UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

"SIMULACIÓN Y PROGRAMACIÓN DE UN ROBOT UR5 INTEGRANDO TÉCNICAS DE VISIÓN ARTIFICIAL"

TRABAJO FIN DE GRADO

Septiembre -2023

AUTOR: Francisco Javier Rico Giner

DIRECTOR/ES: Arturo Gil Aparicio

Mónica Ballesta Galdeano

AGRADECIMIENTOS

Me gustaría transmitir mi más sincero agradecimiento a todos aquellos que me han ayudado en el desarrollo de este proyecto.

En primer lugar, agradecer a mis tutores, Arturo Gil Aparicio y Mónica Ballesta Galdeano, los cuales me brindaron la oportunidad de adentrarme en este proyecto. Gracias por vuestros consejos, por responder siempre que he necesitado ayuda y por ayudarme a expandir mis habilidades en el ámbito que rodea a este proyecto.

En segundo lugar, agradecer a mis padres por brindarme apoyo incondicional y por darme la oportunidad de estudiar y formarme. A mí familia en general, por escucharme y soportarme en los momentos difíciles, y también por compartir los momentos en los que logré alcanzar mis metas.

ÍNDICE

1. Introducción

1.1. Antecedentes y motivación

En la actualidad, el sector de la robótica está en continuo crecimiento y expansión, hay un aumento constante del número de robots instalados en las distintas fábricas alrededor del mundo. Según la Federación Internacional de Robótica, en 2021 el número de robots industriales alcanzó la cifra de 3.5 millones de unidades, un 15% más que el año anterior. Además, el número de instalaciones de robots industriales aumentó un 31% respecto al año predecesor.

En este ambiente en el que la robótica está cobrando una relevancia enorme en la industria, surgió la idea de realizar un proyecto en el que programar el robot *UR5* con el objetivo de que realice una tarea concreta, proyecto con el cual podría adentrarme en el mundo de la robótica industrial.

Además, el robot *UR5* es un robot colaborativo. Este tipo de robots es la opción elegida cada vez en más industrias, debido a que pueden trabajar en compañía de operadores. En el siguiente gráfico se va a mostrar el incremento del número de robots colaborativos que se están instalando anualmente.

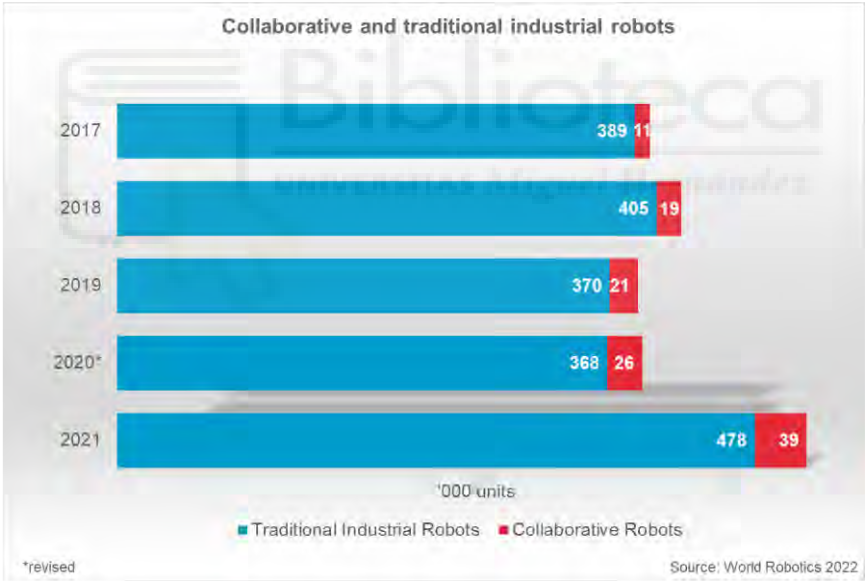


Imagen 1-Instalaciones de robots colaborativos frente a industriales tradicionales

En este gráfico se puede apreciar el aumento de la presencia de la robótica colaborativa en la industria, lo cual supuso una motivación clara, a realizar el proyecto utilizando un robot *UR5*.

