulación descritas previamente.

```
<robot>
        <link>
                ...
        </link>
        <link>
                ...
        </link>
        <joint>
                ...
        </joint>
</robot>
```

Figura 25. Ejemplo de la estructura del archivo URDF

En el archivo XML se distingue dos componentes principales que son necesarios para el diseño de la cadena cinemática de un robot, los cuales son:

- *Links* o eslabones: Son las partes físicas del robot, y como se ha comentado en el capítulo 2, consta de la parte rígida del robot, como masa, inercia o geometría. Además, es la parte visual del robot en las simulaciones de *Gazebo* y *RViz*
- *Joints* o articulaciones: Son las uniones entre eslabones o "links" del robot. Describen el tipo de movimiento relativo entre los eslabones conectados y especificar los límites de colisión. Entre los distintos tipos de movimiento, en URDF se puede aplicar:
    - *Revolute*: Giro sobre un eje con límites superior e inferior.
    - *Continouos*: Giro alrededor de un eje sin límites superior ni inferior.
    - *Prismatic*: Deslizamiento a lo largo de un eje con rango limitado.
    - *Fixed*: No permite movimiento, pero es útil para asociar al enlace un sistema de referencia.
    - *Floating*: Movimiento en los 6 GDL.

- *Planar*: Movimiento en un plano perpendicular al eje.

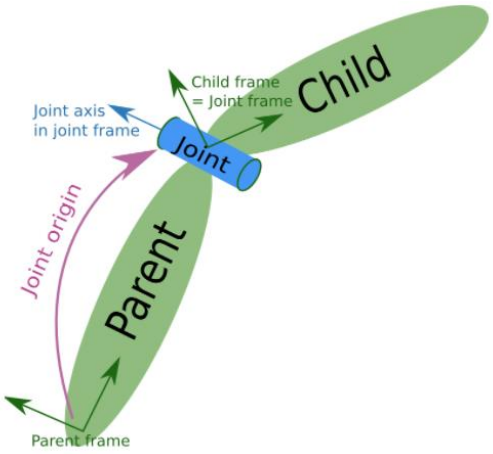


Figura 27. Descripción de la estructura de links y joints

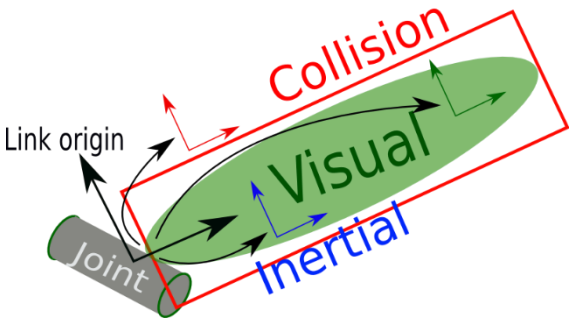


Figura 26. Descripción de la colisión

Para más información sobre la sintaxis de los archivos URDF, se puede acceder al siguiente enlace:

http://wiki.ros.org/roslaunch/XML

4.7.4 MoveIt!

Figura 28. Logotipo de MoveIt!

MoveIt! es uno de los diversos paquetes que se encuentran disponibles en ROS. Se trata de una librería de código abierto para manipuladores que proporciona una variedad de funciones, que entre ellas destacan el análisis rápido de cinemática inversa para planificación de movimientos, control manual del robot, algoritmos para la manipulación, dinámica y controladores. Y provee al usuario de una plataforma fácil de utilizar para desarrollar nuevas aplicaciones robóticas.

Al estar basado en ROS, ofrece las mismas ventajas que éste, haciendo uso del sistema de comunicación entre distintos nodos para poder funcionar. Como se observa en Figura 29, se muestra la arquitectura que presenta *MoveIt!*.

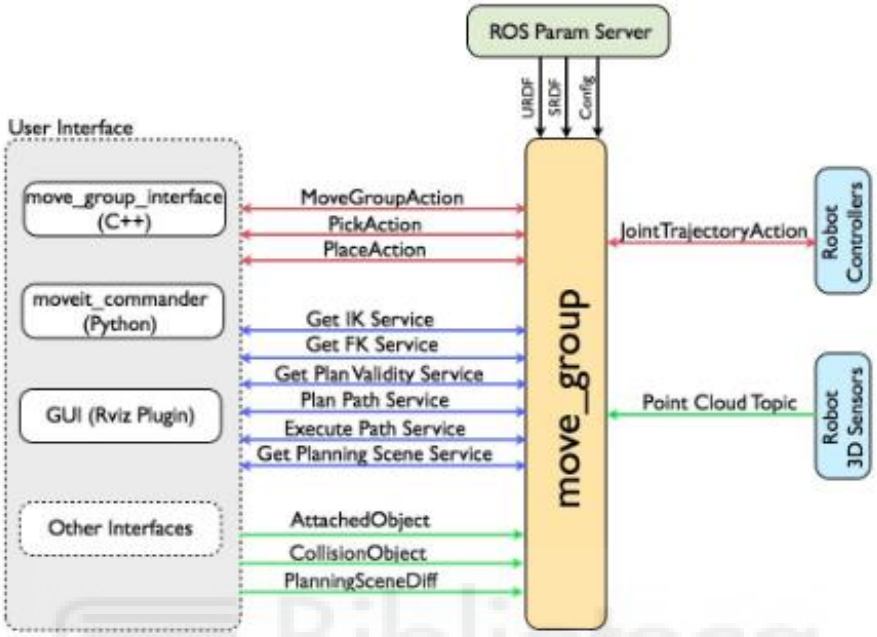


Figura 29. Arquitectura de MoveIt!

El nodo principal, llamado *move_group*, sirve como nodo integrador, enlazando todos los componentes individuales en un cuerpo que puede ofrecer un conjunto de acciones ROS y servicios para el usuario.

CAPÍTULO 5: HERRAMIENTAS MATEMÁTICAS Y VISIÓN ARTIFICIAL

En el presente capítulo se detallan las distintas herramientas matemáticas que se han utilizado en el trabajo para el cálculo del posicionamiento y la orientación del brazo robótico UR5, la calibración de la cámara y las distintas funciones que contiene la librería *OpenCV* para el procesamiento de imágenes y detectar el objeto correspondiente.

5.1 CINEMÁTICA

Se conoce como cinemática de un robot al estudio del movimiento del extremo y resto de eslabones respecto de un sistema de referencia, siendo esta referencia la base del robot. No se tiene en cuenta las causas que originan ese movimiento, es decir, las fuerzas. Se limita al estudio de la posición del extremo del brazo, su orientación y al estudio de las velocidades y aceleraciones porque describen como cambian la posición en función del tiempo.

En robótica se estudia tanto el problema de cinemática directa como el problema de cinemática inversa:

- Cinemática Directa: Es la determinación de la posición y orientación del extremo respecto del sistema de referencia de la base, conociendo los parámetros geométricos del brazo y las coordenadas articulares.
- Cinemática Inversa: Es lo contrario a la cinemática directa, es la determinación de las coordenadas articulares del brazo que permiten llevar el extremo a una posición y orientación conocidas.

El paquete de *MoveIt!* emplea una infraestructura de *plugins*, especialmente dirigida a los usuarios que quieran escribir sus propios algoritmos de cinemática inversa, que es el problema que interesa resolver para que el robot se mueva correctamente a la posición correspondiente.

En el trabajo, el *plugin* que se utiliza para el cálculo de la cinemática inversa es solucionador numérico basado en el jacobiano, de la biblioteca KDL (*Kinematics and Dynamics Library*) y es el solucionador predeterminado de *MoveIt!*.

Para el problema de la cinemática directa del robot y la búsqueda de jacobianos, la biblioteca KDL también lo calcula gracias a las clases *RobotModel* y *RobotState*, que se encarga de contener las relaciones entre todos los vínculos y articulaciones, y contener la información sobre el robot en una instantánea en el tiempo, respectivamente.

5.2 CALIBRACIÓN DE LA CÁMARA

Como se ha mencionado en capítulos anteriores, en el presente trabajo se utiliza una webcam para la detección de objetos. Si se busca la correcta visualización y detección de objetos para el posterior cálculo de su orientación, se debe realizar la calibración de la webcam y conocer sus parámetros intrínsecos.

Para la calibración, se modela un mundo donde aparezca el robot UR5 y un tablero de ajedrez que se ha creado de dimensiones 8x6 y se ejecute *MoveIt!* para poder mover el robot.

Para activar *MoveIt!* y poder mover el robot se debe ejecutar dos comandos:

```
roslaunch ur5_moveit_config ur5_moveit_planning_execution.launch sim:=true
roslaunch ur5_moveit_config moveit_rviz.launch config:=true
```

A continuación, cuando se haya cargado el mundo en Gazebo y el robot en *RViz*, ROS tiene una herramienta que se encarga de realizar la calibración de las cámaras automáticamente dentro del paquete *camera_calibration*, por lo tanto, en otra terminal se ejecuta el siguiente comando:

```
$ rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.025 image:=/ur5/usbcam/image_raw camera:=/ur5/usbcam
```

Este comando ejecuta dentro del paquete *camera_calibration* el programa cameracalibrator.py con los parámetros del tamaño del tablero de ajedrez, el tamaño de los cuadrados y los nombres de los tópicos y cámara que se utiliza en la simulación. Y aparece la siguiente ventana, Figura 30.

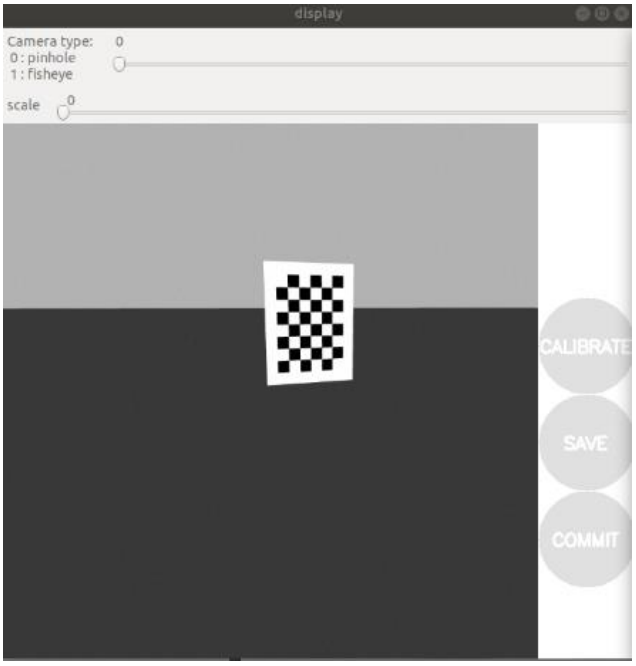


Figura 30. Ventana para realizar la calibración con el paquete camera_calibration

Para obtener los parámetros de la calibración, se debe mover el robot en distintas posiciones y ángulos, pero siempre detectando el tablero para hacer capturas de la imagen, cuando el programa detecte que tiene suficientes, en la ventana anterior aparece el botón *Calibrate* de otro color y se puede pulsar. Al pulsar, en la terminal donde se ha ejecutado el programa aparece todos los parámetros de calibración de la cámara, que son la Matriz de Cámara (K), Coeficientes de Distorsión (D), Matriz de Rotación (R) y Matriz de Proyección (P).

Ahora se muestra el programa que se ha creado para hacer las capturas de pantalla, *capture_image.py*:

```
#!/usr/bin/python3.8

import rospy
import cv2
from sensor_msgs.msg import Image
from cv_bridge import CvBridge


def save_image(image, ID):
    date_file = rospy.Time.now()
    filename = f"./Image_calibration/image{ID}.{date_file}.jpg"
    cv2.imwrite(filename, image)
    # cv2.imwrite("/TFG_ws/Image_calibration/image_{}.png".format(rospy.Time.now()), image)
    print("Image saved!")
```

```
def run():
    while not rospy.is_shutdown():
        pass


class ImageCapturer:
    def __init__(self):
        rospy.init_node("image_capture", anonymous=False)
        self.image_topic = "/ur5/usbcam/image_raw"  # Adjust this topic according to your camera topic
        self.bridge = CvBridge()
        rospy.Subscriber(self.image_topic, Image, self.image_callback)
        self.key_pressed = None

    def image_callback(self, msg, ID_FILE= 1):
        try:
            # Convert ROS Image to OpenCV format
            cv_image = self.bridge.imgmsg_to_cv2(msg, "bgr8")

            if self.key_pressed == ord('c'):
                save_image(cv_image, ID_FILE)
                ID_FILE += 1

            cv2.imshow("window", cv_image)
            self.key_pressed = cv2.waitKey(0)
        except Exception as e:
            print(e)


if __name__ == "__main__":
    capturer = ImageCapturer()
    run()
```

Cuando se obtiene los parámetros, se pueden insertar las matrices en el código que trate de la cámara y realizar la calibración, al igual que se puede añadir los valores de distorsión en el plugin de la cámara.

5.3 OPENCV

Como se ha comentado en la introducción de este apartado, mediante la webcam, el programa debe detectar los objetos que aparezcan en la simulación y, para ello, se ha implementado que detecte también colores y calcule el centro del objeto y su orientación.

La librería empleada en el trabajo para esa detección es *OpenCV* (*Open Source Computer Vision Library*) que se trata de una librería de código fuente abierto de visión por computador y software de aprendizaje automático.

Figura 31. Logotipo de OpenCV

Se ha elegido *OpenCV* por el conocimiento previo en la asignatura de Visión por Computador, además de contar con más de 2500 algoritmos optimizados entre los cuales para el trabajo interesan la identificación de objetos, detección de colores y seguir objetos en movimiento.

Cuenta con interfaces de C++, C, Python, Java y MATLAB y compatible tanto con Windows y Linux como en Android y MacOS.

También, hay que destacar que existe un puente de comunicaciones entre *OpenCV* y ROS, que permite la interacción entre las librerías de *OpenCV* y el software empleado, llamado *CvBridge*.

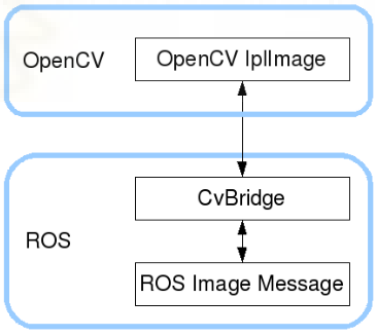


Figura 32. Estructura de funcionamiento de CvBridge

En la aplicación, *OpenCV* se utiliza, como se ha mencionado previamente para el tratamiento de vídeos a color, donde se debe detectar un color en específico y a partir de esta detección, calcular su centroide y si orientación respecto la cámara.

Para la detección del color, que debe ser una interacción constante, se opta por utilizar la representación de clores por *HSV* (*Hue, Saturation* y *Value*) que emplea parámetros llamados Matriz de Saturación y Valor. Se muestra un ejemplo:

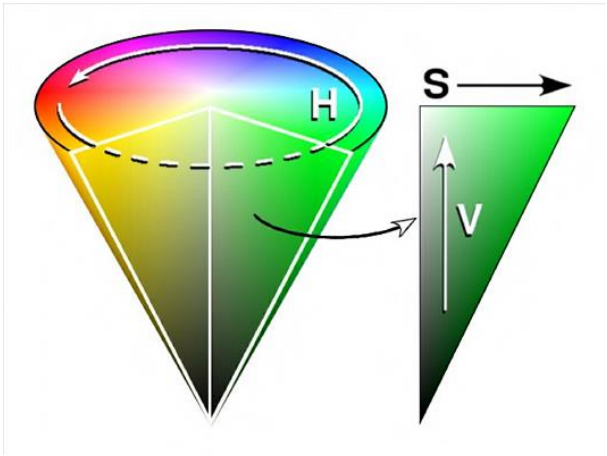


Figura 33. Representación del Color HSV

En el modelo *HSV* cada dimensión representa uno de los atributos: el matiz representa en el rango de una circunferencia los distintos colores, por ejemplo, en el grado 0 se encuentra el rojo, en el 120 el verde, y entre medias los tonos que los separan. La saturación juega con la cantidad el color que se tiene dentro de ese valor de grado en el que se encuentra.

Para el cálculo del centroide, se utiliza los momentos espaciales, características geométricas de la imagen y se tratan de parámetros ponderados calculados por la intensidad de los píxeles. Cuando se haya detectado el objeto, se puede utilizar la función de *OpenCV* que permite calcular el centro, llamado *Moments*, del cual se extraen los valores *m00, m10* y *m01*. Para obtener el valor 'x' del centro se divide *m10/m00* y para el valor 'y', se divide *m01/m00*. Así se obtiene los valores del centro del objeto detectado.

CAPÍTULO 6: DESARROLLO DEL SOFTWARE

En este capítulo de describe con detalle todo el proceso práctico llevado a cabo con el software ROS y sus herramientas para realizar la simulación del presente trabajo.

Paquetes de ROS y Gazebo utilizados

A lo largo de este documento, se ha comentado varias veces que una de las grandes ventajas que dispone ROS es que se trata de un software colaborativo y libre, y permite utilizar el trabajo previo de otras personas como punto de partida.

Los paquetes, ya creados, que se han empleado para la presente aplicación son:

- *Moveit_tutorials*: Este paquete contiene toda la información necesaria para empezar aprender a utilizar todas las funcionalidades de *MoveIt!* de forma sencilla por medio de la aplicación *RViz*. Este paquete se complemente con la siguiente página web:

  MoveIt Tutorials — moveit_tutorials Noetic documentation (ros-planning.github.io)

  En este trabajo, se ha utilizado el paquete a modo de aprendizaje de la herramienta y, posteriormente, la verificación de los nodos y *topics* correspondientes para el correcto funcionamiento de la aplicación.
- Universal_robots: Este repositorio contiene toda la información relevante para poder controlar los robots desarrollados por Universal Robots en ROS. Dentro del repositorio se pueden encontrar los distintos archivos de todos los modelos, desde el UR3 hasta el UR10, dentro de la carpeta *ur_description*; sus paquetes de configuración para poder ser controlados y simulados con *MoveIt!*, dentro de las carpetas *urX_moveit_config*; los archivos necesarios para poder simular los modelos en Gazebo, en la carpeta *ur_gazebo*, a su vez, se encuentran los paquetes para poder simular los robots desde ROS (*ur_bringup*); y, por último, el paquete incluye toda información para que el robot pueda moverse como corresponde, calculando los problemas de la cinemática directa e indirecta, dentro del paquete *ur_kinematics*.
- *MoveIt!*: Con este paquete se basa completamente esta aplicación. Su explicación se encuentra desarrollada en el capítulo 4 de este documento.
- *Gazebo_ros_pkgs*: A parte del paquete de *MoveIt!*, la aplicación también se base en este conjunto de paquetes ROS, porque proporcionan las interfaces necesarias

para simular los modelos en el simulador 3D de Gazebo junto con los distintos *plugins* necesarios que se utilizan en la aplicación, como son la webcam y la ventosa al vacío, y los controladores. Se integran con ROS a través de mensajes ROS, servicios y reconfiguración dinámica.

Como se observa, entre los paquetes empleados, no existe ninguno que conecte con robots reales y poder controlarlos desde los programas de ROS, eso es debido a que el trabajo está basado íntegramente en la simulación del modelo en *Gazebo* y de la aplicación realizada para este trabajo. La conexión con el robot real es uno de los trabajos futuros expuestos en el capítulo 8 de este documento. Pero mencionar que existe un paquete, llamado *ur_modern_driver*, que sustituye al paquete *ur_bringup* y su función es permitir la conexión con el robot real.

6.1 PASOS PREPARATIVOS

Para los próximos apartados, primero se ha comentar todo el trabajo previo.

Antes de realizar las modificaciones, se crea un *workspace* llamado “PROYECTO_ws”, donde se encuentran todos los paquetes que se necesitan para realizar el trabajo. En el *workspace*, se crea una carpeta (*ur5_ROS-Noetic*) donde se guarda el modelo UR5, con su configuración para el paquete de *MoveIt!* (*universal_robot*), y el paquete donde se encuentre los nodos creados para mover el brazo, activar la ventosa al vacío y activar la webcam, además de los distintos elementos que se introducen el mundo de gazebo (*ur5_aplicacion*). También, en el *workspace*, se añade el paquete *gazebo_ros_pkgs*, ya mencionado, para que no se produzca ningún error a la hora de utilizar los *plugins* necesarios. Este paquete se puede encontrar en el siguiente enlace:

GitHub - ros-simulation/gazebo_ros_pkgs: Wrappers, herramientas y API adicionales para usar ROS con Gazebo

El repositorio que se ha utilizado para tener los modelos de *Universal Robots* ya diseñados se encuentran en el enlace siguiente:

GitHub - ros-industrial/universal_robot at kinetic-devel

Hay que destacar que se ha utilizado *ROS Noetic* por la versión de Ubuntu, pero el repositorio que se ha utilizado es el de *ROS Kinetic* (una versión anterior), porque presenta una forma de organización más comprensible. Este cambio no produce ningún problema a la hora de lanzar el robot y ejecutar los nodos, como se muestra en el presente trabajo.

6.2 MODIFICACIONES DEL MODELO UR5

En capítulos anteriores se explica que se necesita un fichero que contenga el modelo del robot, o la información referente a este, llamado URDF. Como se ha mencionado cuando se han explicado los paquetes que se van a utilizar, el modelo del UR5 viene configurado en el paquete *ur_description* dentro del repositorio de *github* de *Universal_robots*. Pero en este paquete, se modela el robot sin elementos terminales ni sensores añadidos, por ello, es necesario modificar el archivo implementando la ventosa al vacío como *end_effector* y la webcam como sensor.

Partiendo del fichero *ur5.urdf.xacro* del paquete *ur_description*, se quiere añadir al UR5 los materiales explicados en el capítulo 3. Como se menciona en dicho capítulo los elementos que se añaden son genéricos y no se utilizan ficheros CAD específicos de cada elemento, por su caso, se han utilizado formas geométricas primitivas para simular su implementación en el brazo. Para el caso de la webcam, se añade al brazo acoplado a la muñeca 3 por medio del *link* llamado *${prefix}ee_link*, y se presenta como una forma cúbica de color verde. Por otro lado, la ventosa al vacío se añade a la muñeca 3 por medio del *link ${prefix}ee_link*, al igual que la web cam, y se presenta como una forma cilíndrica de color blanco. A continuación, se presenta una parte del archivo *ur5.urdf.xacro*, el código entero con las modificaciones realizadas se encuentra en los ficheros adjuntos al presente trabajo:

```
<gazebo>
  <plugin name="gazebo_ros_vacuum_gripper" filename="libgazebo_ros_vacuum_gripper.so">
    <robotNamespace>/ur5/vacuum_gripper</robotNamespace>
    <bodyName>vacuum_gripper</bodyName>
    <topicName>grasping</topicName>
    <maxForce>20</maxForce>
    <maxDistance>0.05</maxDistance>
    <minDistance>0.01</minDistance>
  </plugin>
</gazebo>
```

Esta parte del código muestra el *plugin* que se utiliza en la simulación para que se active la ventosa al vacío. Como se observa, el *plugin* de la ventosa se utiliza en gazebo, especificando el nombre del archivo de la biblioteca compartida y presenta una serie de

parámetros que se deben ajustar para la correcta ejecución. Lo primero es relacionar el *robotNamespace* con el complemento asociado, que, en este caso, es la ventosa al vacío, y para que el complemento interactúe con el cuerpo correspondiente, se especifica el nombre del cuerpo en el *bodyName*. Por último, se establece el nombre del *topic* por el cual se utiliza para enviar los comandos de control a la ventosa al vacío. Los otros parámetros especifican la máxima fuerza que emplea la ventosa y el rango de funcionamiento.

El aspecto final del modelo es el que se puede observar en Figura 34:

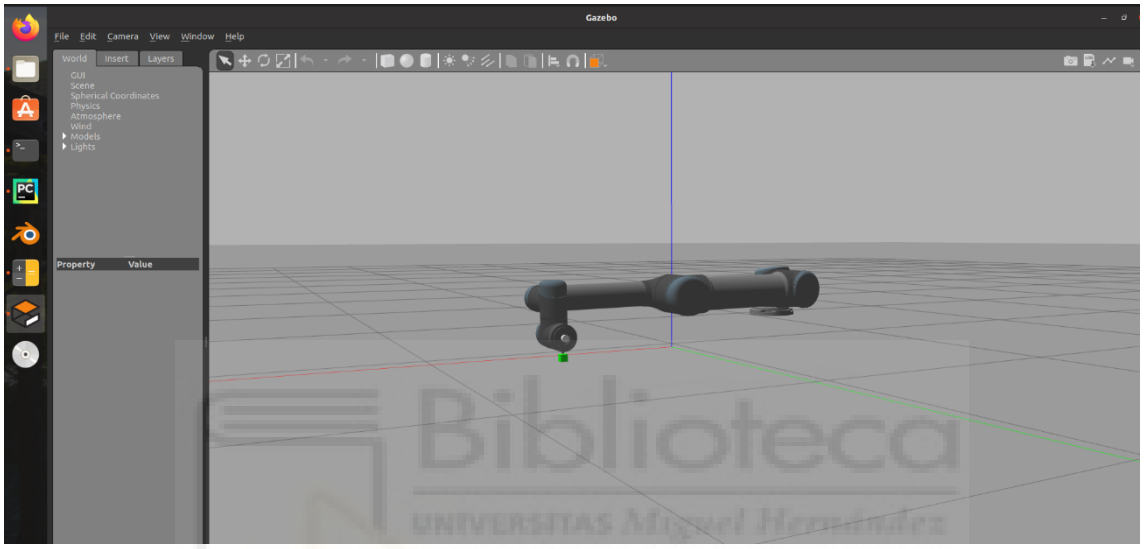


Figura 34. Modelo del UR5 con la webcam y ventosa implementadas

Los archivos que han sido modificados se mantendrán en las carpetas correspondientes, sin necesidad de crear otros paquetes. Pero se tendrá que modificar otros archivos secundarios para la correcta ejecución del programa.

6.3 CREACIÓN DEL ENTORNO EN GAZEBO

Después de modificar el modelo del brazo UR5 con las herramientas adecuadas, se dispone a crear el mundo de *Gazebo* donde aparezcan el robot, una cinta de transporte, una caja donde colocar los objetos recogidos y el objeto a recoger colocado sobre la cinta. Para llevarlo a cabo, se crea por separado cada elemento.

Estos scripts se han creado en una carpeta llamada *urdf* dentro del paquete *ur5_aplicacion*, en los archivos adjuntos al presente trabajo se encuentran los códigos enteros de cada elemento que se ha añadido al mundo de gazebo.

A continuación, se muestra la parte inicial del código creado para que aparezca la cinta de transporte, *conveyor_belt.urdf*:

```
<?xml version="1.0"?>
<robot name="conveyor_belt">

<!-- conveyor belt is just a long flat box for objects to slide on it -->

<!-- world link -->
<link name="world"/>

<!-- base_link and its fixed joint -->
<joint name="joint_fix" type="fixed">
  <parent link="world"/>
  <child link="base_link"/>
</joint>
```

Esta parte de código muestra la unión que existe entre el eslabón principal de la cinta con el mundo de gazebo mediante una articulación fija, indicando que no presenta ejes de rotación.

El siguiente código corresponde a un eslabón de la caja donde se colocan los objetos después de ser cogidos, *bin.urdf*:

```
<link name ="bottom">
  <collision>
    <origin xyz="0 0 -0.1" rpy="0 0 0"/>
    <geometry>
      <box size="0.4 0.6 0.0001"/>
    </geometry>
  </collision>

  <visual>
    <origin xyz="0 0 -0.1" rpy="0 0 0"/>
    <geometry>
      <box size="0.4 0.6 0.0001"/>
    </geometry>
  </visual>

  <inertial>
    <origin xyz="0 0 -0.1" rpy="0 0 0" />
    <mass value="0.05" />
    <inertia
      ixx="0.001" ixy="0.0"  ixz="0.0"
      iyy="0.001" iyz="0.0"
      izz="0.001" />
  </inertial>
</link>
```

En código mostrado previamente, se crea un eslabón llamado *bottom* que corresponde a la parte de debajo de la caja, donde se especifica ciertos parámetros que se deben definir para que en el mundo de gazebo aparezca el objeto correctamente. Esos parámetros son

su colisión (define las propiedades relacionadas con la colisión del eslabón con el modelo), su visual (define las propiedades relacionadas con la representación visual del eslabón el modelo) y su inercia (define las propiedades inerciales del eslabón, que sirven para el cálculo de la dinámica en la simulación).

Y, por último, se crea el objeto a recoger, *red_box.urdf*, con unas modificaciones para el reconocimiento con la webcam, y que se explicará después. El código que se muestra a continuación corresponde a la última parte del programa que crea el objeto a recoger. En él se comenta la articulación fija que se ha añadido para agregarle al objeto una cara de distinto color, y esos colores se establecen después de declarar la articulación:

```
 <joint name="joint_top" type="fixed">
   <parent link="base_link"/>
   <child link="face_red"/>
   <origin xyz="0 0 0" rpy="0 0 0"/>
 </joint>

 <gazebo reference="base_link">
  <material>Gazebo/White</material>
    <mu1>5</mu1>
    <mu2>5</mu2>
 </gazebo>

 <gazebo reference="face_red">
  <material>Gazebo/Red</material>
    <mu1>5</mu1>
    <mu2>5</mu2>
 </gazebo>

</robot>
```

6.4 PROGRAMAS PARA GENERAR LOS CUBOS

Cuando se inicia la simulación, ésta no se detiene, conforme el brazo recoge y deposita el objeto ya cogido, se genera un nuevo objeto. Para que se genere ese objeto, se crea dos programas en C++.

- El primer fichero, *blocks_spawner.cpp*, actúa como nodo y se encarga de recibir el archivo del objeto y modificar su orientación o giro. La parte que se muestra es la parte del código que se encarga de realizar los cambios de orientación de los objetos de forma aleatoria entre los valores de -0.4 y 0.4:

```
std::stringstream red_strStream;
std::string red_xmlStr;
doc.save(red_strStream,"  ");
red_xmlStr = red_strStream.str();
spawn_model_req.initial_pose.position.x = 0;
```

```
    spawn_model_req.initial_pose.position.y = 0;
    spawn_model_req.initial_pose.position.z = 0.25;
    spawn_model_req.initial_pose.orientation.x=0.0;
    // spawn_model_req.initial_pose.orientation.x= - (M_PI / 4.0 + 0.45) ;
    spawn_model_req.initial_pose.orientation.y=0.0;
    spawn_model_req.initial_pose.orientation.z=0.2;
    spawn_model_req.initial_pose.orientation.w=1.0;
    spawn_model_req.reference_frame = "world";
    ros::Time time_temp(0, 0);
    ros::Duration duration_temp(0, 1000000);
    ros::Duration(1.0).sleep();
       std::string index = intToString(i);
       std::string model_name;
       spawn_model_req.initial_pose.orientation.z = (float)rand()/(float)(RAND_MAX) * 0.4;  // random
between -0.4 to 0.4
       ROS_INFO_STREAM("z orientation of new box: "
       << spawn_model_req.initial_pose.position.y);
```

- El segundo fichero, *blocks_poses_publisherr.cpp*, actúa como nodo y es el encargado de ir publicando los detalles del objeto. La parte de código que se muestra, es un bucle donde se van actualizando los valores de los *poses* y publicándolos:

```
  while (ros::ok()) {
    if (g_poses_updated) {
      g_poses_updated = false;
      int local_quantity = g_x.size();
      current_poses_msg.x.resize(local_quantity);
      current_poses_msg.y.resize(local_quantity);
      current_poses_msg.z.resize(local_quantity);
      current_poses_msg.x = g_x;
      current_poses_msg.y = g_y;
      current_poses_msg.z = g_z;
      poses_publisher.publish(current_poses_msg);
    }
    ros::spinOnce();
  }
  return 0;
}
```

6.5 MENSAJES

Antes de hacer los programas que permitan el movimiento del brazo, entre ellos se tienen que comunicar entre sí, para que haya una buena conexión entre ellos. Los mensajes que se han creado son:

- *blocks_info.msg:*

```
string name
float64 x  # x size in the world
float64 y  # y size in the world
float64 z  # z size in the world
```

- *blocks_poses.msg:*

```
float64[] x  # x coordinate in the world
float64[] y  # y coordinate in the world
float64[] z  # z coordinate in the world
```

- *Tracker.msg:*

```
string name
float64 x  # x coordinate in the world
float64 y  # y coordinate in the world
float64 z  # z coordinate in the world
float64 error_x
float64 error_y
float64 error_z
bool flag1
bool flag2
bool flag3
bool face_top
float64 roll
```

6.6 APLICACIÓN PARA EL PROCESAMIENTO DE LA IMAGEN DE LA WEBCAM

Este programa, *ur5_vision.py*, escrito en Python, activa la webcam implementada en el robot. Se encarga de detectar el color correspondiente y calcula el centro de la cara del objeto, para enviarle esos datos al programa encargado para mover el robot. Además, calcula la orientación del objeto para que también le envíe esa información y la muñeca 3 del robot se ajuste adecuadamente.

```
#!/usr/bin/python3.8
import cv2
import cv_bridge
import numpy as np
import rospy
from sensor_msgs.msg import Image
```

```
from ur5_mensajes.msg import Tracker
tracker = Tracker()
class ur5_vision:
    def __init__(self):

        self.face_top = True
        self.track_flag = False
        self.default_pose_flag = True
        self.cx = 400.0
        self.cy = 400.0

        rospy.init_node("ur5_vision", anonymous=False)

        self.bridge = cv_bridge.CvBridge()

        self.image_sub = rospy.Subscriber('/ur5/usbcam/image_raw', Image, self.image_callback)

        self.cxy_pub = rospy.Publisher('cxy', Tracker, queue_size=1)

        # Estas son las matrices de calibración que se han calculado.
        self.K = np.array([[476.748394, 0.0, 399.475799],
                           [0.0, 476.725312, 399.270110],
                           [0.0, 0.0, 1.0]])
        self.D = np.array([9.931872964111051e-05, -0.00034175122438275603,
                           -0.0001022084459179837, 1.05968856143981e-05, 0.0])
        self.R = np.eye(3)
        self.P = np.array([[476.713417, 0.0, 399.486656, 0.0],
                           [0.0, 476.677986, 399.167259, 0.0],
                           [0.0, 0.0, 1.0, 0.0]])
    def image_callback(self, msg):

        image = self.bridge.imgmsg_to_cv2(msg, desired_encoding='bgr8')
        corrected_image = cv2.undistort(image, self.K, self.D)

        hsv = cv2.cvtColor(corrected_image, cv2.COLOR_BGR2HSV)

        lower_red = np.array([0, 100, 100])
        upper_red = np.array([10, 255, 255])
        mask = cv2.inRange(hsv, lower_red, upper_red)
        cnts, _ = cv2.findContours(mask.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

        h, w, d = corrected_image.shape

        M = cv2.moments(mask)
        if M['m00'] > 0:
            cx = int(M['m10'] / M['m00'])
            cy = int(M['m01'] / M['m00'])

            for i, c in enumerate(cnts):
                area = cv2.contourArea(c)
                if area > 70:
```

```
                self.face_top = True
                self.cx = cx
                self.cy = cy
                self.error_x = self.cx - w / 2
                self.error_y = self.cy - (h / 2 + 190)
                tracker.x = cx
                tracker.y = cy
                tracker.error_x = self.error_x
                tracker.error_y = self.error_y

                if 200 < self.cy < 265 and self.face_top:
                    self.face_top = False
                    tracker.face_top = self.face_top

                    rect = cv2.minAreaRect(c)
                    box = cv2.boxPoints(rect)
                    box = np.intp(box)
                    delta_x = box[2][0] - box[1][0]
                    delta_y = box[2][1] - box[1][1]
                    angle_radians = round(np.arctan2(delta_y, delta_x), 2)
                    roll = round((np.pi / 2), 2) - angle_radians
                    tracker.roll = roll

                if self.face_top:
                    tracker.face_top = self.face_top
                    rect = cv2.minAreaRect(c)
                    box = cv2.boxPoints(rect)
                    box = np.intp(box)
                    delta_x = box[2][0] - box[1][0]
                    delta_y = box[2][1] - box[1][1]
                    angle_radians = round(np.arctan2(delta_y, delta_x), 2)
                    roll = round((np.pi / 2), 2) - angle_radians
                    tracker.roll = roll

                cv2.circle(corrected_image, (cx, cy), 10, (0, 0, 0), -1)
                cv2.putText(corrected_image, "({}, {})".format(int(cx), int(cy)), (int(cx - 5), int(cy + 15)),
                            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 1)

                cv2.drawContours(corrected_image, cnts, -1, (255, 255, 255), 1)
                break
            else:
                rospy.logerr('NO SE ENCONTRÓ LA CARA ROJA DEL CUBO')

        self.cxy_pub.publish(tracker)

        cv2.namedWindow("window", 1)
        cv2.imshow("window", corrected_image)
        cv2.waitKey(1)


follower = ur5_vision()
rospy.spin()
```

Una vez se ha mostrado el programa que permite la detección del objeto, se explica como se calcula el centro de la cara por la cual debe coger el objeto y su orientación. Para ello, primero se crea la clase al cual se suscribe al *topic* "/ur5/usbcam/image_raw", recibe cada 10 fps (frames por segundo que se establece en el archivo *ur.gazebo.xacro*) información de lo que capta la webcam, además de ir llamado a la función que detecta la cara correspondiente y su orientación, y la siguiente clase que es la encargada de publicar esa información para que sea recibida al programa encargado de realizar el movimiento del brazo.

La función encargada de esos cálculos transforma la imagen que capta la cámara en un formato que pueda ser leído por la herramienta ROS mediante la librería *cv_Bridge* y se corrige la imagen para que el cálculo sea más preciso. La imagen resultante se encuentra en la representación de colores BGR, pero en este caso, se necesita pasar la imagen a la representación HSV, ya que resulta más fácil trabajar con esta representación a la hora de detectar colores. El color de la cara que se desea detectar es de color rojo y, para ello, se crea una máscara donde se establece el valor bajo y el valor alto del tono, saturación y brillo, para que detecte una amplia gama del color rojo, y devuelve el contorno de la cara roja, que se muestra en la ventana emergente tras ejecutar el programa. Mediante la máscara creada con el rango de tonos que debe detectar y una función de OpenCv llamada *moments,* que se encarga para extraer características útiles y realizar la detección de contornos, cálculo de centroides, áreas, orientación y otros descriptores. Tras crear el objeto, que es de tipo "Moments", y que contiene los momentos calculados, se utilizan los momentos "00", "01" y "10", que representan el área, la posición horizontal del centroide y la posición vertical del centroide, respectivamente, se procede al cálculo del centro de la cara, dividiendo cada posición entre el área, para normalizar el resultado y se calcule en relación con el área del objeto, y así se obtiene el valor del centro de la cara y para que el brazo se mueva a esa posición, no se envía esos valores, sino que se calcula, respecto de los valores del centro en cada instante, el error entre la posición del extremo del brazo y el centro del objeto, y este error, tanto en horizontal como vertical, se envía al programa que permite el movimiento. Para el cálculo de la orientación, se realiza de forma distinta, el procedimiento que se ha seguido es el siguiente, primero se crea un rectángulo con los valores del contorno calculados y en ese rectángulo se añaden las esquinas que se detectan de la cara; tras detectar las esquinas, se calcula la diferencia que existe entre ellas respecto de la posición horizontal como vertical y se guardan en dos variables distintas; por último, para establecer la orientación, se calcula el ángulo que hay

entre las diferencias de las posiciones de las esquinas y la línea horizontal de la webcam, cuyo resulta es un ángulo en radianes al cuál se debe restar 90º para que la información que se envíe al programa encargado de ejecutar el movimiento del brazo sea la correspondiente, y se realiza esta corrección porque las esquinas establecidas no son las óptimas.