Este proyecto también nace con la intención de empezar a explorar tanto el *software ROS* como *Gazebo*. Ambos *softwares*, son ampliamente utilizados en la industria de la robótica. *ROS* es utilizado tanto para el desarrollo *software* para robots como para comunicación de *hardware*, mientras que *Gazebo* es un simulador 3D. Durante la realización del proyecto exploraré muchas de las utilidades de estas

herramientas, lo que me dará cierta capacidad de adaptarme a necesidades futuras de la industria.

Hoy en día, se busca que los robots sean capaces de reconocer su entorno y tomar decisiones por sí mismos, por esa razón cada vez más se integran técnicas de visión por computador e inteligencia artificial a los sistemas robóticos.

Con la motivación de explorar los diferentes algoritmos que se podrían integrar a un sistema robótico, surgió la idea de realizar una aplicación de *pick and place* con el robot *UR5*, utilizando técnicas de visión por computador. Gracias a esto, con la realización del proyecto lograría no solo aprender a comandar el robot *UR5*, sino que también lograría explorar diferentes técnicas de visión artificial, con las cuales conseguir, que el robot pueda tomar decisiones en base a una serie de imágenes tomadas desde una cámara.

1.2. Objetivos

En este apartado, se van a mostrar los objetivos que van a servir como punto de partida para desarrollar todos los aspectos del proyecto. Durante la realización de dicho proyecto, todos los pasos que se van a dar, tendrán la intención de cumplir dichos objetivos.

1.2.1. Objetivos de aprendizaje

- Lograr introducirme en el mundo de la robótica industrial y algunas de sus aplicaciones.
- Realizar una investigación sobre el ámbito de la robótica colaborativa y sus diversas utilidades frente a la robótica industria convencional.
- Aprender nociones sobre el manejo del entorno de trabajo del sistema operativo Ubuntu y sobre el uso de comandos por terminal.
- Introducción al *software ROS* y familiarización con su sistema de entorno de trabajo y con su funcionamiento por medio de *nodes*, *topics* y *services.*
- Aprendizaje de las funcionalidades de *ROS* relacionadas con la robótica.
- Entender la estructura y el proceso de creación, modificación y uso, de los modelos robóticos mediante el lenguaje *URDF*.
- Adquirir la capacidad suficiente para aprovechar la capacidad de *path planning* de *MoveIt.*
- Averiguar cómo utilizar la interfaz de *MoveIt*, que nos permite utilizar dicho *software* con lenguaje *Python*.
- Adquirir los conocimientos suficientes para simular, los sistemas robóticos en el espacio 3D de *Gazebo.*

1.2.2. Objetivos de simulación

- Generar una simulación del robot *UR5* en un entorno 3D de *Gazebo*.
- Ser capaz de mover el robot *UR5* en simulación, tanto por cinemática directa como inversa, utilizando la herramienta *MoveIt*.
- Incluir una pinza en la simulación del robot *UR5*.
- Ser capaz de abrir y cerrar la pinza utilizando *MoveIt*.
- Incluir una cámara en la simulación del robot.
- Ser capaz de capturar la imagen que recibe la cámara en simulación.
- Preparar la imagen capturada mediante la cámara, para poder procesarla con un algoritmo que cumpla los requisitos que necesitemos.
- Construir una simulación de *Gazebo* en la que se encuentre el robot con la pinza y la cámara, y además todos los objetos necesarios para llevar a cabo la aplicación de *pick and place*.
- Tener las habilidades necesarias, para simular en el entorno de *Gazebo* los programas que se desarrollen durante el proyecto, con el fin de solventar los objetivos de aplicación que veremos en el siguiente apartado.

1.2.3. Objetivos de aplicación

- Realizar una aplicación manual de manejo del robot, que englobe todas las funcionalidades que necesitamos utilizar del paquete *moveit_commander*, con la finalidad de familiarizarnos con las distintas formas de mover el robot, previo a la realización de la aplicación final de *pick and place*.
- Programación y simulación de una aplicación de *pick and place*, utilizando el robot *UR5*, integrando técnicas de visión por computador mediante la ayuda de una cámara ubicada sobre el último eslabón del robot. Dichas técnicas nos permitirán integrar en la aplicación, un algoritmo de visión que consiga detectar las piezas del escenario y clasificarlas según el color, para conseguir que el robot mueva las piezas y las deje en un lugar u otro según su color.
- Mejorar la aplicación de *pick and place*, con el fin de que, si durante la ejecución de la aplicación una de las piezas de la escena es movida, eliminada u otra pieza es añadida a la escena, el nuevo programa sea capaz de detectar dichos cambios y manipular a pesar de esto, todas las piezas correctamente, sin cometer errores de posición causados por esas modificaciones de la escena.