6.7 APLICACIÓN PARA LA ACTIVACIÓN DE LA VENTOSA

Este programa, *ur5_gripper.py*, escrito también en Python, solamente se encarga de activar el plugin de la ventosa al vacío cuando esté suficientemente cerca el *end_effector* de la cara correspondiente, y se desactiva cuando se encuentra encima de la caja. Para saber cuándo activarse o desactivarse recibe información del programa que se encarga de mover el brazo:

```
#!/usr/bin/python3.8
import rospy
from ur5_mensajes.msg import Tracker
from std_msgs.msg import Bool
# from std_srvs.srv import Empty
from gazebo_msgs.srv import SpawnModel
def gripper_status(msg):
    if msg.data:
        return True
def gripper_on():

    rospy.wait_for_service('/ur5/vacuum_gripper/on')
    try:

        turn_on = rospy.ServiceProxy('/ur5/vacuum_gripper/on', service_class=SpawnModel)

        resp = turn_on(spawnModel, None, None, None, None)
        return resp
    except rospy.ServiceException as e:

        print("Service call failed: %s" % e)

def gripper_off():
    rospy.wait_for_service('/ur5/vacuum_gripper/off')
    try:
        turn_off = rospy.ServiceProxy('/ur5/vacuum_gripper/off', SpawnModel)
        resp = turn_off(spawnModel, None, None, None, None)
        return resp
    except rospy.ServiceException as e:
        print("Service call failed: %s" % e)
```

```
def trigger(msg):
    global spawnModel
    spawnModel = msg.name
    if msg.flag2:
        gripper_on()
    else:
        gripper_off()

rospy.init_node("ur5_gripper", anonymous=False)

spawnModel = "cube"

gripper_status_sub = rospy.Subscriber('/ur5/vacuum_gripper/grasping', Bool, gripper_status,
queue_size=1)

gripper_sub = rospy.Subscriber('gripperOnOff', Tracker, trigger, queue_size=1)
rospy.spin()
```

6.8 APLICACIÓN PARA CALCULAR Y EJECUTAR EL MOVIMIENTO DEL BRAZO

El último programa, en Python también, es el encargado de mover el brazo robótico mediante el cálculo de la cinemática directa e inversa y las distintas funciones que ofrece ROS. Este programa recibe la posición del robot en cada instante para saber dónde se encuentra en todo momento y los publica para que los demás programas reciban esa información. El programa, llamado *ur5_mp.py*, es el siguiente:

```
#!/usr/bin/python3.8
from copy import deepcopy
import moveit_commander
import rospy
import sys
import tf
from geometry_msgs.msg import Point, Pose, Quaternion
from ur5_mensajes.msg import Tracker
from ur5_mensajes.msg import blocks_info
tracker = Tracker()
class ur5_mp:

    def blocks_info_callback(self, msg):

        self.arm.set_named_target("pickup")
        self.arm.go(wait=True)
        rospy.sleep(3)

        if not self.face_top:

            self.arm.set_named_target("around")
```

```
self.arm.go(wait=True)
rospy.sleep(2)

self.flag_orient = 1
for t in range(1, 12):

    start_pose = self.arm.get_current_pose(self.end_effector_link).pose

    self.waypoints = []

    wpose = deepcopy(start_pose)

    wpose.position.x -= self.error_x * 0.03 / 105
    wpose.position.y += (self.error_y - 190) * 0.02 / 105
    wpose.position.z = 0.07
    self.waypoints.append(deepcopy(wpose))

    self.arm.set_start_state_to_current_state()

    plan, fraction = self.arm.compute_cartesian_path(self.waypoints, 0.01, 0.0, True)

    if 1 - fraction < 0.2:
        rospy.loginfo("Path computed successfully. Moving the arm.")
        num_pts = len(plan.joint_trajectory.points)
        rospy.loginfo("\n# intermediate waypoints = " + str(num_pts))
        self.arm.execute(plan)
        rospy.loginfo("Path execution complete.")
    else:
        rospy.loginfo("Path planning failed")

if self.flag_orient == 1:
    self.top_joint_states = self.arm.get_current_joint_values()
    self.top_joint_states[4] = 0.78 + round(self.roll, 2)
    self.arm.set_joint_value_target(self.top_joint_states)
    self.arm.set_start_state_to_current_state()
    plan = self.arm.plan()
    self.arm.execute(plan[1])
    self.arm.stop()
    self.arm.clear_pose_targets()
    self.flag_orient = 2
else:
    pass

rospy.sleep(1)

tracker.flag2 = 1
tracker.name = msg.name
self.gripper_pub.publish(tracker)
rospy.sleep(2)

self.arm.set_named_target("pickup")
self.arm.go(wait=True)
```

```
            self.block_pickedup_pub.publish(tracker)

            self.arm.set_named_target("drop")
            self.arm.go(wait=True)
            droppose = self.arm.get_current_pose(self.end_effector_link).pose
            droppose.position.x = -0.5 - 0
            droppose.position.y = 1.0 - 0.7
            self.arm.set_pose_reference_frame(self.reference_frame)
            self.arm.set_pose_target(droppose, self.end_effector_link)
            self.arm.go(wait=True)
            self.arm.stop()
            self.arm.clear_pose_targets()
            tracker.flag2 = 0
            self.gripper_pub.publish(tracker)

            if 1.55 == self.roll or self.roll == 1.56 or self.roll == 1.57 or 0.01 == self.roll or self.roll == 0.02
or self.roll == 0.00:
                orient = Quaternion(*tf.transformations.quaternion_from_euler(3.14, 1.57, 0))
                pose_goal = Pose(Point(0, -0.7, msg.z + .02), orient)
                self.arm.set_pose_reference_frame(self.reference_frame)
                self.arm.set_pose_target(pose_goal, self.end_effector_link)
                self.arm.go(wait=True)
                self.arm.stop()
                self.arm.clear_pose_targets()
                tracker.flag2 = 1
                tracker.name = msg.name
                self.gripper_pub.publish(tracker)
                self.arm.set_named_target("pickup")
                self.arm.go(wait=True)
                self.block_pickedup_pub.publish(tracker)
                self.arm.set_named_target("drop")
                self.arm.go(wait=True)
                droppose = self.arm.get_current_pose(self.end_effector_link).pose
                droppose.position.x = -0.5 - 0
                droppose.position.y = 1.0 - 0.7
                self.arm.set_pose_reference_frame(self.reference_frame)
                self.arm.set_pose_target(droppose, self.end_effector_link)
                self.arm.go(wait=True)
                self.arm.stop()
                self.arm.clear_pose_targets()
                tracker.flag2 = 0
                self.gripper_pub.publish(tracker)
            else:
                self.flag_orient = 1
                for t in range(1, 9):
                    start_pose = self.arm.get_current_pose(self.end_effector_link).pose

                    self.waypoints = []

                    wpose = deepcopy(start_pose)

                    wpose.position.x -= self.error_x * 0.01 / 105
```

```
    wpose.position.y += self.error_y * 0.01 / 105
    wpose.position.z = 0.30

    self.waypoints.append(deepcopy(wpose))

    self.arm.set_start_state_to_current_state()

    plan, fraction = self.arm.compute_cartesian_path(self.waypoints, 0.01, 0.0, True)

    if 1 - fraction < 0.2:
        rospy.loginfo("Path computed successfully. Moving the arm.")
        num_pts = len(plan.joint_trajectory.points)
        rospy.loginfo("\n# intermediate waypoints = " + str(num_pts))
        self.arm.execute(plan)
        rospy.loginfo("Path execution complete.")
    else:
        rospy.loginfo("Path planning failed")
if self.flag_orient == 1:
    self.top_joint_states = self.arm.get_current_joint_values()
    self.top_joint_states[5] = 1.45 - round(self.roll, 2)
    self.arm.set_joint_value_target(self.top_joint_states)
    self.arm.set_start_state_to_current_state()
    plan = self.arm.plan()
    self.arm.execute(plan[1])
    self.arm.stop()
    self.arm.clear_pose_targets()
    self.flag_orient = 2
else:
    pass
drop_pose = self.arm.get_current_pose(self.end_effector_link).pose
self.waypoints = []
wpose = deepcopy(drop_pose)
wpose.position.z = msg.z + .02
self.waypoints.append(deepcopy(wpose))
self.arm.set_start_state_to_current_state()
plan, fraction = self.arm.compute_cartesian_path(self.waypoints, 0.01, 0.0, True)
if 1 - fraction < 0.2:
    rospy.loginfo("Path computed successfully. Moving the arm.")
    num_pts = len(plan.joint_trajectory.points)
    rospy.loginfo("\n# intermediate waypoints = " + str(num_pts))
    self.arm.execute(plan)
    rospy.loginfo("Path execution complete.")
else:
    rospy.loginfo("Path planning failed")
rospy.sleep(1)
tracker.flag2 = 1
tracker.name = msg.name
self.gripper_pub.publish(tracker)
self.arm.set_named_target("pickup")
self.arm.go(wait=True)
self.block_pickedup_pub.publish(tracker)
self.arm.set_named_target("drop")
```

```
            self.arm.go(wait=True)
            droppose = self.arm.get_current_pose(self.end_effector_link).pose
            droppose.position.x = -0.5 - 0
            droppose.position.y = 1.0 - 0.7
            self.arm.set_pose_reference_frame(self.reference_frame)
            self.arm.set_pose_target(droppose, self.end_effector_link)
            self.arm.go(wait=True)
            self.arm.stop()
            self.arm.clear_pose_targets()
            tracker.flag2 = 0
            self.gripper_pub.publish(tracker)
    def __init__(self):

        rospy.init_node("ur5_mp", anonymous=False)
        rospy.loginfo("Starting node ur5_mp")

        rospy.on_shutdown(self.cleanup)

        self.cxy_sub = rospy.Subscriber('cxy', Tracker, self.tracking_callback, queue_size=1)

        self.gripper_pub = rospy.Publisher('gripperOnOff', Tracker, queue_size=1)

        self.block_info_sub = rospy.Subscriber('blocks_info', blocks_info,
callback=self.blocks_info_callback)

        self.block_pickedup_pub = rospy.Publisher('block_pickedup', Tracker, queue_size=1)

        moveit_commander.roscpp_initialize(sys.argv)
        self.arm = moveit_commander.MoveGroupCommander('manipulator', wait_for_servers=10.0)

        self.end_effector_link = self.arm.get_end_effector_link()

        self.reference_frame = "base_link"
        self.waypoints = []
        self.flag_orient = 1

        self.arm.allow_replanning(True)

        self.arm.set_goal_tolerance(0.01)
        self.arm.set_planning_time(0.5)
        self.arm.set_max_acceleration_scaling_factor(.00001)
        self.arm.set_max_velocity_scaling_factor(.5)

    def cleanup(self):
        rospy.loginfo("Stopping the robot")

        self.arm.stop()

        rospy.loginfo("Shutting down Moveit!")
        moveit_commander.roscpp_shutdown()
        moveit_commander.os._exit(0)
```

```
    def tracking_callback(self, msg):

        self.cx = msg.x
        self.cy = msg.y
        self.error_x = msg.error_x
        self.error_y = msg.error_y
        self.face_top = msg.face_top
        self.roll = msg.roll
mp = ur5_mp()

rospy.spin()
```

6.9 ARCHIVO DE LANZAMIENTO

Toda la simulación se lanza desde un solo archivo, llamado *proyecto_final.launch*, donde en él se activan los programas anteriores como nodos, se carga un mundo vacío en *Gazebo* donde se colocan los elementos creados por separado, se carga el nuevo modelo del brazo UR5 y por último, se carga y activa los controladores del brazo y el paquete de *MoveIt!*. El archivo de lanzamiento es el siguiente:

```
<?xml version="1.0"?>
<launch>
  <param name="red_box_path" type="str" value="$(find ur5_aplicacion)/urdf/red_box.urdf"/>
  <arg name="limited" default="true"/>
  <arg name="paused" default="false"/>
  <arg name="gui" default="true"/>
  <arg name="debug" default="false" />
  <arg name="sim" default="true" />
  <!-- Crea el mundo vacío de simulación -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
    <arg name="world_name" default="worlds/empty.world"/>
    <arg name="paused" value="$(arg paused)"/>
    <arg name="gui" value="$(arg gui)"/>
  </include>
  <!-- Genera el brazo UR5 -->
  <include file="$(find ur_description)/launch/ur5_upload.launch">
    <arg name="limited" value="$(arg limited)"/>
  </include>
  <!-- Se añade como un nodo el spawn del robot UR5 con unos valores definidos de colocación -->
  <node name="spawn_gazebo_model" pkg="gazebo_ros" type="spawn_model" args="-urdf -param
robot_description -model robot -z 0.2 -y 0.7" respawn="false" output="screen" />
  <!-- Se incluye el controlador -->
  <include file="$(find ur_gazebo)/launch/controller_utils.launch"/>
  <!-- Se añade como parámetro el controlador del UR5 -->
  <rosparam file="$(find ur_gazebo)/controller/arm_controller_ur5.yaml" command="load"/>
  <!-- El parámetro anterior es para establecer el controlador del siguiente controlador -->
  <node name="arm_controller_spawner" pkg="controller_manager" type="controller_manager"
args="spawn arm_controller" respawn="false" output="screen"/>
```

```
  <!-- Remapear el follow_joint_trajectory -->
  <remap if="$(arg sim)" from="/follow_joint_trajectory" to="/arm_controller/follow_joint_trajectory"/>
  <!-- Lanzar el paquete de MoveIt -->
  <include file="$(find ur5_moveit_config)/launch/move_group.launch">
    <arg name="limited" default="$(arg limited)"/>
    <arg name="debug" default="$(arg debug)" />
  </include>
  <!-- generar la cinta transportadora -->
  <node name="spawn_conveyor_belt" pkg="gazebo_ros" type="spawn_model" args="-file $(find ur5_aplicacion)/urdf/conveyor_belt.urdf -urdf -model conveyor_belt" />
  <!-- generar la caja -->
  <node name="bin" pkg="gazebo_ros" type="spawn_model" args="-file $(find ur5_aplicacion)/urdf/bin.urdf -urdf -model bin -y 1.0 -x -0.5 -z 0.1" />
  <!-- crear el nodo del movimiento del robot -->
  <node name="ur5_mp" pkg="ur5_aplicacion" type="ur5_mp.py" output="screen" />
  <!-- crear el nodo de la visión del robot -->
  <node name="ur5_vision" pkg="ur5_aplicacion" type="ur5_vision.py" output="screen"/>
  <!-- generar los cubos -->
  <node name="blocks_spawner" pkg="ur5_aplicacion" type="blocks_spawner" output="screen" />
  <!-- nodo del controlador -->
  <node name="robot_controller" pkg="ur5_aplicacion" type="robot_controller.py" output="screen" />
  <!-- nodo del publicador de los cubos -->
  <node name="blocks_poses_publisher" pkg="ur5_aplicacion" type="blocks_poses_publisher" output="screen" />
  <!-- crear el nodo del gripper del robot -->
  <node name="ur5_gripper" pkg="ur5_aplicacion" type="ur5_gripper.py" output="screen"/>
</launch>
```

CAPÍTULO 7: RESULTADOS

Después de observar la simulación completa del robot con las funciones adecuadas que debía hacer en distintas situaciones que se podría dar, los ha podido super con éxito, pudiendo aplicarse a la industria, como se detalla a continuación.

7.1 APLICACIONES

Observando toda la automatización de la industria y de las fábricas, con las cadenas de montaje y cintas transportadoras donde no se observa a ningún trabajador y solo hay robots, una posible aplicación es en la manipulación de objetos que llegan por una cinta transportadora, donde el brazo debe coger los objetos de una determinada forma o desde una cara concreta y si, por alguna condición externa, ese objeto no llega de forma correcta, que se ha volcado o girado, que el robot se capaz de detectar donde se encuentra la cara correspondiente o calcular la orientación que debe girar el extremo para poder coger el objeto correctamente, sin que ningún operario actúe para colocar el objeto como debería estar.

7.2 CAPTURAS DE LA APLICACIÓN

En este apartado de los resultados, se muestran todas las capturas de pantalla que muestran el cómo funciona la aplicación sobre el objeto que se tiene que recoger. La aplicación consiste en una cinta transportadora que simula un objeto detenido en ella, como si un sensor la detectase y parase la cinta. Cuando se genera el cubo, el robot se coloca en la posición de reposo y detecta la cara roja, que es la cara que debe reconocer y calcular el movimiento para colocar el efector final centrado en el objeto, y después proceder con el giro de la muñeca 3, para orientarse y cuadrarse al cubo. Una vez se haya centrado, se acerca al cubo y activa la ventosa al vacío, y lleva el cubo a la caja y lo suelta. Las primeras imágenes corresponden cómo se comporta el robot cuando detecta que la cara se encuentra en la cara de arriba del cubo, y funciona correctamente. Pero las imágenes de después, muestra cómo se comporta el robot cuando detecta que la cara roja se encuentra en un lado, y funciona bien hasta que se activa la ventosa al vacío porque el plugin de la ventosa no funciona correctamente, solo funciona en vertical. En la simulación se produce ciertos errores de aproximación, debido a que se ha trabajado

mediante una máquina virtual, y se ha cogido parte de los decimales, y la precisión no es exacta.

Las capturas que se muestran a continuación corresponden a las de la primera aplicación, donde se observa que la webcam detecta que la cara roja del objeto se encuentra arriba y el brazo realiza los movimientos que ha calculado el programa y consigue centrarse, al igual que la orientación de la muñeca 3. Y consigue realizar la función de *pick and place*. Hay que destacar que, en la última captura, cuando recoge el cubo y se vuelve a orientar la muñeca, el cubo no gira solidario a él, debido a que plugin de la ventosa no presenta esa funcionalidad, pero se llega a la conclusión que en la realidad sí que lo haría.