1.2.4. Resumen de objetivos

El objetivo principal de este proyecto es la programación y simulación de una aplicación de *pick and place*, utilizando el robot *UR5*, integrando técnicas de visión por computador mediante la ayuda de una cámara ubicada sobre el último eslabón del

robot. Dicha cámara servirá para detectar y clasificar por color, las distintas piezas que existan en la escena para que, una vez detectadas, se pueda comandar al robot a manipular dichas piezas. Las piezas se clasificarán utilizando un algoritmo de visión que detecte el color de la pieza y una función decisoria que decidirá que pieza coger en cada momento.

Todo lo anterior se acompañará por la exploración y aprendizaje del *software ROS* y con la ayuda de la capacidad de *path planning* del *software RVIZ*. Además, se realizará un aprendizaje exhaustivo de la herramienta *Gazebo*, la cual nos permitirá realizar simulaciones con el *UR5*, en un entorno de simulación 3D.

2. Evolución histórica de la robótica y estado del arte

2.1. Evolución histórica

Para empezar a hablar de la evolución histórica de la robótica, debemos hablar primero de los autómatas. Ya que mucho antes de que existiera el concepto de robot o robótica, ya se conocía el concepto de autómata como máquina que imitaba la figura y movimientos de un ser animado.

Primeramente, estos autómatas solo se hacían servir de divertimento o espectáculo, y no fue hasta la época entre el siglo VIII y el siglo XV, cuando la cultura árabe empezó a utilizarlos para diversas aplicaciones prácticas de la vida cuotidiana. Un ejemplo de los autómatas de esta época es el "Gallo de Estrasburgo" de 1352. Según [1], este autómata que es el más antiguo que se conserva en la actualidad, formaba parte del reloj de la torre de la catedral de Estrasburgo y al dar las horas movía las alas y el pico.

Imagen 2-Gallo de Estrasburgo

En los siglos XV y XVI se realizaron se desarrollaron diversos autómatas primigenios como, por ejemplo, el 'León mecánico' desarrollado por Leonardo Da Vinci o el 'Hombre de palo' por Juanelo Turriano. En este caso, ambos autómatas se desarrollaron con el objetivo de entretener a la gente de la corte.

Según lo expuesto en [1], durante los siglos XVII y XVIII se crearon ingenios mecánicos que tenían alguna de las características de los robots actuales, pero su misión principal era la de entretener a las gentes de la corte y servir de atracción en las ferias. Se puede destacar el 'Pato de Vaucanson' de 1739, realizado por Jacques Vaucanson.

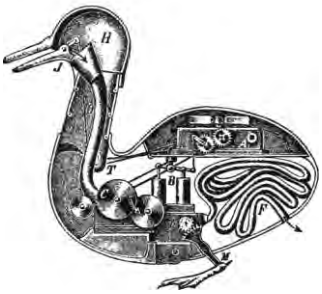


Imagen 3-Pato de Vaucanson

Según [2], el pato de Vaucanson era capaz de moverse gracias a un conjunto de levas e imitaba el comportamiento del mismo animal vivo.

Entre finales del siglo XVIII y principios del XIX, se empezaron a desarrollar ingeniosas invenciones mecánicas orientadas a la industria textil. Y a su vez, empezaron a surgir máquinas programables como lo era el 'Telar de Jacquard' de 1801.

Imagen 4-Telar de Jacquard

Este telar era programable usando tarjetas perforadas. Como argumenta [1], partir de este telar se empiezan a utilizar dispositivos automáticos en la producción, dando paso a la automatización industrial.

Hasta entonces no existía el concepto de robot, y no fue hasta el 1921 cuando surgió la palabra robot. Fue usada por primera vez en una obra de teatro del escritor checo *Karel Capek*, llamada *Rossum's Universal Robot*.

Su origen viene del vocablo checo *Robota*, que significa esclavitud o servidumbre. Ya que la obra trata unas máquinas androides llamadas robots que servían a sus jefes humanos. Y en los siguientes años, el término robot se mantuvo gracias a los escritores del género literario de ciencia ficción.

Cómo dice [1], fue el escritor Isaac Asimov quien popularizó la palabra robot, sobre todo cuando en 1945 publicó en la revista *Galxy Science Fiction,* una historia en la que enunció sus tres leyes de la robótica. Leyes las cuales seguiría en sus historias en las que aprecian robots. Además, a Asimov se le atribuye la creación del término robótica.

Para la aparición del concepto de robot como máquina industrial hay que avanzar un poco más en el tiempo hasta la época de los 50. En esa época existían los telemanipuladores, progenitores más directos de los robots, que eran máquinas en las que era indispensable la interacción continua de un operador. A partir de ese periodo se procede a la sustitución del operador por un programa de ordenador que controlase los movimientos del manipulador; esto da paso al concepto de robot.

Atendiendo a [1], George Devol, ingeniero norteamericano, estableció las bases del robot industrial moderno en 1954. Devol concibió la idea de un dispositivo de

transferencia de artículos programada que se patentó en 1961. Sería en 1956 cuando Devol y Joseph Engelberger empiezan a trabajar juntos, con la intención de darle utilidad a la idea de George Devol. Con esa intención fundaron la compañía de robots llamada, *Unimation* y en 1960 instalaron en una planta de General Motors el primer robot industrial, el robot *Unimate*.

En 1969 *Unimate* se alían con *Kawasaki* y crean el robot llamado, *Kawasaki-Unimate 2000*.

Imagen 5-Kawasaki-Unimate 2000

En los siguientes años siguieron surgiendo nuevas compañías de robótica y nuevos modelos de robots, todos ellos con configuraciones esférica y antropomórfica, de uso específicamente válido para la manipulación. Pero siguiendo lo expuesto en [1], un cambio notable llego en 1982 cuando se desarrolla el concepto de robot SCARA, que reducía los grados de libertad y tenía un coste más limitado, configurado sobre todo para el ensamblado de piezas.

Hasta día de hoy la evolución de los robots industriales está siendo extremadamente rápida, teniendo a día de hoy robots en casi todas las áreas productivas y tipos de industria.

Hoy en día, la mayor parte de robots de base estática son utilizados para ensamblado, soldadura, etc. Además, existe un continuo desarrollo de la robótica con el objetivo de aumentar su movilidad, destreza y autonomía. Aunque no todos los robots están orientados a aplicaciones exclusivamente industriales, también hay robots dedicados a aplicaciones espaciales, submarinas o subterráneas entre otras. Vamos a exponer algunas de esas aplicaciones en el siguiente apartado.

2.2. Estado del arte

Dentro del apartado del estado del arte se pretende contextualizar el proyecto dentro de la situación en la que se encuentra la industria hoy en día. Se hablará tanto de las principales áreas de investigación relacionadas, como de los avances y contribuciones más recientes y de las metodologías que se siguen hoy en día, todo esto relacionado con la temática ligada al proyecto, que es, la industria 4.0, la robótica, la robótica colaborativa y la visión artificial.

2.2.1. Industria 4.0

El término industria 4.0 se refiere a la tendencia actual que siguen la gran parte de la industria ligadas a la automatización y la interconexión en sus diferentes sistemas de fabricación industriales. Esta nueva industria viene relacionada con la cuarta revolución industrial, revolución la cual se caracteriza por la intercomunicación inteligente tanto de las máquinas como de los procesos industrial.

Imagen 6-Industria 4.0

La industria 4.0, término que surgió en Alemania en 2011, está revolucionando la forma en que las empresas producen y distribuyen sus productos. Dichas empresas están mejorando sus procesos de fabricación, integrando nuevos conceptos como pueden ser, por ejemplo, el internet de las cosas (*Internet of Things*), conceptos como *cloud computing*, *data managment*, o la inteligencia artificial, que junto a la automatización industrial consiguen crear procesos de fabricación inteligentes y más eficientes.