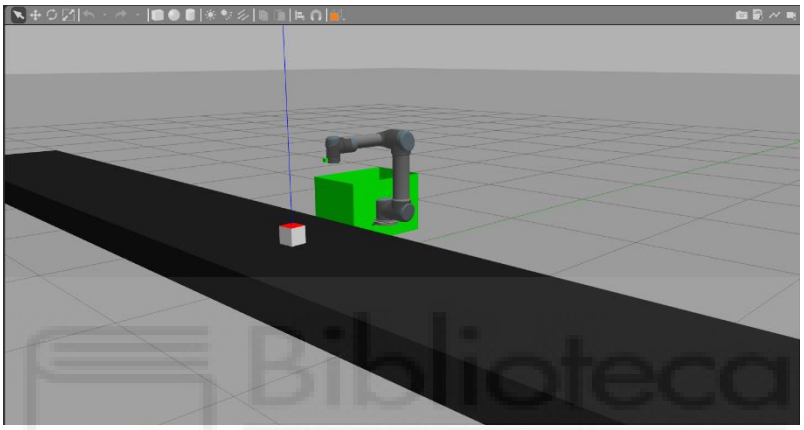


Figura 36. UR5 en la posición inicial y el cubo generado sobre la cinta

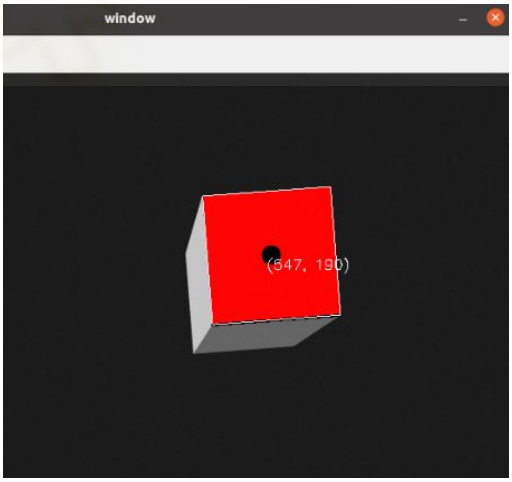


Figura 35. Visualización de la webcam sobre el objeto (cálculo del centroide)

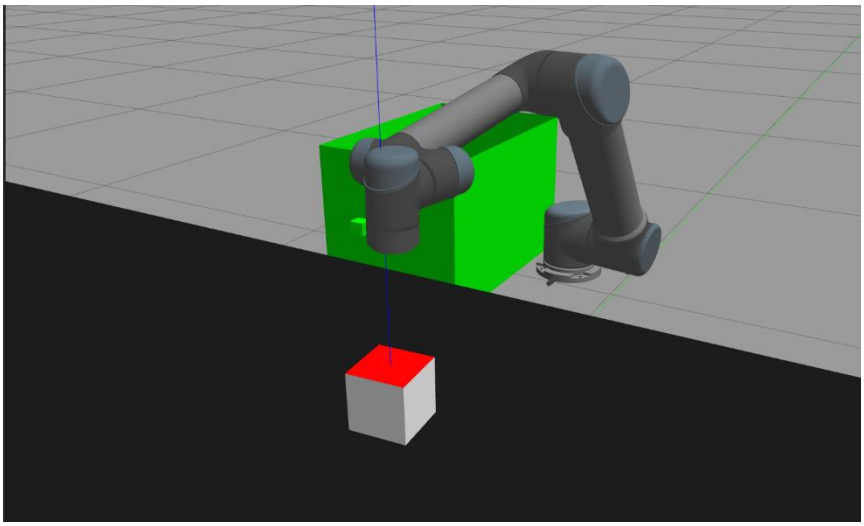


Figura 37. Movimiento del robot, acercándose al centro de la cara, mediante el cálculo de los errores y la orientación de la muñeca 3

Figura 38. Visualización de la webcam sobre el objeto, ya orientada la cámara y colocado el robot encima

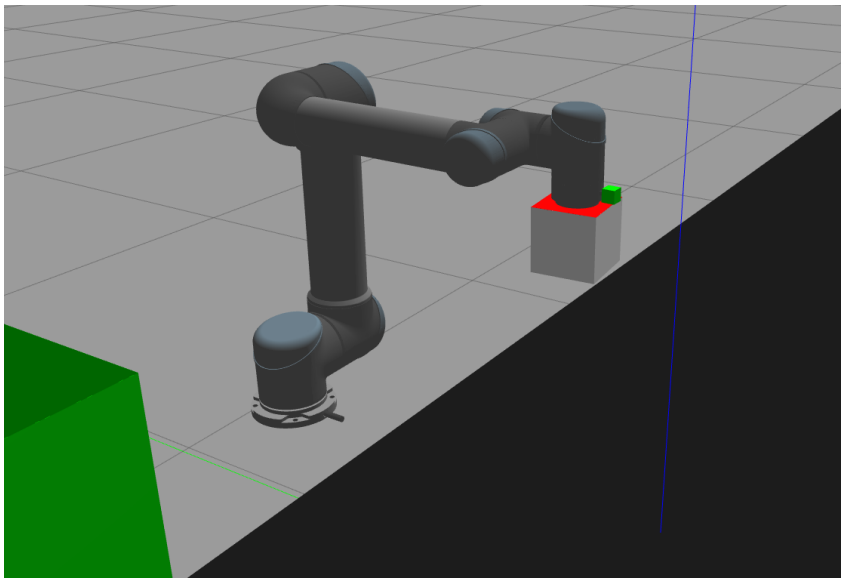


Figura 39. Posición intermedia después de coger el cubo

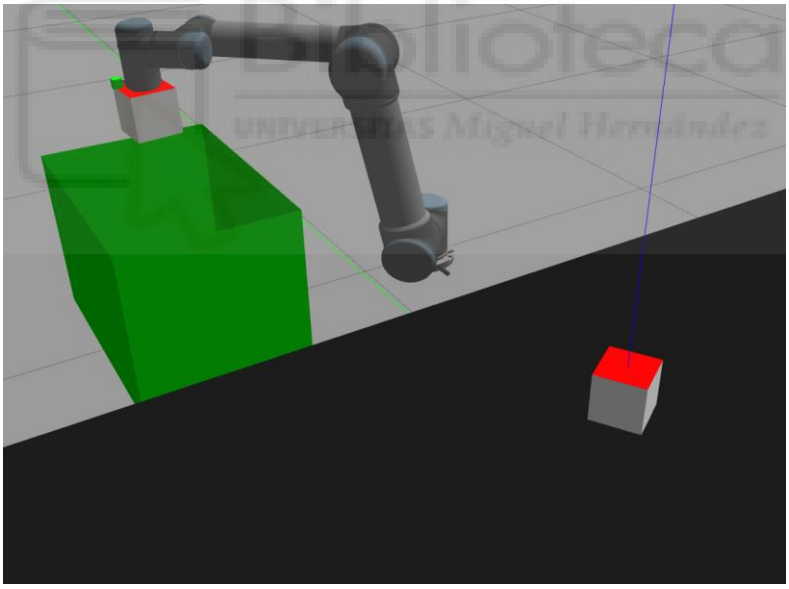


Figura 40. Posición del brazo para dejar el cubo y la generación de un cubo nuevo

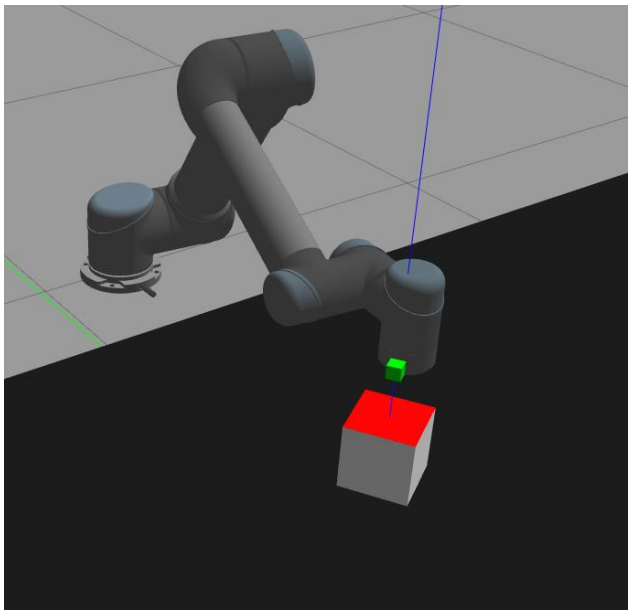


Figura 41. Nuevo cálculo del movimiento y orientación

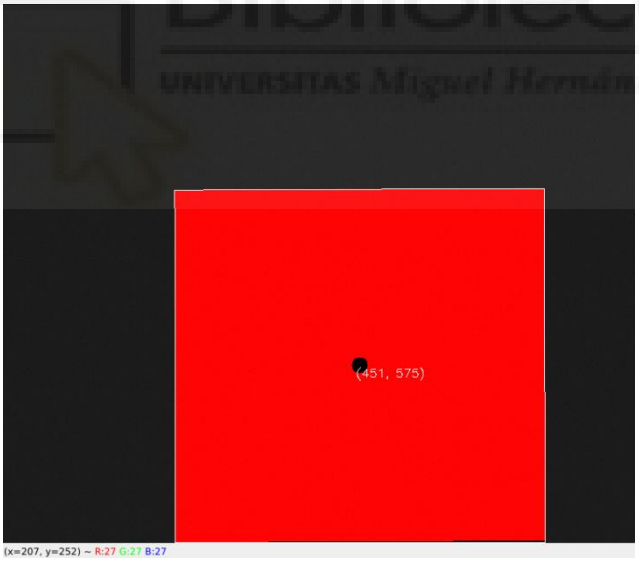


Figura 42. Comprobación de la correcta orientación

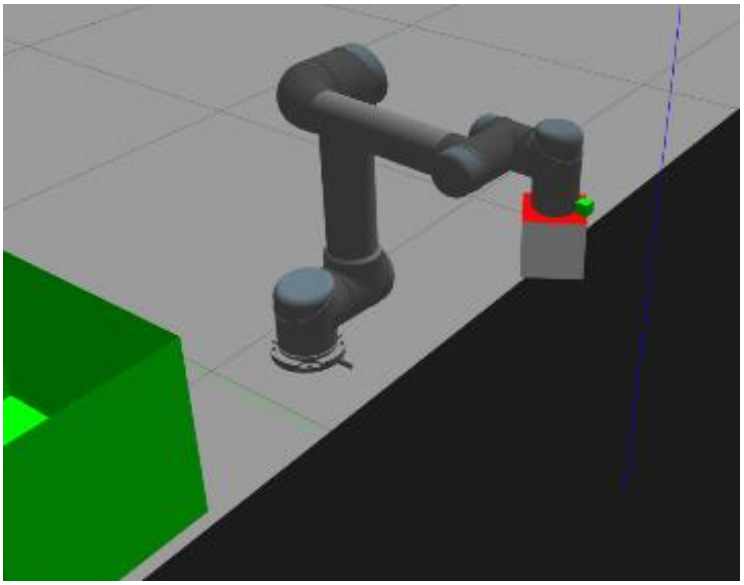


Figura 43. Muestra de que no gira el cubo con la muñeca 3

Las imágenes que se muestran ahora corresponden a la segunda aplicación, donde se puede comprobar que funciona el código perfectamente cuando el robot se acerca al cubo y se orienta para cuadrarse y poder coger bien el cubo, pero cuando se activa la ventosa, el cubo acelera y choca con el robot, hasta que se coloca encima de él, como si el plugin de la ventosa funcionase solo verticalmente. Se ha añadido estas capturas para demostrar el trabajo realizado con los inconvenientes y errores que presentan, al igual del correcto funcionamiento cuando se realiza la primera aplicación.

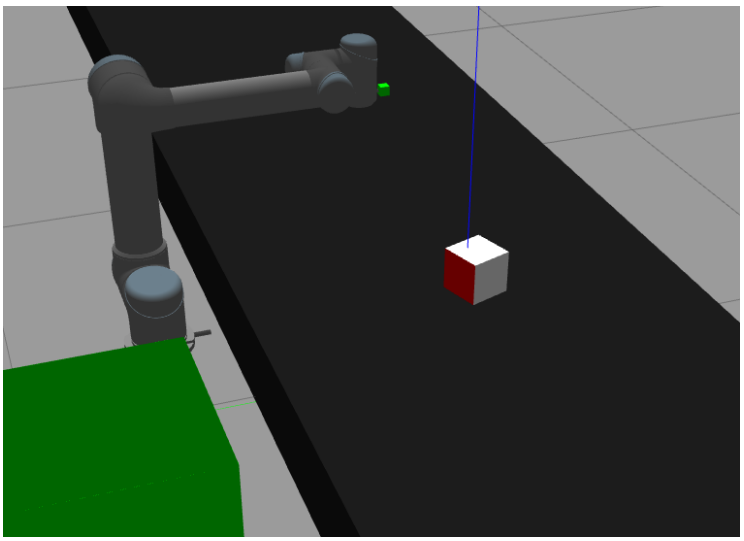


Figura 44. Inicio de la segunda aplicación

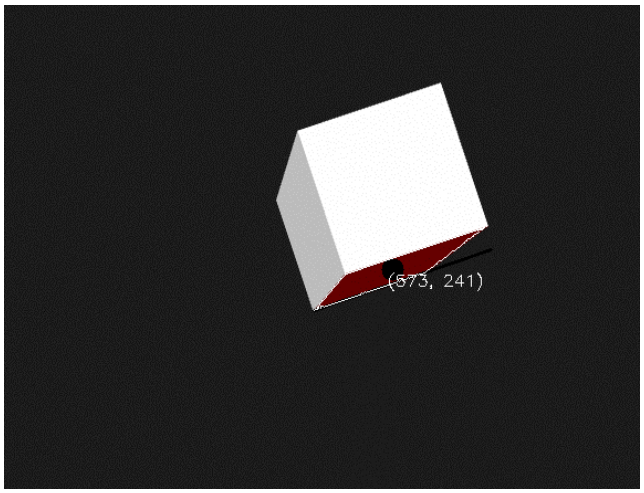


Figura 47. Visualización de la webcam sobre el cubo

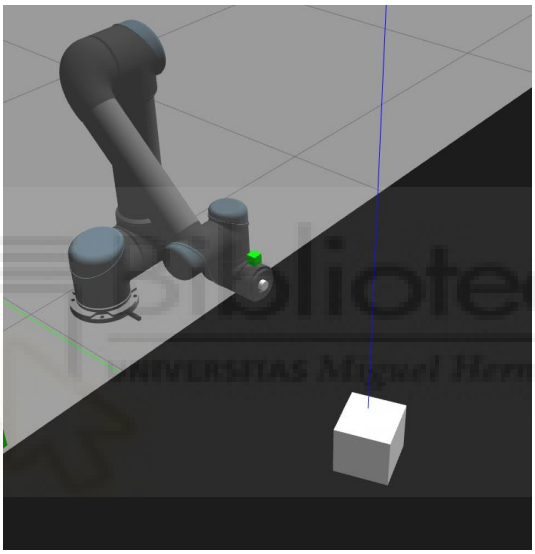


Figura 46. Cambio de posición inicial para un mejor ajuste

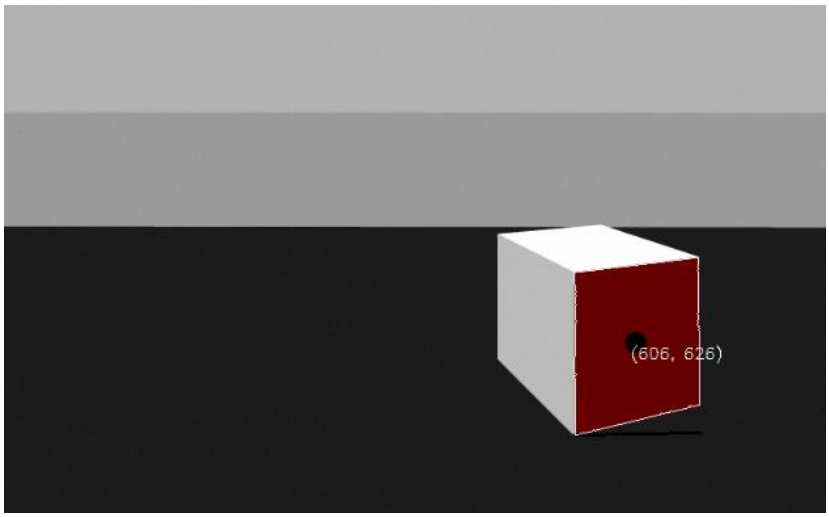


Figura 45. Nueva visualización de la webcam sobre el cubo

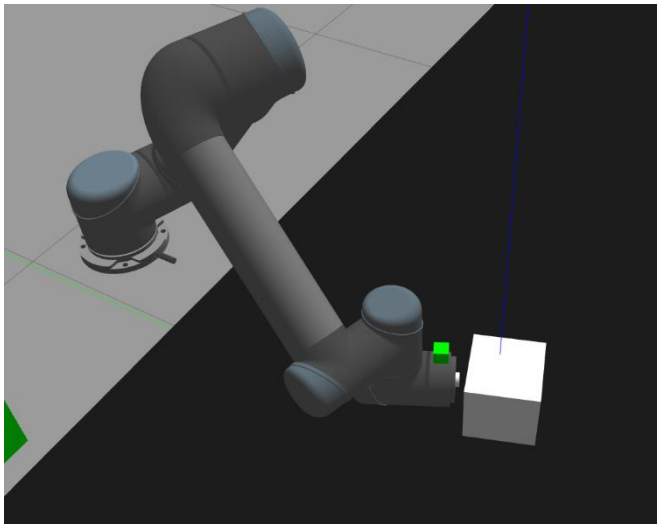


Figura 49. Cálculo del movimiento y orientación, para centrase al centroide de la cara

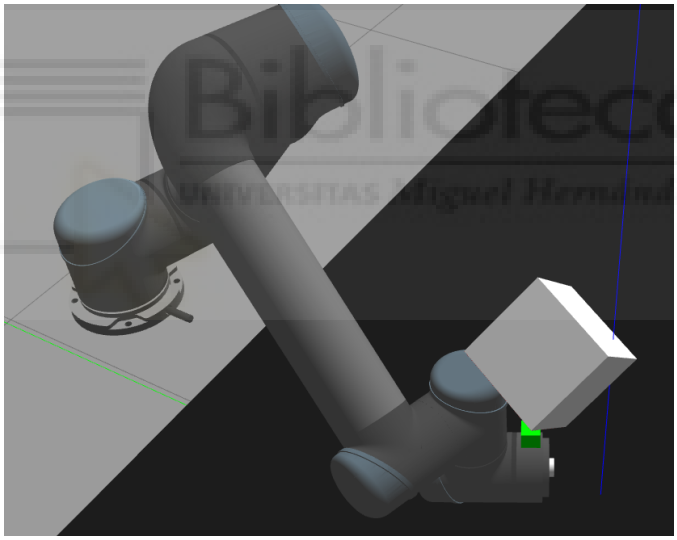


Figura 48. Error que se produce al activarse la ventosa, el cubo se mueve arriba

CAPÍTULO 8: CONCLUSIONES

8.1 EVALUACIÓN

Tras desarrollar el presente TFG empleando ROS y Visión por computador para el posicionamiento de un brazo robótico UR5 se ha llegado a las siguientes conclusiones:

- La más importante de las conclusiones es que se ha cumplido el objetivo principal que se había marcado para el Trabajo Final de Grado.

  Para conseguirlo, se han ido cumpliendo todos los pasos correctamente, comenzando por una correcta instalación del sistema operativo e instalación y configuración de todos los programas empleados en el trabajo. Entre esos pasos, hay que destacar la implementación de la webcam y de la ventosa al vacío en el extremo del brazo, seguidamente de poder generar una imagen y la activación de la ventosa para poder coger los objetos. A continuación, la creación de un entorno, asemejándose a una cinta transportadora con el objeto detenido, para simular que ha llegado a la posición indicada y la cinta se ha detenido, y una caja para colocar los objetos. Y, por último, pero es lo más importante a destacar, se escribe una serie de programas en Python, que permiten la detección del objeto, el movimiento del robot, y la activación y desactivación de la ventosa.
- Se ha comprendido y utilizado correctamente el entorno de Ubuntu, ROS y las distintas herramientas que este software ofrece.
- El meta-sistema operativo ROS ofrece un mundo lleno de posibilidades para implementar aplicaciones en robótica. Haciendo énfasis que su curva de aprendizaje es lenta y compleja, cuando se consigue llegar a un conocimiento suficiente sobre su utilización, te das cuenta de que es un software que te permite implementar aplicaciones con un enorme grado de abstracción respecto al código que las lleva a cabo.
- Gazebo es una herramienta muy útil, ya que, te permite simular prácticamente cualquier robot del mercado y poder observar su cinemática. En cuanto a su curva de aprendizaje, comparando con ROS, es bastante sencilla y permite al usuario realizar simulaciones de sistemas robóticos, a los que se pueden implementar sensores, láser, IMU, etc. Además de contar con *plugins* que permiten el movimiento de los robots.

8.2 DIFICULTADES ENCONTRADAS

Es interesante reflexionar sobre las distintas dificultades que se han encontrado a lo largo de la realización del trabajo:

- La familiarización con el entorno de Ubuntu, siendo la primera vez que se trabaja con él, ya que, resulta complejo y puede consumir mucho tiempo encontrar los elementos y el aprendizaje del uso de la terminal con todos sus comandos.
- El aprendizaje de ROS y su entorno resulta ser muy complicado si no se tiene experiencia alguna. Los tutoriales, que no son pocos, aportan información de los distintos conceptos, pero no están tan completos como debieran, y para comprender todo lo relacionado se debe consultar páginas externas e incluso libros.
- Aprender otro lenguaje de programación, URDF, aparte de los lenguajes de programación de C++ y Python. Supone un retraso para poder entender cómo funciona y cómo se estructura.

8.3 TRABAJOS FUTUROS

Por último, es posible realizar trabajos adicionales que aumenten las funcionalidades de la aplicación realizada, como, por ejemplo, si el objeto no se detuviera, mejorar el programa para que se detecte el objeto y la cara correspondiente, y durante el movimiento del objeto se calcule su orientación y que el brazo se colocase correctamente mientras realiza un seguimiento del objeto, hasta poder cogerlo. También se le podría añadir la mejora de que detectase distintos objetos y pueda clasificarlos, aparte de todo el código de posicionamiento y orientación del robot.

Otra posibilidad es realizar un estudio de los distintos planificadores y los distintos métodos de resolución de cinemática, para observar sus diferencias a la hora de realizar el mismo camino, como, por ejemplo, el uso de los planificadores que se encuentran en *MoveIt!*, que son los de tipo CHOMP o STOMP, en lugar de utilizar OMPL.

Asimismo, otra posible mejora es poder integrar la aplicación en el modelo del robot real y analizar cómo funciona el brazo robótico controlado por ROS.

CAPÍTULO 9: BIBLIOGRAFÍA

[1] Telefónica.com, "Tipo de robots: clasificación, aplicaciones y ejemplos," [En línea]. Disponible en: https://www.telefonica.com/es/sala-comunicacion/blog/tipos-de-robots-clasificacion-aplicaciones-y-ejemplos/.

[2] Clasificaciode.org, "Clasificación de Robots," [En línea]. Disponible en: https://www.clasificacionde.org/clasificacion-de-robots/.

[3] Esneca.com, "Clasificación de los robots según su función,". Disponible en: https://www.esneca.com/blog/clasificacion-de-los-robots-segun-su-funcion/.

[4] Tecnología-tecnica.com, "Morfología Básica de un Robot Industrial," [En línea]. Disponible en: https://www.tecnologia-tecnica.com.ar/pdfrobotica/PDFmorfologiaderobotindustrial.pdf.

[5] Neobotik,com, "Robots Colaborativos," [En línea]. Disponible en: https://www.neobotik.com/robots-colaborativos/#:~:text=%C2%BFQu%C3%A9%20son%20los%20robots%20colaborativos%3F%20Los%20robots%20colaborativos,de%20un%20entorno%20laboral%2C%20de%20ah%C3%AD%20su%20nombre.

[6] Universal-robots.com, "ur5-robot," [En línea]. Disponible en: https://www.universal-robots.com/products/ur5-robot/.

[7] Universal-robots.com, "Universal Robots UR5: características y aplicaciones," [En línea]. Disponible en: https://www.universal-robots.com/es/blog/universal-robots-ur5/.

[8] IBM.com, "Middleware," [En línea]. Disponible en: https://www.ibm.com/es-es/topics/middleware.

[9] Macanás Valera, Joaquín. "Interacción entre webcam y brazo robot para el posicionamiento final del efector final," Trabajo Final de Grado, Universidad Politécnica de Cartagena, 2014.

[10] Orocos.org, "Website," [En línea]. Disponible en: https://orocos.org/node/26.

[11] Marina Valles, Jose I. Cazalilla, Angel Valera, Vicente Mata y Alvaro Page, "Implementación basada en el middleware OROCOS de controladores dinámicos pasivos para un robot paralelo," Revista Iberoamericana de Automática e Informática industrial 10, 2013. [En línea]. Disponible en: www.siencedirect.com.

[12] ROS Wiki, "Introduction," [En línea]. Disponible en: http://wiki.ros.org/ROS/Introduction.

[13] YoonSeok Pyo, HanCheol Cho, RyuWoon Jung y TaeHoon Lim, "ROS Robot Programming," #1505, 145, Gasan Digital 1-ro, GeumCheon-gu, Seoul, Republic of Korea, ROBOTIS Co.,Ltd., Dec 22, 2017.

[14] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler y Andrew Ng, "ROS: an open-source Robot Operating System," [En línea]. Disponible en: http://www.robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf.

[15] Gracia Ruiz, Inazio. "Manipulación robótica colaborativa mediante el Sistema Operativo Robótico," Trabajo Final de Grado, Universidad de Zaragoza, 2019.

[16] ROS Wiki, "Concepts," [En línea]. Disponible en: http://wiki.ros.org/ROS/Concepts.

[17] Gazebo.org, "home," [En línea]. Disponible en: https://gazebosim.org/home.

[18] Gazebo.org, "features," [En línea]. Disponible en: https://gazebosim.org/features.

[19] Programadorclick.com, "Qué es URDF," [En línea]. Disponible en: https://programmerclick.com/article/29081682599/.

[20] Robótica fácil con ros2, "Todo lo que necesitas saber de los archivos URDF," [En línea]. Disponible en: https://robotica-facil-con-ros2.es/?p=1439#Que_son_y_para_que_sirven.

[21] ROS Wiki, "moveit," [En línea]. Disponible en: http://wiki.ros.org/moveit

[22] Moveit.ros.org, "Concepts," [En línea]. Disponible en: https://moveit.ros.org/documentation/concepts/.

[23] ROS planning, "moveit_tutorials," [En línea]. Disponible en: https://ros-planning.github.io/moveit_tutorials/.

[24] Moveit.ros.org, "Plugins," [En línea]. Disponible en: https://moveit.ros.org/documentation/plugins/.

[25] Mineduc.gob.gt, "Cinemática" [En línea]. Disponible en: https://www.mineduc.gob.gt/DIGECADE/documents/Telesecundaria/Recursos%20Digitales/1o%20Recursos%20Digitales%20TS%20licencia%20CC%20BY-SA%203.0/01%20CIENCIAS%20NATURALES/U9%20pp%20191%20Cinem%C3%A1tica.pdf.

[26] Opencv.org, "opencv," [En línea]. Disponible en: https://opencv.org/.

[27] ROS Wiki, "visión_opencv," [En línea]. Disponible en: http://wiki.ros.org/vision_opencv.

[28] Github.com, "universal_robot," [En línea]. Disponible en: https://github.com/ros-industrial/universal_robot/tree/kinetic-devel.

[29] Github.com, "moveit_tutorials," [En línea]. Disponible en: https://github.com/ros-planning/moveit_tutorials.

[30] Github.com, "gazebo_ros_pkgs," [En línea]. Disponible en: https://github.com/ros-simulation/gazebo_ros_pkgs.

[31] Github.com, "ur5_ROS-Gazebo," [En línea]. Disponible en: https://github.com/lihuang3/ur5_ROS-Gazebo.

[32] Universal robots.com, "Ficha técnica," [En línea]. Disponible en: https://www.universal-robots.com/media/1801300/esp_semea_199912_ur5_tech_spec_web_a4.pdf.

[33] Universal Robots, "Manual de usuario," [En línea]. Disponible en: https://www.cfzcobots.com/wp-content/uploads/2017/03/ur5_user_manual_es_global.pdf.

[34] Keyence.com, "PL (Performance Level)," [En línea]. Disponible en: https://www.keyence.com/ss/products/safetyknowledge/performance/level/.

[35] Tameson.es, "Tabla de clasificación ip y definiciones," [En línea]. Disponible en: https://tameson.es/pages/tabla-de-clasificacion-ip-y-definiciones#:~:text=La%20clasificaci%C3%B3n%20IP%20(tambi%C3%A9n%20conocida,humedad%2C%20los%20l%C3%ADquidos%2C%20etc.

[36] Labson.es, "Clasificación de salas blancas," [En línea]. Disponible en: https://labsom.es/blog/clasificacion-de-salas-blancas-segun-el-iso-14644/.

[37] Releases.Ubuntu.com, "jammy," [En línea]. Disponible en: https://releases.ubuntu.com/jammy/.

[38] Virtualbox.org, "Downloads," [En línea]. Disponible en: https://www.virtualbox.org/wiki/Downloads.

[39] ROS Wiki, "noetic," [En línea]. Disponible en: http://wiki.ros.org/noetic.

[40] ROS Wiki, "Installation," [En línea]. Disponible en: http://wiki.ros.org/noetic/Installation/Ubuntu.

[41] ROS Wiki, "Tutorials," [En línea]. Disponible en: http://wiki.ros.org/ROS/Tutorials.

[42] ROS Wiki, "Installing and Configuring Your ROS Environment," [En línea]. Disponible en: http://wiki.ros.org/ROS/Tutorials/InstallingandConfiguringROSEnvironment.

[43] ROS Wiki, "Navigating the ROS Filesystem," [En línea]. Disponible en: http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem.

[44] ROS Wiki, "Creating a ROS Package," [En línea]. Disponible en: http://wiki.ros.org/ROS/Tutorials/CreatingPackage.

[45] ROS Wiki, "Understanding ROS Nodes," [En línea]. Disponible en: http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes.

[46] ROS Wiki, "Understanding ROS Topics," [En línea]. Disponible en: http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics.

[47] ROS Wiki, "Understanding ROS Services and Parameters," [En línea]. Disponible en: http://wiki.ros.org/ROS/Tutorials/UnderstandingServicesParams.

[48] ROS Wiki, "Using rqt_console and roslaunch," [En línea]. Disponible en: http://wiki.ros.org/ROS/Tutorials/UsingRqtconsoleRoslaunch.

[49] ROS Wiki, "XML," [En línea]. Disponible en: http://wiki.ros.org/roslaunch/XML.

CAPÍTULO 10: ANEXOS

ANEXO 1: Características Técnicas del robot colaborativo UR5

En este primer Anexo se describen los datos técnicos más importantes del modelo UR5, fabricado por Universal Robots.

En la figura siguiente se ha realizado un despiece del brazo, comentado todos eslabones y articulaciones que posee.

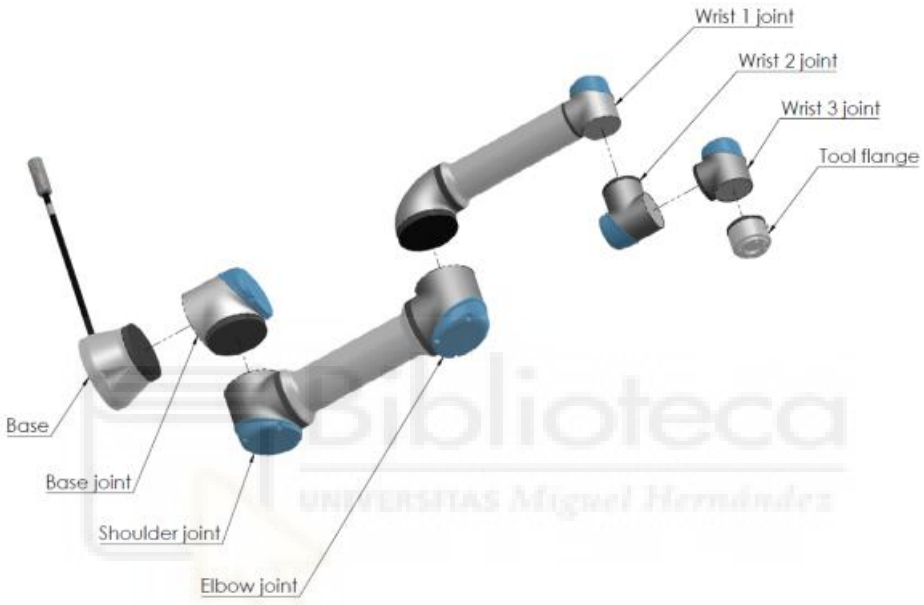


Figura 50. Despiece del robot UR5

El espacio de trabajo del robot UR5 ocupa 850 mm desde la junta de la base. Es importante tener en cuenta el volumen cilíndrico debajo y encima de la base del robot, a la hora de elegir el lugar de instalación. Se debe evitar mover la herramienta cerca de ese

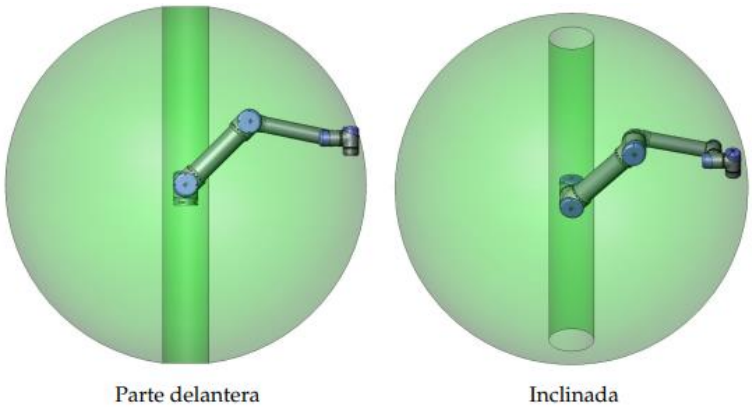


Figura 51. Espacio de trabajo del UR5

volumen, porque provoca que las juntas se muevan rápido y hace que el robot se mueva de forma ineficiente y que la realización de la evaluación de riesgos sea difícil.

Primero se comenta todas las características que presenta el brazo sin profundizar en su caja de control y consola de programación, es decir, el rendimiento, especificaciones, movimiento, funciones y características físicas:

- Rendimiento:

Rendimiento	
Repetibilidad	±0,1 mm / ±0,0039 in (4 mil.)
Rango de temperatura ambiente	0-50°
Consumo de energía	Mín. 90 W, estándar 150 W, máx. 325 W
Operación de colaboración	15 funciones avanzadas de seguridad regulables. Función de seguridad con certificación TÜV NORD Probado de acuerdo con las normas: EN ISO 13849:2008 PL d

Figura 52. Rendimiento del UR5

Hay que destacar que dos aspectos importantes. El primero es que la norma ISO que cumple el robot, dentro de la operación de colaboración, significa que las partes y elementos del robot que deben ser capaces de actuar en caso de emergencia lo hacen bajo unas directrices predecibles, pero este estándar requerido ha de ser igual o superado. En el caso del UR5, se observa que su nivel es de PL d.

Nivel de rendimiento (PL)	Probabilidad de fallo peligroso por hora (PFHd) 1/h
un	$\geq 10^{-5}$ y $< 10^{-4}$ ⟨0.001% a 0.01%⟩
b	$\geq 3 \times 10^{-6}$ y $< 10^{-5}$ ⟨0.0003% a 0.001%⟩
c	$\geq 10^{-6}$ y $< 3 \times 10^{-6}$ ⟨0.0001% a 0.0003%⟩
d	$\geq 10^{-7}$ y $< 10^{-6}$ ⟨0.00001% a 0.0001%⟩
e	$\geq 10^{-8}$ y $< 10^{-7}$ ⟨0.000001% a 0.00001%⟩

Figura 53. Tabla con los índices de probabilidad de fallo

El segundo aspecto es que en el caso de que las articulaciones estén en uso continuado a elevadas velocidades, la temperatura de trabajo del robot se reduce.

- Especificaciones:

Especificación	
Carga útil	5 kg / 11 lb
Alcance	850 mm / 33,5 in
Grados de libertad	6 articulaciones giratorias
Programación	Interfaz gráfica del usuario PolyScope con pantalla táctil de 12" in con soporte

Figura 54. Especificaciones del UR5

Hay que considerar que, dentro de la carga útil, el robot no cuenta con ningún elemento terminal ni herramienta. Por lo tanto, si el diseño de la aplicación consiste en el desplazamiento de elementos dentro de un espacio, se ha de considerar tanto el peso del objeto como la herramienta o elemento terminal que utilice.

En cuanto a la programación, se comenta en el apartado correspondiente a la consola de programación.

- Movimiento:

Movimiento		
Movim. del eje del brazo robot.	Radio de acción	Velocidad máxima
Base	± 360°	± 180°/s
Hombro	± 360°	± 180°/s
Codo	± 360°	± 180°/s
Muñeca 1	± 360°	± 180°/s
Muñeca 2	± 360°	± 180°/s
Muñeca 3	± 360°	± 180°/s
Herramienta típica		1 m/s / 39,4 in/s

Figura 55. Movimiento del UR5

En el caso de la muñeca 3, el radio de acción no es infinita, está limitada entre el rango +/- 360º, y condiciona las operaciones como atornillado o taladro. Para ello, se añade ese último eslabón, que corresponde al típico TCP o elemento terminal, el cual, no se considera radio de acción.

- Funciones:

Funciones		
Clasificación IP	IP54	
Clase ISO Sala limpia	5	
Ruido	72dB	
Montaje del robot	Todos	
Puertos de E/S en herramienta	Entrada digital	2
	Salida digital	2
	Entrada analógica	2
	Salida analógica	0
E/S de fuente de aliment. en herramienta	12 V / 24 V 600 mA en herramienta	

Figura 56. Funciones del UR5

Los grados IP (Ingress Protection) se define en la norma internacional EN 60529 y para el UR5 es 54, lo que significa que el robot tiene protección contra las partículas de polvo, pero no es totalmente estanco, según el primer dígito, y que presenta una protección contra las salpicaduras de agua desde cualquier dirección, según el segundo dígito.

Además, el robot debe estar situado en una sala limpia de clase 5 según la norma ISO 14644 1, cuyas características son las siguientes:

ISO 14644-1 Standard de salas limpias

Clase	Máximo de partículas /m³					
	>=0.1 µm	>=0.2 µm	>=0.3 µm	>=0.5 µm	>=1 µm	>=5 µm
ISO 1	10	2				
ISO 2	100	24	10	4		
ISO 3	1,000	237	102	35	8	
ISO 4	10,000	2,370	1,020	352	83	
ISO 5	100,000	23,700	10,200	3,520	832	29
ISO 6	1,000,000	237,000	102,000	35,200	8,320	293
ISO 7				352,000	83,200	2,930
ISO 8				3,520,000	832,000	29,300

Figura 57. Tabla de cantidad de partículas en salas limpias

- Características físicas:

Características físicas	
Huella	Ø 149 mm
Materiales	Aluminio, plásticos de PP
Tipo de conector para herramientas	M8
Long. cable del brazo robótico	6 m / 236 in
Peso con cable	18,4 kg / 40,6 lb

Figura 58. Características físicas del UR5

El robot es muy práctico gracias a que el espacio necesario para montarlo no supera el diámetro de su huella, que es de 149 mm. Además, su instalación es rápida porque pesa poco (18.a kg) porque su estructura está fabricada de aluminio y plásticos, y solo necesita una toma de corriente estándar.