Incluir todo lo mencionado anteriormente y además lograr una interconexión absoluta de los elementos de una fábrica se logra a partir de un proceso de digitalización. Este proceso viene respaldado por las nuevas tecnologías informáticas, incluyendo los nuevos sistemas de comunicación emergentes, sistemas ciber-físicos, el internet de las cosas, etc.

Aunque suponga muchos beneficios para la empresa, este proceso de digitalización, supone realizar el paso supone una ardua tarea. Debido a que requiere, la digitalización y comunicación de todos los procesos, además de tener que asegurar un sistema de protección de datos que, en empresas muy extensas, puede conllevar una gran dificultad y un coste muy alto.

Tanto la robótica como la visión artificial son claves en la industria 4.0, ambas refuerzan el concepto de la automatización industrial. Por su parte la visión artificial es muy utilizada en sistemas de detección de objetos, sistemas de clasificación o incluso sistemas de percepción de errores. Juntando ambas tecnologías podemos tener sistemas robotizados autónomos guiados por la visión artificial, que pueden ser incluidos en cadenas de producción, aumentando así la productividad de la cadena y reduciendo costes a medio/largo plazo.

La robótica es fundamental para la automatización de sistemas de producción. Los robots colaborativos son uno de los avances más grandes dentro de esta cuarta revolución ya que permiten que los humanos pueden interaccionar directamente con los robots para realizar diversas tareas.

2.2.2. Robótica

La robótica es un campo en constante evolución y crecimiento, y cada vez se está incorporando en más sectores. En la actualidad, la robótica en la industria se está centrando en la realización de las tareas repetitivas de la industria, además de tareas que puedan resultar peligrosas para un operador humano. Esto último permite que humanos no corran el peligro presente en algunas producciones industrial y además ganen tiempo para otras tareas.

Además de la robótica industrial, existen diversas áreas de investigación en el campo de la robótica. Algunos ejemplos de las principales áreas de investigación son:

- **Enjambre robótico**

Se están investigando robots que pueden comunicarse unos con otros, para funcionar como hormigas.

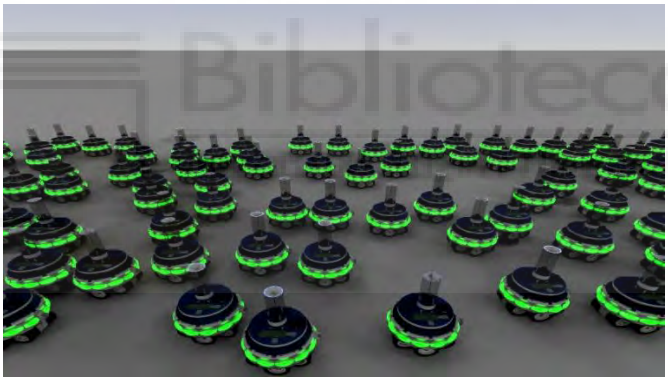


Imagen 7-Enjambre robótica

- **Inteligencia artificial para la robótica**

Muchos esfuerzos en la actualidad se centran en combinar las técnicas de la inteligencia artificial con la robótica, con algoritmos de *machine learning,* con la intención de que el sistema robótico pueda tomar decisiones por su cuenta.

- **Robótica biomédica y de servicios**

En esta área se busca servir a humanos que necesiten de servicios por incapacitación o necesidad. Por ejemplo, desarrollo de robots que puedan llevar comida a los pacientes en los hospitales.

Pero esta área no se queda ahí, ya que también investiga la forma de controlar robots mediante ondas cerebrales, utilizar exoesqueletos para facilitar los esfuerzos físicos de trabajadores o incluso llegando a procedimientos tan complejos como la cirugía robótica.

- **Robótica colaborativa**

Básicamente, la robótica colaborativa centra sus esfuerzos en desarrollar robots que puedan realizar diversas tareas junto a los humanos. Para ello deben adaptarse a la presencia de los humanos y evitar que ellos corran peligro. Pueden utilizarse para trabajos de soldadura donde el operados le marque la trayectoria de soldado o incluso en cirugía, en la cual el cirujano utilice el robot como una herramienta más para operar.

2.2.3. Robótica colaborativa

Los robots colaborativos, también llamados *cobots*, tienen por objetivo automatizar procesos compartiendo espacio de trabajo con personas y otras máquinas. Es un robot de tipo industrial, que suele tener menores dimensiones que los robots industriales al uso, y su funcionalidad principal es que puede trabajar con humanos sin resultar ser un peligro para ellos.