Seguidamente, se describen las funciones y las características físicas que tiene la caja de control del robot UR5:

- Funciones:

Funciones		
Clasificación IP	IP20	
Clase ISO Sala limpia	6	
Ruido	<65 dB (A)	
Puertos de E/S	Entrada digital	16
	Salida digital	16
	Entrada analógica	2
	Salida analógica	2
E/S de fuente de alimentación	24 V 2 A	
Comunicación	TCP/IP 100 Mbit, Modbus TCP, Profinet, EthernetIP	
Fuente de alimentación	100-240 V CA, 50-60 Hz	
Rango de temperatura ambiente	0-50°	

Figura 59. Funciones de la caja de control

Para la caja de control, su clasificación IP es de 20, es decir, muestra una protección contra partículas sólidas de 12.5 mm de diámetro o más, según el primer dígito, y no posee ninguna protección contra los líquidos ni humedad, según el segundo dígito.

Como el brazo en sí, la caja de control debe situarse en una sala limpia, pero en este caso, de clase 6, cuyas características se pueden apreciar en Figura 60.

- Características físicas:

Características físicas	
Tamaño de la caja de control (an.×al.×la.)	475 × 423 × 268 mm / 18,7 × 16,7 × 10,6 in
Peso	15 kg / 33,1 lb
Materiales	Acero

Figura 60. Características físicas de la caja de control

Por último, los modelos de Universal Robots poseen una consola de programación que permite al usuario poner a punto y programar trayectorias al brazo, y el robot UR5 no es una excepción. Se muestran las características de la consola:

- Funciones y especificaciones físicas:

Funciones	
Clasificación IP	IP20

Características físicas	
Materiales	Aluminio, PP
Peso	1,5 kg
Longitud del cable	4,5 m / 177 in

Figura 61. Funciones de la consola

Muestra la misma clasificación IP que la caja de control.

La consola es una pantalla táctil de 12'' in con soporte configurada con el software Polyscope, que permite al usuario generar movimientos en el UR5 mediante el control manual o cargar scripts directamente en el robot con el propio lenguaje de Universal Robot, URScript.

ANEXO 2: Instalación y Configuración de Linux

Este anexo recoge los pasos a seguir para instalar de forma satisfactoria Linux, más concretamente Ubuntu, en un PC junto con Windows por medio de un disco duro externo.

Para seguir correctamente este anexo, el usuario debe poseer un disco externo, para el trabajo, se ha utilizado un disco duro externo SSD con 500 Gb de almacenamiento y con conexión USB 3.0.

Descarga de VirtualBox

Para la instalación, lo primero que se necesita es descargarse VirtualBox porque es el medio que se utiliza para proceder con la instalación de Ubuntu en el disco duro. Se dirigen al siguiente enlace:

https://www.virtualbox.org/wiki/Downloads

Donde se descargan VirtualBox, dependiendo del sistema operativo donde quieran arrancarlo, en este caso, se ha elegido Windows.

Asimismo, en la misma página, se descarga la Extension Pack, lo que permite que VirtualBox detecte la conexión del USB 3.0. En cualquier caso, si el disco duro no tiene conexión USB 3.0, no es necesario la instalación del Extension Pack, pero la instalación de Ubuntu sería más lenta.

Cuando se hayan descargado la máquina virtual y la Extension Pack, el orden de instalación es primero VirtualBox y después la Extension Pack.

Obtención de la ISO

Cuando se haya descargado e instalado VirtualBox con la extensión, seguidamente, se busca la página web oficial de Ubuntu y descargar la ISO que contiene el sistema operativo. En el presente trabajo se ha descargado la versión 20.04.03 LTS (Jammy Jellyfish), porque es la que tiene soporte completo para la versión de Noetic de ROS. Por lo tanto, se dirigen al siguiente enlace:

https://releases.ubuntu.com/jammy/

Una vez se encuentren dentro de la página, se descargan la versión Imagen de escritorio, porque se busca instalar Ubuntu en el disco y arrancar el sistema operativo desde ahí.

Configuración de la Máquina Virtual

En el momento que se haya descargado e instalado todo, abrimos la aplicación de VirtualBox y creamos una nueva máquina virtual.

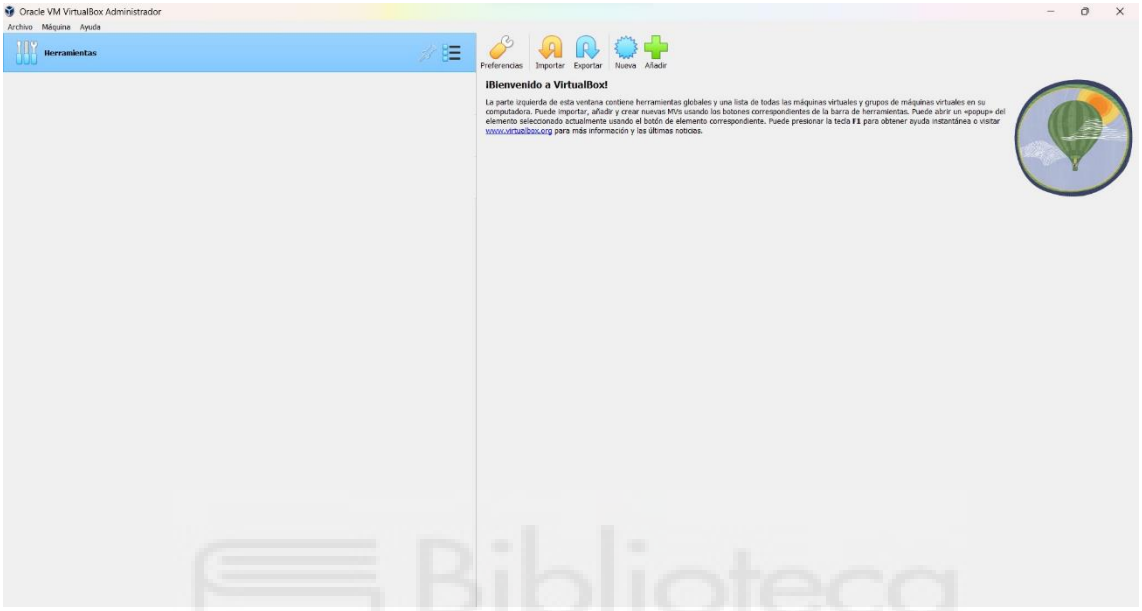


Figura 62. Página principal de VirtualBox

Al crear la máquina virtual, lo primero es ponerle nombre a la máquina, la carpeta de destino y se añade la versión ISO descargada.

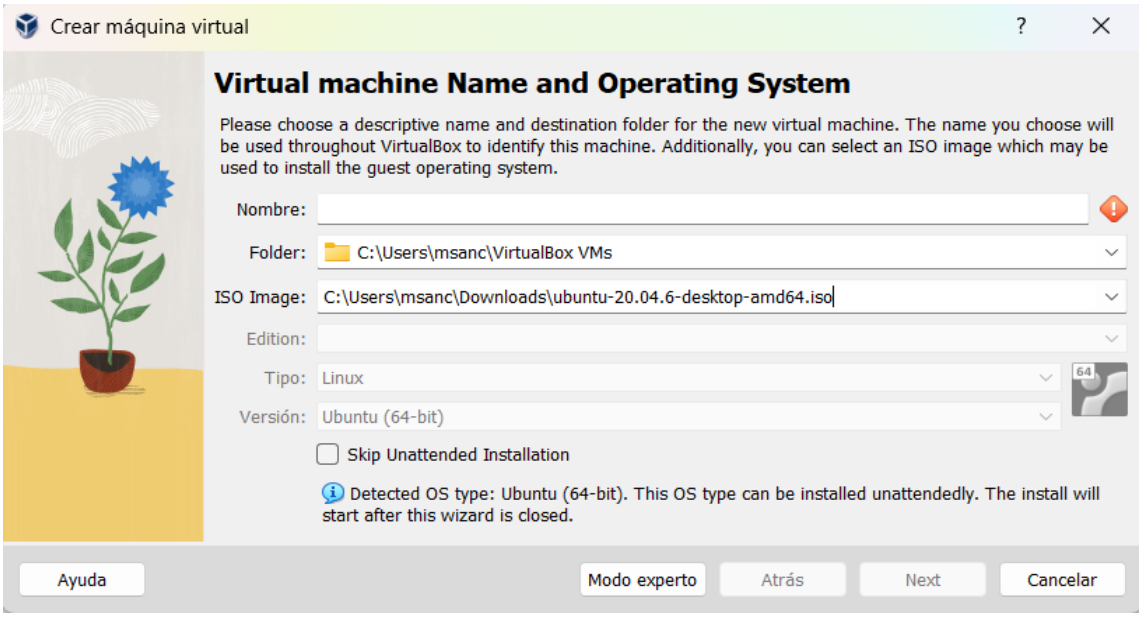


Figura 63. Primera ventana al crear una máquina virtual (nombre y sistema operativo)

Después emerge otra ventana, donde se escribe el nombre de usuario y la contraseña:

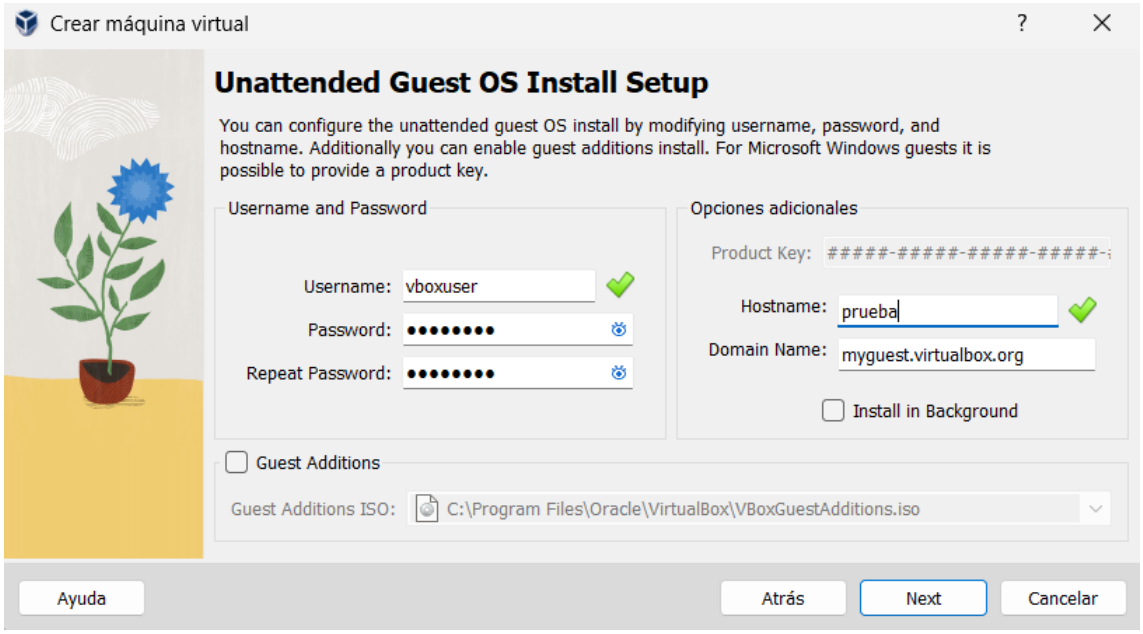


Figura 64. Ventana para poner el nombre de usuario y contraseña

Al completarlo, vuelve a surgir otra ventana. En ella, se elige la configuración del hardware, es decir, la capacidad de la memoria base y las CPUs del procesador. La elección es preferencia del usuario, pero cuanta más memoria se elija mejor para la máquina virtual. Para el trabajo se ha elegido una memoria base de 9732 MB y 3 CPUs.

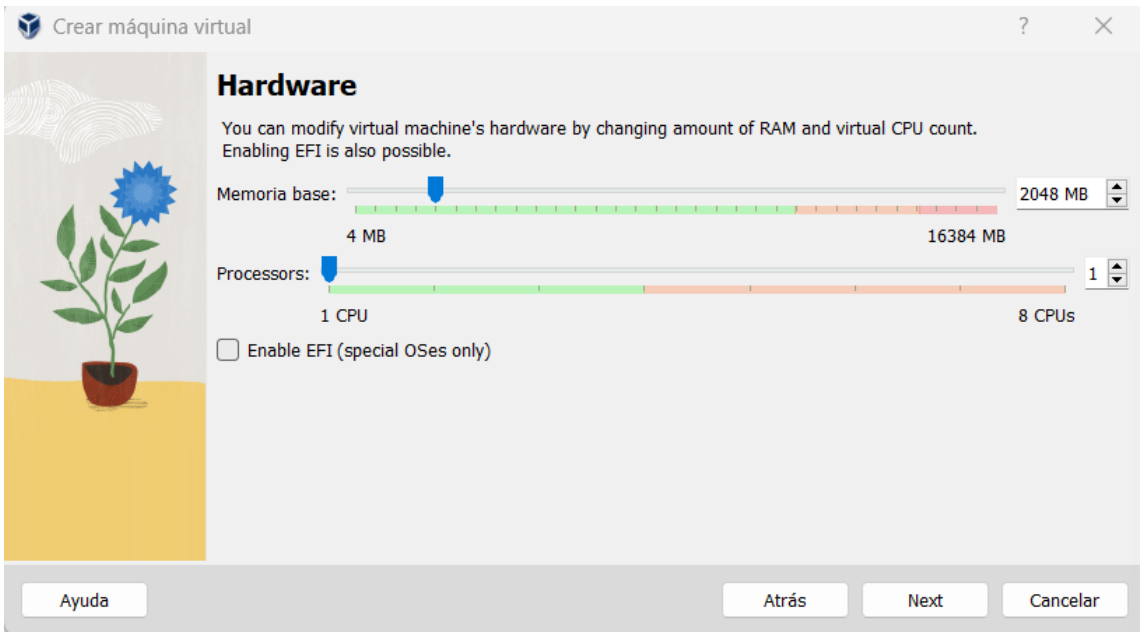


Figura 65. Ventana para elegir capacidad de máquina

A continuación, pregunta si se quiere añadir un disco duro virtual, pero al utilizar ya un disco duro externo, se selecciona no añadir un disco duro virtual:

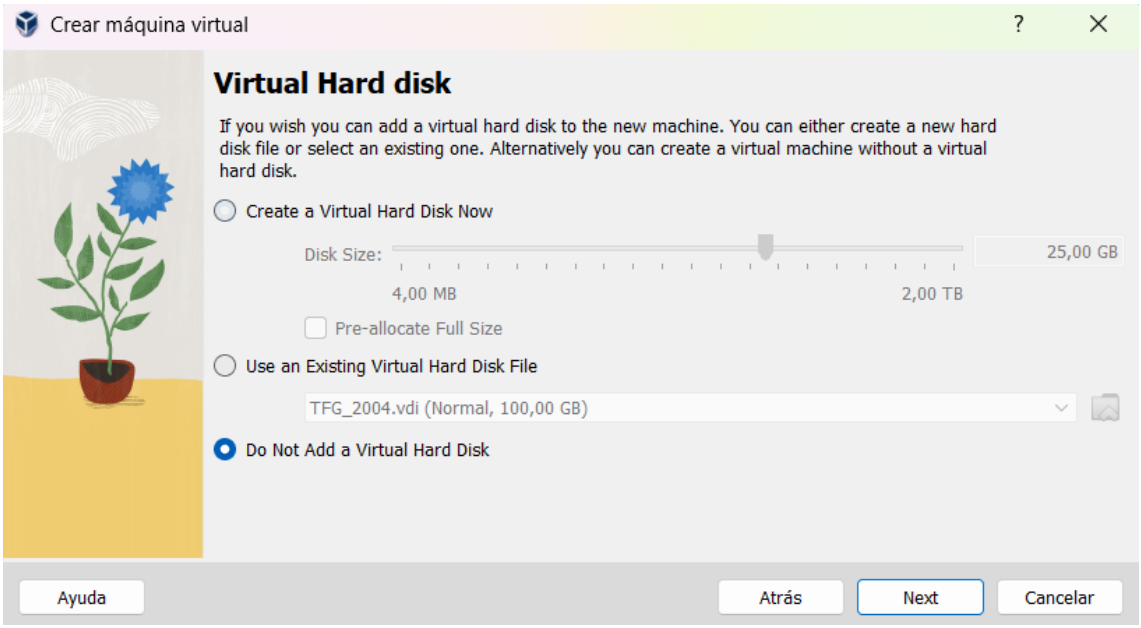


Figura 66. Ventana para elegir disco duro

Por último, se muestra un resumen de toda la configuración que se ha elegido y se termina la configuración inicial. Pero antes de continuar con la instalación de Ubuntu, para que se pueda arrancar desde el disco duro externo, entramos a la configuración, donde en el almacenamiento volvemos añadir la imagen ISO de Ubuntu en el disco vacío del controlador IDE.

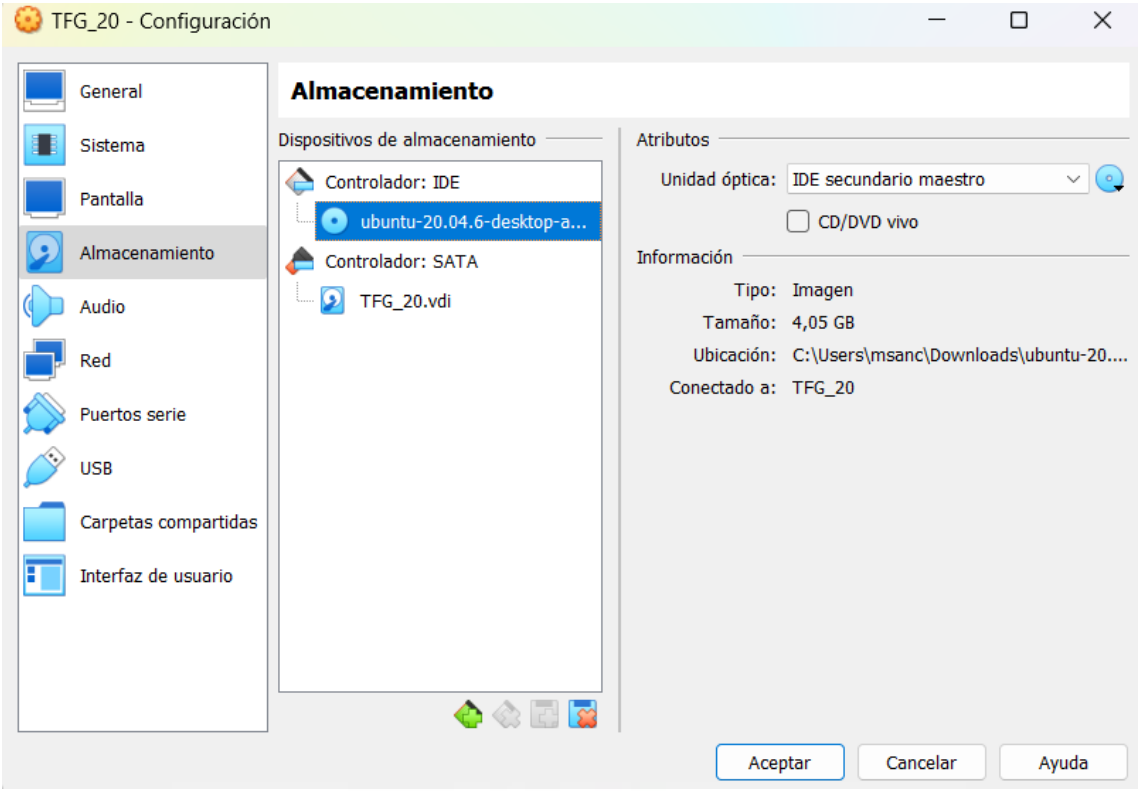


Figura 67. Ventana de configuración para añadir la ISO

Para acabar con la configuración, en el apartado de USB, se elige el controlador USB 3.0 gracias a la extensión que se ha instalado, en este momento se conecta el disco duro externo y aparecerá el nombre del dispositivo, el cual, se elegirá.

Instalar Ubuntu

Cuando ya se haya configurado todo, se procede a la instalación de Ubuntu desde la máquina virtual. Para ello, se inicia la máquina y después de cargarse completamente la iniciación volverá a aparecerse varias pestañas para configurar la instalación.

La primera ventana es para elegir el idioma que el usuario prefiera, ya que, también se aplicará al teclado, y elegir si se prefiere instalar en un CD para probar Ubuntu o instalarlo junto con el sistema operativo actual, para que la instalación funcione sin problemas, se elegirá la segunda opción:

Figura 68. Ventana para seleccionar idioma, configuración de teclado y modo de instalación

Después, se volverá a elegir el idioma, pero referido a la disposición del teclado, la decisión será tomada por el usuario.

Figura 69. Ventana para elegir disposición del teclado

Seguidamente pregunta al usuario si quiere que las aplicaciones predeterminadas y otras opciones de software se instalen, en tal caso, en la presente instalación se ha procedido a una instalación mínima y seleccionar que descargue las últimas actualizaciones e instalar programas de terceros para evitar problemas:

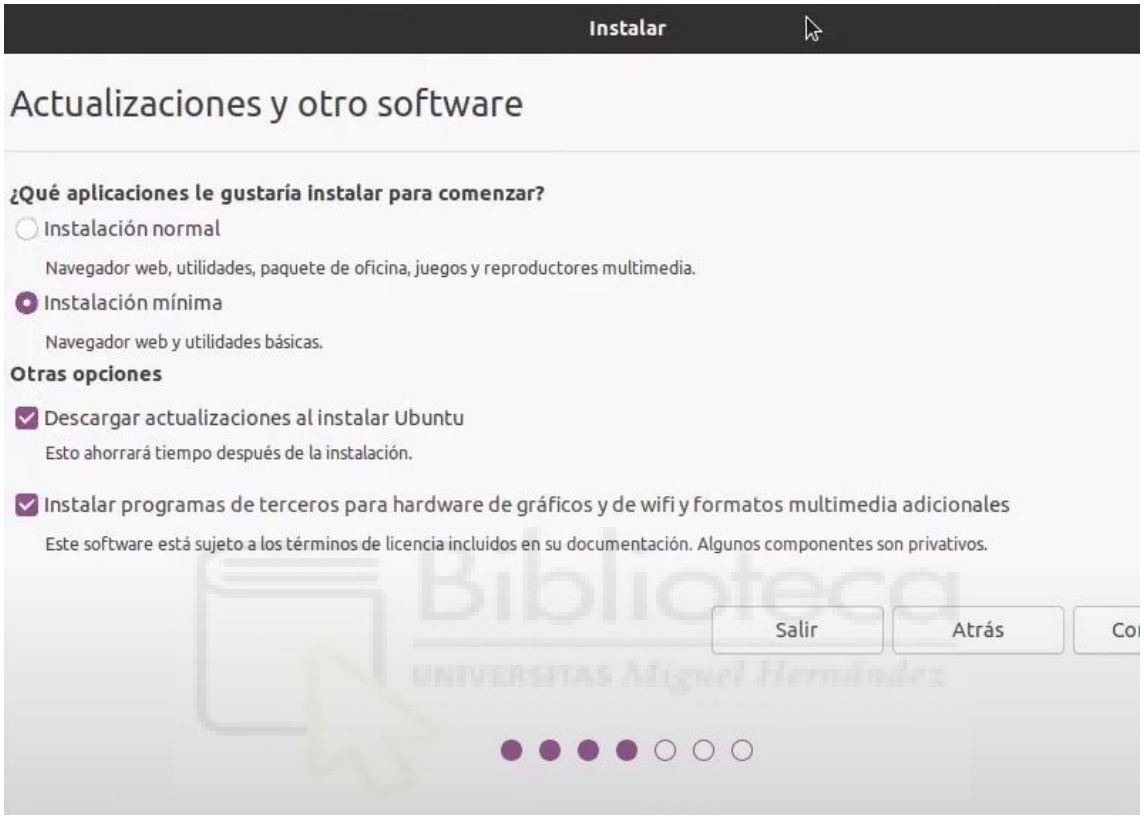


Figura 70. Ventana para elegir tipo de instalación

En el tipo de instalación que se le pregunta al usuario, se selecciona borrar el disco e instalar Ubuntu, para empezar de nuevo:

Figura 71. Ventana para elegir empezar de nuevo la instalación

Por último, en las dos últimas ventanas se escogerá el huso horario del usuario y la configuración de los datos del usuario:

Figura 73. Ventana para elegir huso horario

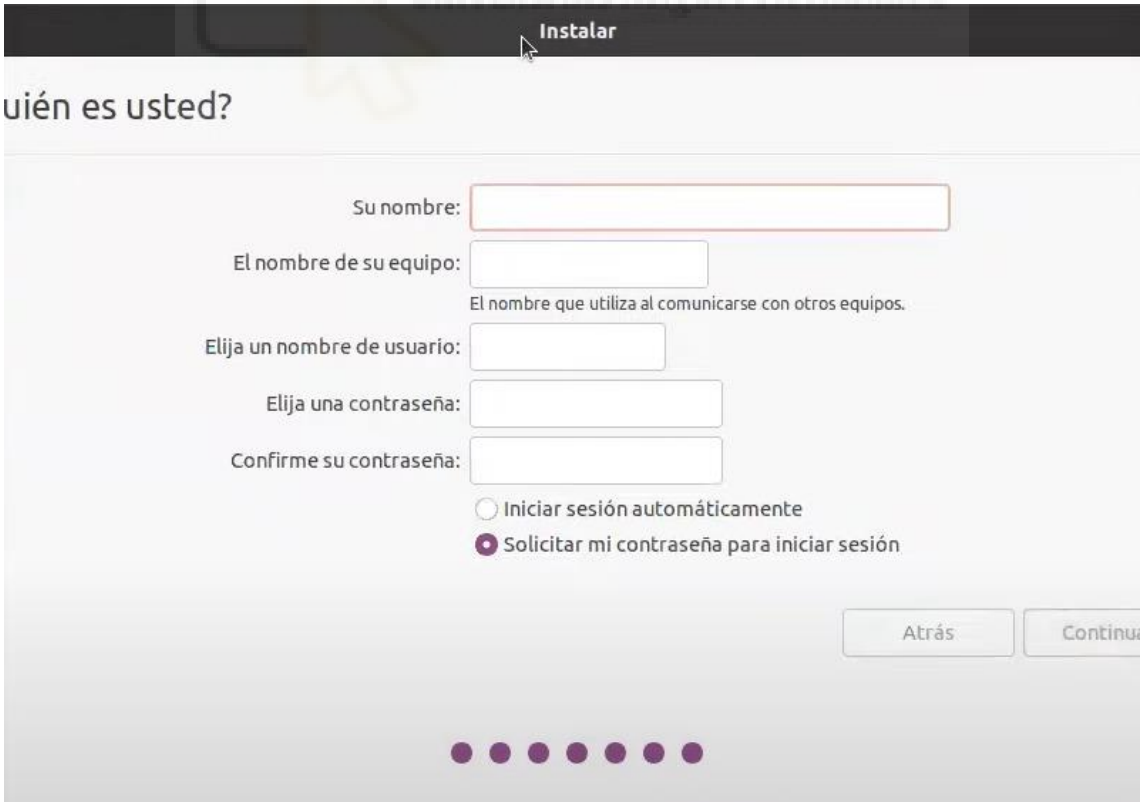


Figura 72. Última ventana para datos

Realizada la configuración, tardará unos minutos en instalarse todo, y si no ha habido problemas, saldrá la siguiente ventana diciendo que la instalación se ha completado correctamente y que debe ser reiniciado el equipo:

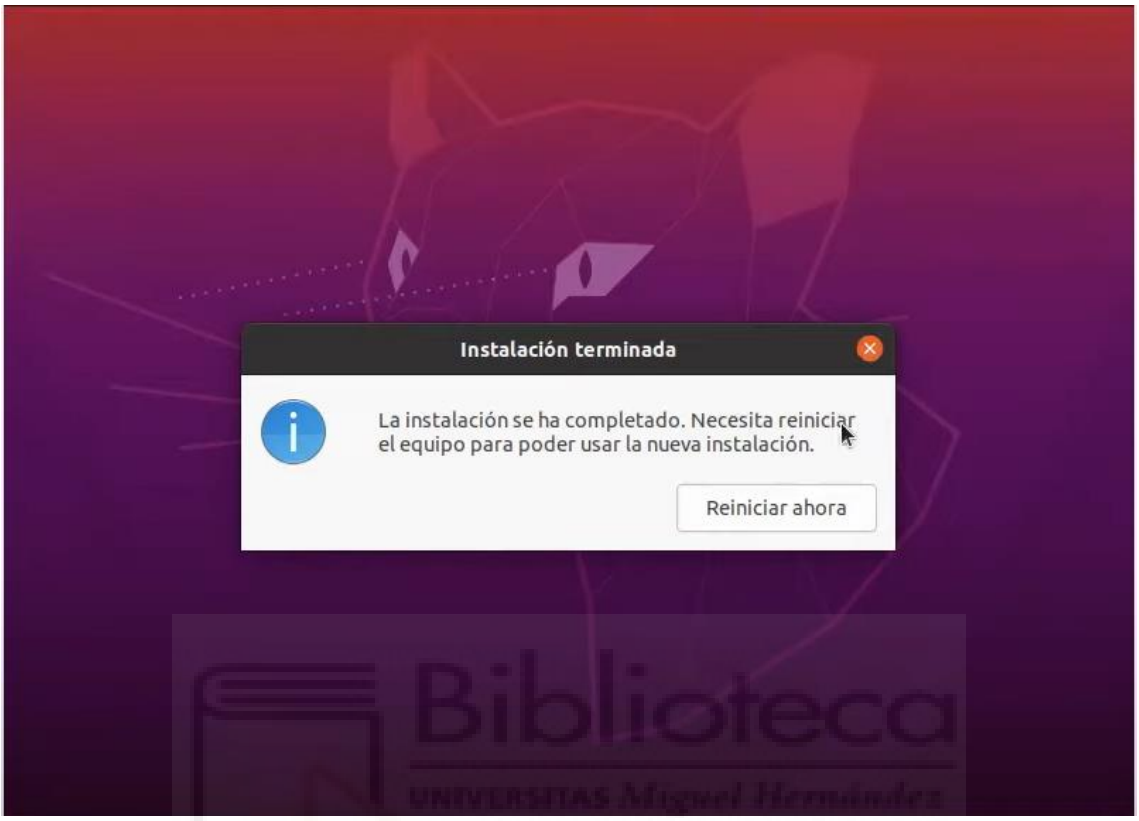


Figura 74. Confirmación de que se ha instalado correctamente

Y nos pedirá que desconectemos el disco duro del equipo. Cuando se desconecte se cerrará la máquina virtual y ya tendremos Ubuntu instalado en el disco duro externo y se podrá arrancar desde el dispositivo.

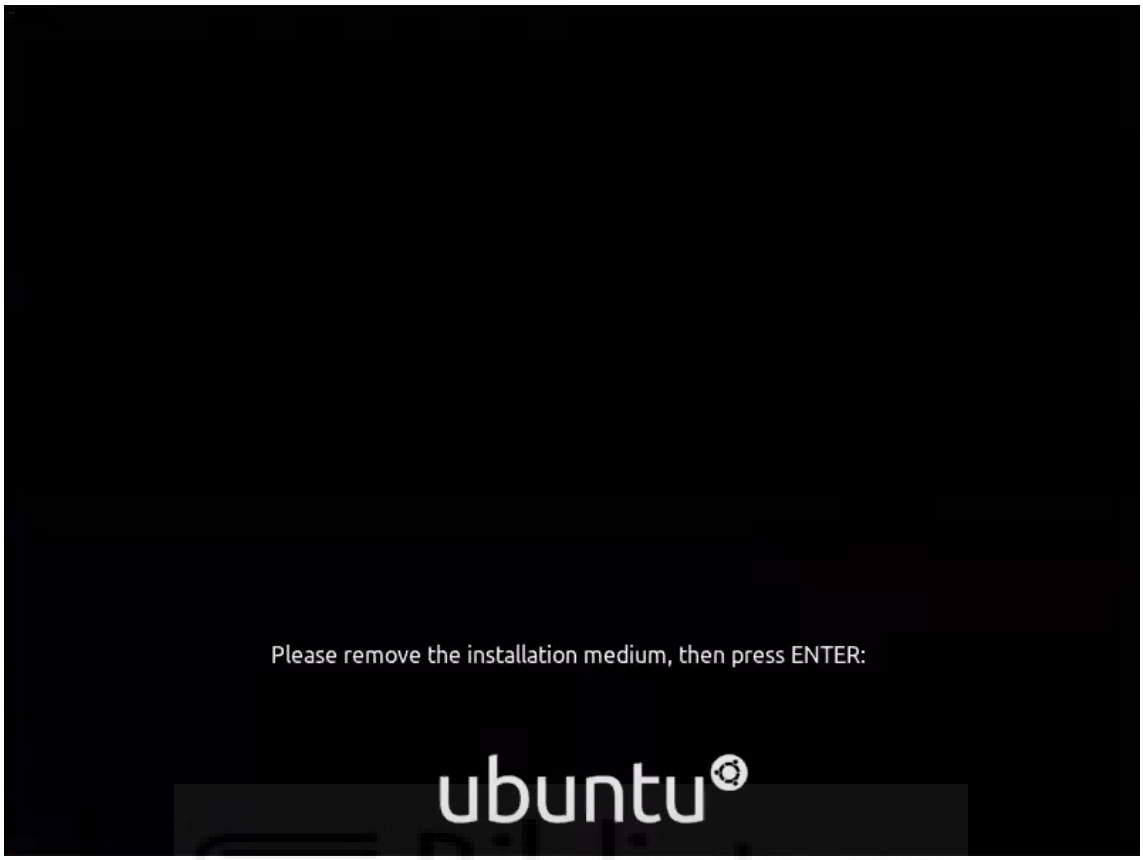


Figura 75. Pantalla para informar la retirada del disco duro

Arrancar desde el dispositivo

Con el ordenador apagado, se conecta el disco duro al ordenador y se enciende. Durante la iniciación del ordenador pulsamos la tecla de la BIOS, que cada modelo de ordenador tiene el suyo, y si el usuario no sabe cuál es, puede buscarlo en internet.

Mediante esa tecla se accede al Boot Option Menu, donde se puede elegir desde donde quiere el usuario arrancar el equipo, en nuestro caso, se elige el disco duro externo, y ya se podrá trabajar con Ubuntu, sin crear ninguna partición de disco. Cada vez que el usuario quiera trabajar con Ubuntu, tendrá que volver a darle a la tecla para seleccionar el dispositivo.

ANEXO 3: Instalación, configuración e introducción a ROS

En este anexo se verá cómo hay que instalar ROS en un PC con Ubuntu. En el trabajo instalaremos la versión Noetic Ninjemys, pues el brazo UR5 está completamente desarrollado para esta distribución y es la distribución disponible en Ubuntu 20.04.3 LTS.

Instalación

Antes de comenzar con la instalación, se debe configurar los repositorios de Ubuntu. Para ello, abrimos la aplicación de "Software y Actualizaciones", y en la primera pestaña "Software de Ubuntu", seleccionamos todas las opciones disponibles de "Descargable de Internet".

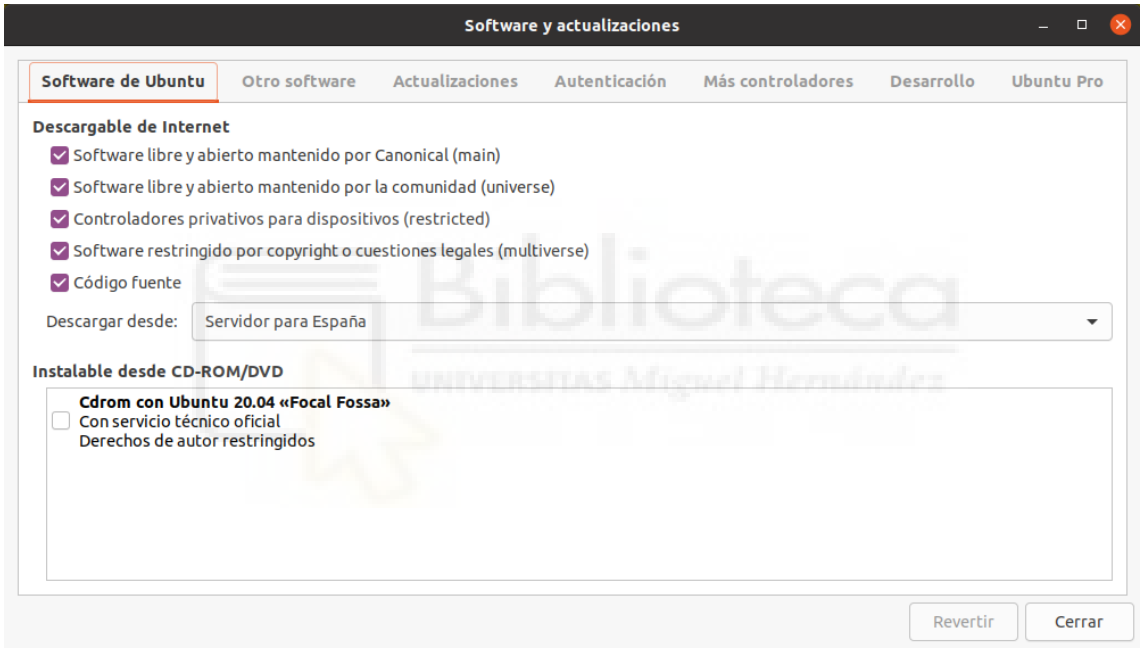


Figura 76. Aplicación "Software y actualizaciones" de Ubuntu

Una vez se haya configurado Ubuntu, se tiene que configurar la lista de los repositorios disponibles para que el PC acepte el software de packages.ros.org. Para ello, abrimos el terminal (CTRL+ALT+T) y ejecutamos el siguiente comando:

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

Prestar atención que todos los comandos que se han mostrado en el presente anexo y en el trabajo vienen precedidos por el símbolo del dollar ($). Simplemente es una distinción

entre los comandos que se han de introducir y cuales no. Por tanto, no se debe escribir ese símbolo antes de los comandos que se introduzcan en el terminal.