Estos *cobots* se utilizan para una gran variedad de tareas como, por ejemplo, ensamblaje, atornillado, soldadura, operaciones de pick and place, inspección de piezas, cirugía, etc. El diseño de robots colaborativos sigue diversas metodologías como, por ejemplo:

- **Fácil programación y reprogramación**

Los *cobots* deben ser fáciles de programar, ya que pueden estar en contacto diario con operadores que no tengan experiencia con la programación.

- **Diseño centrado en el operador**

Este tipo de robots deben estar diseñados para trabaja cooperando con usuarios, por tanto, deben ser fáciles de usar y fáciles de maniobrar, de manera que cubran las necesidades del usuario final.

- **Garantizar la seguridad del usuario**

En el diseño de un robot colaborativo, lo más importante es garantizar la seguridad en su uso por un operador. Se deben incluir medidas de seguridad en su diseño, ya que podrán trabajar a escasa distancia del usuario.

Debido a la evolución de la industria 4.0 y a la influencia de los robots colaborativos en ella, muchos fabricantes de robots han creado su propia línea de robots colaborativos. Entre las empresas más importantes se encuentran, *Kuka*, *Abb*, *Omron*, etc. Pero la empresa pionera en la creación e implementación de *cobots* en las industrias es *Universal Robots*. Además, cabe destacar que los robots de esta marca se utilizan en más de 50000 centros de producción en todo el mundo.

Proyecto BROCA

En la actualidad se están realizando infinidad de proyectos con las tecnologías colaborativas de *Universal Robot*. Un ejemplo de proyecto, en este caso usando el robot UR5, es el proyecto BROCA.

El proyecto BROCA, presentó el prototipo de lo que podría ser el primer robot quirúrgico desarrollado en España. El robot BROCA está compuesto por tres brazos UR5, los cuales interactúan coordinadamente o autónomamente según las necesidades.

Imagen 8-Proyecto BROCA

Estos robots UR5 han sido seleccionados por su sencillo sistema de programación, que ha permitido adaptar sencillamente su software a las necesidades específicas, además de por ser fácilmente instalable debido a que el UR5 pesa 18 kg.

El principal objetivo del proyecto BROCA es que exista uno de estos robots en cada centro hospitalario y, por tanto, permitir que cirujanos de todo el país puedan controlarlo de forma segura para realizar intervenciones sobre todo en cirugía laparoscópica. Una pantalla de visión 3D permitiría al cirujano ver en todo momento los detalles de la intervención, además de tener el control del robot a través de unos mandos.

Con la integración del robot UR5 en este proyecto queda demostrado que los robots colaborativos no solo se usan en aplicaciones industriales de tareas repetitivas en las que tengan que colaborar con operarios, si no que pueden llegar a múltiples ámbitos de la sociedad, teniendo una relevancia muy grande en la evolución de la industria 4.0.

2.2.4. Visión por computador

La visión por computador o visión artificial consiste en el procesamiento o tratamiento de imágenes, para varios fines. Estos fines pueden ser muy diversos, ya que pueden ir desde eliminar ruido de una imagen, realizar procedimientos de edición de imágenes artísticas, realizar transformaciones morfológicas en una imagen y pueden escalar hasta operaciones tan complejas como interpretar la escena o ayudar a máquinas a "comprender" el mundo visual que les rodea.

Hoy en día, la visión artificial se está integrando en la gran mayoría de industrias alrededor del mundo. Es una pieza clave en la cuarta revolución industrial, ya que permite a la maquinaria estar al corriente de su entorno. Eso implica que sea uno de los ámbitos en los que más se están centrando las investigaciones hoy en día. Algunos de los principales ámbitos de investigación que se siguen en este campo son:

- **Detección, parametrización y clasificación de objetos**

Esta área de la visión por computador es muy importante sobre todo en la industria. Abarca varias funcionalidades, ya que trata tanto del reconocimiento y la localización de objetos, como de su clasificación a través del cálculo de las características de cada objeto.