El siguiente es configurar las llaves de descarga, mediante “curl”, si no se encuentra instalado en el equipo, primero se ejecuta el comando para instalarlo y después la configuración de las llaves:

```
$ sudo apt install curl
$ curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add –
```

A continuación, es importante saber si nuestro sistema se encuentra actualizado. De cara a tener el PC con las últimas versiones de código de los repositorios se ejecuta el siguiente comando:

```
$ sudo apt update
```

Cuando se haya actualizado el sistema, se puede proceder a la instalación del paquete completo de ROS. En este momento, se presentan diferentes opciones de instalaciones, se puede instalar el sistema completo de escritorio con todos los paquetes, una versión de escritorio con la base de ROS y más herramientas o solamente la base de ROS sin herramientas. Para más información sobre cada tipo de instalación se recomienda acceder al siguiente enlace:

http://wiki.ros.org/noetic/Installation/Ubuntu

Lo recomendable es instalar la versión completa, que cuenta la base de ROS, las herramientas de rqt, rviz, librerías para robots, simuladores 2D/3D, navegación y percepción 2D/3D, y por ello, para el presente trabajo se ha instalado esta versión, ejecutando el siguiente comando:

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

Durante la instalación, Linux se conectará con los servidores de ROS y nos pedirá la confirmación de la descarga de los archivos necesarios para la instalación. Solamente hemos de aceptar la descarga y esperar a que se acabe la instalación.

Una vez hayan terminado de instalase todos los paquetes de ROS, se debe configurar el entorno y para ello, cada vez que se quiera trabajar en ROS, se tiene que escribir el siguiente comando en cada terminal que se quiera utilizar sus comandos:

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

Pero, es más conveniente obtener automáticamente ese comando cada vez se quiera utilizar ROS, para ello, ejecutando el siguiente comando no nos tendremos que preocupar del comando anterior:

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

Al ejecutarse, lo que hace es que escribe el comando previo en la última línea del archivo "bashrc", que se trata de un archivo de configuración que carga todas las rutas y comandos que Linux puede reconoces en base a los programas que tiene instalados.

Por último, es momento de inicializar las dependencias para crear y administrar espacios de trabajos ROS y paquetes, porque lo que se ha instalado es para poder ejecutar los paquetes de ROS principales.

Para inicializar las dependencias primero se ejecuta el siguiente comando, el cual instalará una herramienta para crear paquetes:

```
$ sudo apt install python3-rosdep python3-rosinstall python3-rosinstall-generator python3-wstool build-essential
```

Ahora se inicializa “rosdep”, que nos permite instalar fácilmente las dependencias del sistema para el código fuente que se desee compilar y es necesario para algunos componentes principales. Primero instalamos “rosdep” desde el terminal:

```
$ sudo apt install python3-rosdep
```

Para inicializarlo, ejecutamos los siguientes comandos:

```
sudo rosdep init
rosdep update
```

Introducción a ROS

En este último apartado del anexo se elabora una breve guía con los conceptos principales del funcionamiento de ROS, basado en los tutoriales que ROS ofrece en su página oficial, accediendo desde este enlace:

es/ROS/Tutoriales - ROS Wiki

Se recomienda, para adquirir más información del funcionamiento de ROS, hacer todos los tutoriales posibles que la página proporciona. Pero como se menciona en las conclusiones, realizarlos no conlleva a tener un gran dominio de ROS y su funcionamiento, aunque permite obtener unas bases sobre qué apoyarse para poder iniciarse en el sistema y crear los primeros programas o entender y modificar los ya creados.

Crear un Workspace:

En ROS, para que los programas puedan ejecutarse sin problemas es necesario crear un workspace o espacio de trabajo, que nos facilitará las cosas cuando queramos compilar los programas y que el Master de ROS pueda reconocer nuestros nuevos programas como nodos ejecutables.

Para crear un espacio de trabajo, que le pondremos de nombre en este ejemplo "catkin_ws", se ejecutará lo siguiente:

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

Al ejecutarlo por primera vez en su espacio de trabajo, se crea un enlace llamado "CMakeLists.txt" en la carpeta "src". Además, aparecerá en el directorio actual dos carpetas más, "build" y "devel". Dentro de la carpeta "devel" aparecerán varios archivos de setup, de los cuales, nos interesa solamente el archivo "setup.bash" y que tendremos que enlazarlo con el terminal para que los programas que se creen puedan ser lanzados por consola. Para ello, se ejecuta lo siguiente:

```
$ source devel/setup.bash
```

Para que siempre esté activo este archivo, podemos añadir el comando anterior en archivo "bashrc" en la última línea. Si se quiere comprobar que el espacio de trabajo esté correctamente superpuesto por el comando de configuración, se puede ejecutar lo siguiente:

```
$ echo $ROS_PACKAGE_PATH
```

Y que el resultado sea el directorio en el que se encuentra:

/home/youruser/catkin_ws/src:/opt/ros/noetic/share

Navegar por el sistema de archivos de ROS:

El código se distribuye en muchos paquetes de ROS, y tener que navegar y buscar algunos de ellos con solamente las herramientas de línea de comandos que ofrece Ubuntu como son "ls" o "cd" puede resultar ser muy tedioso y perder mucho tiempo. Por ello, ROS

proporciona algunas herramientas para que esa búsqueda se vuelva más rápida. Esas herramientas son “rospack”, “roscd”, “roscd log” y “rosls”.

- rospack: Permite obtener información sobre los paquetes. Presenta varias opciones y una de ellas es “find”. Se estructura de la siguiente manera:

  ```
  $ rospack find [package_name]
  ```

  Y devolverá por pantalla la ubicación del paquete en el disco duro. Si el paquete no existe, por pantalla no devolverá nada.

- roscd: Permite cambiar el directorio directamente a un paquete o una biblioteca. Se estructura de la siguiente manera:

  ```
  $ roscd <package-or-stack>[/subdir]
  ```

  Donde en “<package-or-stack>” se sustituye por el nombre de un paquete.

- roscd_log: Permite ir a la carpeta donde ROS almacena los archivos de registro. Si todavía no se ha ejecutado ningún programa ROS, el comando devolverá un error que indica que aún no existe. Su estructura es la siguiente:

  ```
  $ roscd log
  ```

- rosls: Permita listar directamente en un paquete por nombre en lugar depor ruta absoluta. Se estructura de la siguiente forma:

  ```
  $ rosls <package-or-stack>[/subdir]
  ```

  Para ahorrar tiempo en escribir el nombre de los archivo y paquetes en línea de comando, ROS permite la función de auto-completar pulsando la tecla del tabulador. Es decir, solamente escribiendo la mitad del nombre del paquete o menos y pulsar el tabulador, el sistema completará el nombre.

Crear paquetes ROS:

Para que un paquete se considere paquete ROS, se debe crear mediante la herramienta "catkin" y debe cumplir con algunos requisitos para que se consideren como tal:

- Contener un archivo package.xml: Proporciona metainformación sobre el paquete.
- Contener un CMakeLists.txt: Especifica qué se debe crear y que dependencia existen al compilar.
- Cada paquete debe tener su propia carpeta: No hay paquetes anidados ni varios paquetes que compartan el mismo directorio.

La estructura del workspace con los requisitos previos quedaría de la siguiente forma:

```
workspace_folder/        -- WORKSPACE
  src/                   -- SOURCE SPACE
    CMakeLists.txt       -- 'Toplevel' CMake file, provided by catkin
    package_1/
      CMakeLists.txt     -- CMakeLists.txt file for package_1
      package.xml        -- Package manifest for package_1
    ...
    package_n/
      CMakeLists.txt     -- CMakeLists.txt file for package_n
      package.xml        -- Package manifest for package_n
```

Figura 77. Ejemplo de estructura de un workspace

Desde el workspace que se ha creado con anterioridad, crearemos un paquete llamado "beginner_tutorials" dependiente de std_msgs, roscpp y rospy, que son otros tres paquetes para mostrar cómo funciona el comando "catkin_create_pkg, cuya estructura es la siguiente:

```
$ catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

A continuación, se muestra los pasos a seguir para crear el paquete:

```
$ cd ~/catkin_ws/src
$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

Se creará una carpeta con el nombre “beginner_tutorials” con los archivos “package.xml” y “CMakeLists.txt”.

Por último, con la herramienta *catkin* construimos el espacio de trabajo para hacer el paquete funcional, siguiendo estos pasos:

```
$ cd ~/catkin_ws
$ catkin_make
```

Comprendiendo los nodos de ROS:

En este punto del anexo se profundizará el trabajo con los nodos y los distintos usos de las herramientas.

El nodo más importante que se debe ejecutar porque es el que inicia el nodo Maestro y poder trabajar con ROS es el siguiente:

```
$ roscore
```

Apareciendo por pantalla la siguiente información:

```
... logging to ~/.ros/log/9cf88ce4-b14d-11df-8a75-00251148e8cf/roslaunch-machine_name-1303
9.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://machine_name:33919/
ros_comm version 1.4.7

SUMMARY
======

PARAMETERS
 * /rosversion
 * /rosdistro

NODES

auto-starting new master
process[master]: started with pid [13054]
ROS_MASTER_URI=http://machine_name:11311/

setting /run_id to 9cf88ce4-b14d-11df-8a75-00251148e8cf
process[rosout-1]: started with pid [13067]
started core service [/rosout]
```

Figura 78. Ejecución del comando "roscore"

Al ejecutarse y no se produce ningún error, en el terminal no se podrá seguir avanzando porque se queda mantenido el proceso de "roscore". Se abrirá una ventana nueva del terminal para continuar con los siguientes comandos.

El comando "rosnode" muestra información sobre los nodos ROS que se están ejecutando actualmente. Si se quiere enumerar los nodos activos solamente se tiene que ejecutar el siguiente comando:

```
$ rosnode list
```

También, si se quiere saber la información de un nodo específico, se ejecuta el siguiente comando:

```
$ rosnode info /<nombre_del_nodo>
```

Por último, un comando fundamental para iniciar un nodo es "rosrun", cuya estructura es el siguiente:

```
$ rosrun [package_name] [node_name]
```

A modo de ejemplo, se puede ver el funcionamiento de este nodo, llamando al simulador de la tortuga:

```
$ rosrun turtlesim turtlesim_node
```

Y se generará una ventana donde aparecerá una tortuga:

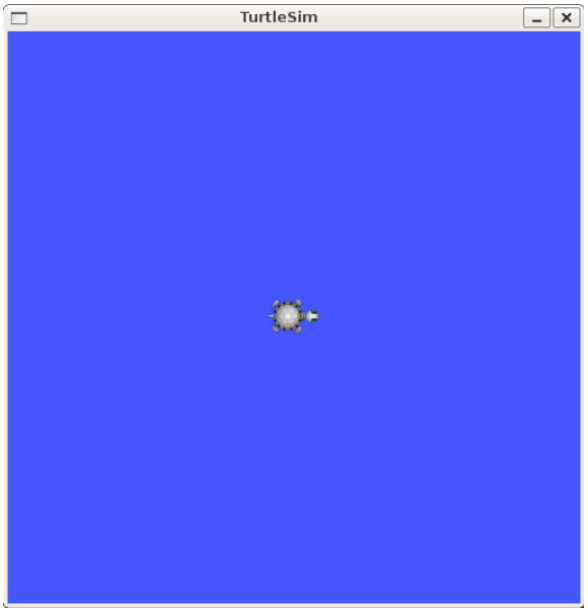


Figura 79. Programa de turtlesim_node

Comprendiendo Tópicos de ROS:

En este punto del anexo se profundizará el trabajo con los tópicos y los distintos usos de las herramientas.

Para observar cómo dos nodos se comunican por un "topic", teniendo ejecutado "roscore" en una terminal y el simulador de la tortuga en otro, abrimos otro terminal y ejecutamos lo siguiente:

```
$ rosrun turtlesim turtle_teleop_key
```

```
[ INFO] 1254264546.878445000: Started node [/teleop_turtle], pid [5528], bound on [aqy], xm
lrpc port [43918], tcpros port [55936], logging to [~/ros/ros/log/teleop_turtle_5528.log],
using [real] time
Reading from keyboard
---------------------------
Use arrow keys to move the turtle.
```

Figura 81. Aparición por pantalla del programa turtle_teleop_key

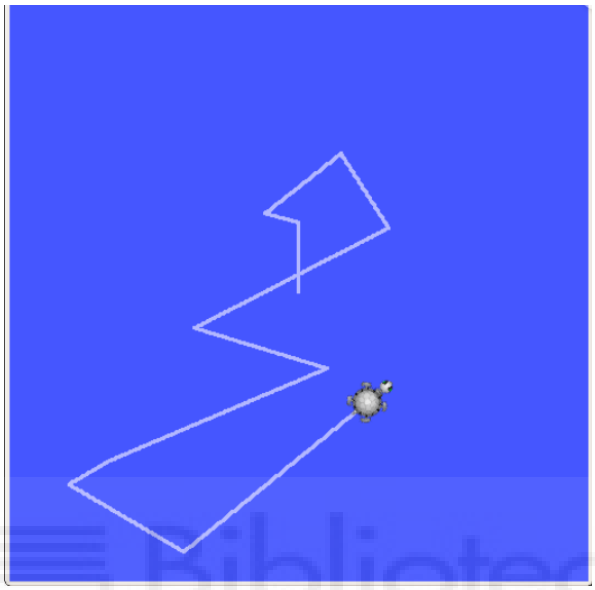


Figura 80. Ejemplo de programa turtlesim_node movido por teclado

Se trata de un nodo que permite controlar a la tortuga usando las flechas del teclado. Lo que significa que estos dos nodos compartirán información a través de "topics" que están suscritos o en los que publican.

Hay una herramienta muy útil, "rostopic", que permite obtener información sobre los "topics" que se encuentran activos y cuyas opciones son:

```
rostopic bw     display bandwidth used by topic
rostopic echo   print messages to screen
rostopic hz     display publishing rate of topic
rostopic list   print information about active topics
rostopic pub    publish data to topic
rostopic type   print topic type
```

Figura 82. Opciones de "rostopic"

También existe otra herramienta, llamada "rqt_graph", que crea un gráfico dinámico de lo que está ocurriendo en el sistema mostrando la relación entre los nodos activos que

existen y los “topics” que los comunican. Si abrimos una nueva terminal y ejecutamos el siguiente comando:

```
$ rosrun rqt_graph rqt_graph
```

Aparecerá algo similar a esto:

Figura 83. Ejemplo de utilización del rqt_graph

Donde aparecen en azul y en verde los nodos, y los tópicos en rojo. La imagen anterior muestra que el nodo azul publica en el topic “/turtle1/command_valocity” y el nodo verde está suscrito al mismo topic. Mediante esta herramienta se puede entender como funciona un programa o depurar su funcionamiento.

Para acabar, otra herramienta útil es “rqt_plot”, porque permite mostrar un gráfico de desplazamiento de tiempo de los datos publicados en los tópicos. Para lanzar la aplicación introducimos lo siguiente en el terminal:

```
$ rosrun rqt_plot rqt_plot
```

Aparecerá una ventana donde en la esquina superior izquierda se puede agregar cualquier tema a la trama. Si se escribe "/turtle1/pose/x" y presionamos el botón más, hacemos los mismo, pero con la posición 'y' ("/turtle1/pose/y"), nos aparece la ubicación x-y de la tortuga:

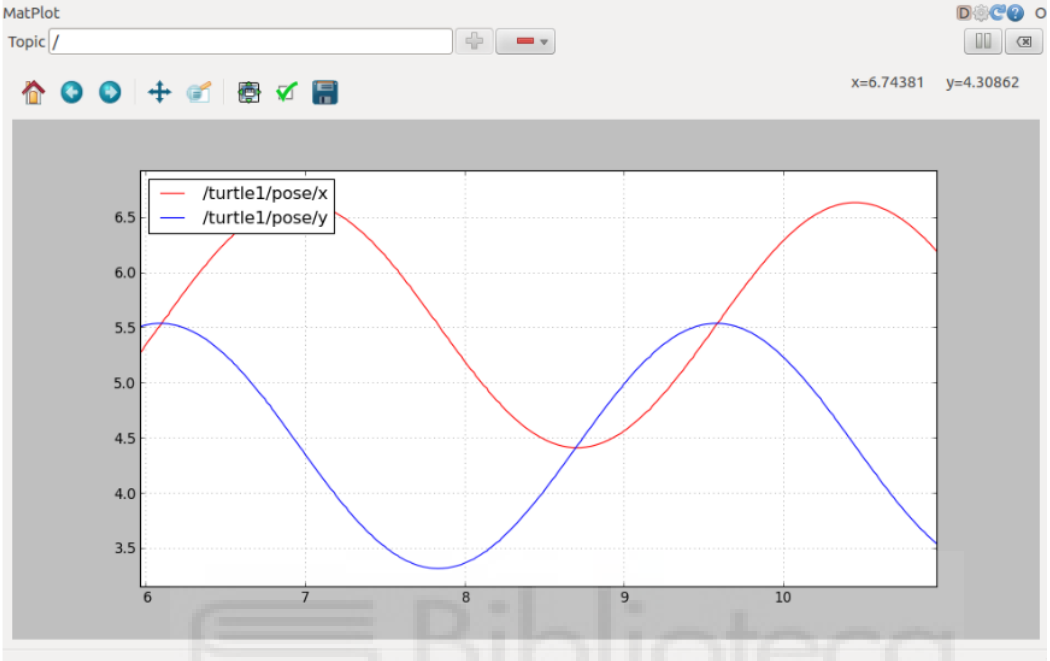


Figura 84. Ejemplo del rqt_plot

Comprendiendo Servicios y Parámetros:

Los servicios son otra forma de en qué los nodos pueden comunicarse entre sí. Permiten a los nodos enviar una solicitud y recibir una respuesta.

Una de las herramientas más importantes de los servicios para obtener información de los servicios activos es "rosservice". Presenta varias opciones en las que se puede utilizar:

```
rosservice list         print information about active services
rosservice call         call the service with the provided args
rosservice type         print service type
rosservice find         find services by service type
rosservice uri          print service ROSRPC uri
```

Figura 85. Opciones de "rosservice"

Otra de las herramientas que ofrece ROS sobre los servicios es "rosparam". Permite almacenar y manipular datos en el Servidor de Parámetros de ROS. Este servidor puede almacenar enteros, floats, booleanos, diccionarios y listas. La herramienta usa el lenguaje

marcado YAML para la sintaxis. Al igual que “rosservice”, presenta varias opciones de uso:

```
rosparam set            set parameter
rosparam get            get parameter
rosparam load           load parameters from file
rosparam dump           dump parameters to file
rosparam delete         delete parameter
rosparam list           list parameter names
```

Figura 86. Opciones de "rosparam"

Comando “roslaunch” y archivos “.launch”:

El comando “roslaunch” es una herramienta de ROS muy útil a la hora de lanzar a la vez varios nodos y mundos de gazebo los cuales están especificados en un archivo con extensión “.launch”.

La gran mayoría de programas están compuestos por diferentes nodos y para poder efectuarse correctamente suelen venir organizados en archivos de este estilo para que no se esté llamando a cada nodo por separado, lo que conlleva a una gran fluidez y sencillez del trabajo.

La estructura del comando es la siguiente:

```
$ roslaunch [package] [filename.launch]
```

Para que se pueda ejecutar el comando, previamente debe ser compilado, para que se añada al ~/.bashrc.

Por último, añadir que para poder entender la estructura de los archivos de lanzamiento con extensión “.launch” se recomienda acceder a los siguientes enlaces:

ROS/Tutorials/UsingRqtconsoleRoslaunch - ROS Wiki

roslaunch/XML/launch - ROS Wiki

LISTA DE FIGURAS