Además, es muy importante para la robótica ya que permite detectar la existencia de un objeto en la escena y además hallar sus coordenadas. La clasificación sirve principalmente para el reconocimiento y diferenciación de objetos.

- **Mapeo y reconstrucción 3D**

El mapeo y reconstrucción 3D, están cobrando cada vez más importancia. El mapeo 3D es habitual principalmente en la robótica móvil, en tareas tan complejas como las aeroespaciales, por ejemplo, en las que algún tipo de dispositivo tenga que mapear un lugar desconocido. Pero también en utensilios de uso tan común como un robot aspirador, que puede encontrarse hoy en día en cualquier casa. La funcionalidad del mapeo 3D suele ser realizar un reconocimiento del lugar donde se realizará una tarea con el fin de evitar colisiones, o también puede servir para mapear un lugar desconocido.

La reconstrucción 3D, puede tener usos en control de calidad o extracción de características. Principalmente consiste en, a través de algún tipo de sensor, ser capaz de reconstruir un objeto concreto o una escena concreta.

- **Aprendizaje profundo**

El aprendizaje profundo está directamente ligado con la inteligencia artificial y consiste en el análisis de datos visuales con el objetivo de conseguir una IA, capaz de reconocer patrones complejos tanto en imágenes como en videos.

Además de estos tres campos de investigación que hemos comentado, existen bastantes más ya que el ámbito de la visión artificial está en continuo desarrollo y evolución.

Pero, aunque todavía esté en desarrollo, la visión por computador ya está latente en muchos ámbitos de la industria. Algunas de las aplicaciones más reconocibles son:

- **Logística**

La logística tradicionalmente había estado a cargo de operadores, lo que requería largas jornadas de descarga de camiones y de ordenado de los, incluso llegando a crear problemas físicos severos en operadores que realizaran ordenaciones o descargas manuales.

La visión artificial en conjunto con la robótica permite contrarrestar los problemas derivados del factor humano. La corporación "Amazon", utiliza sistemas de visión por computador que pueden ayudar a descargar un tráiler de inventario en solo 30 minutos, lo cual podría tomar horas sin estas soluciones.

- **Control de calidad**

Gracias a la visión artificial, haciendo uso de sensores adecuados se puede automatizar todo un proceso de selección y control de calidad. Estas operaciones de descarte suelen ser tediosas y repetitivas para operadores, y requieren muchas horas. Esas condiciones pueden causar que un operario cometa errores.

La visión permite realizar ese control de calidad, en toda clase de industrias, bajando notablemente el porcentaje de error y aumentando la velocidad.

Imagen 9-Visión artificial para control de calidad

- **Medicina**

Hoy en día se están empezando a generar e investigar aplicaciones de visión artificial, para varios campos de aplicación en la medicina. Por ejemplo, estas aplicaciones se han llevado a cabo en realización de diagnósticos, detección de tumores e incluso cirugías mínimamente invasivas.

También existen otras aplicaciones en las que la visión simplemente ayuda a los especialistas tengan una mejor interpretación de la imagen, acondicionándola y extrayendo la información necesaria.

- **Automoción**

Con la evolución de la automoción, prácticamente todos los nuevos modelos de cualquiera de las marcas están utilizando sensores y avisadores, para avisar de posibles colisiones, problemas que pueda tener el vehículo internamente. Tradicionalmente, esos vehículos capaces de detectar obstáculos, utilizaban tecnología *LiDAR*, tecnología la cual utiliza ondas láser de pulso para trazar un mapa de distancia entre objetos.

Empresas como Tesla están empezando a sustituir esos dispositivos, y están empezando a utilizar la visión por computador para desarrollar sistemas de autoconducción. Hablando de Tesla, todos los modelos de esta empresa llevan decenas de cámaras implantadas, lo que le permite tener una visibilidad 360 para que, con su sistema *AutoPilot* se pueda conducir el vehículo en cuestión sin la supervisión humana. Cabe destacar que estos sistemas de piloto automático en vehículos como coches, todavía están en desarrollo.

- **Sistemas de *picking* y *packing***

A nivel industrial, es la aplicación más utilizada. Sobre todo, son dos de los procesos más demandados dentro del control logístico. Pueden ser realizados tanto por robots industriales al uso, como por *cobots*. Estos pueden resolver problemas complejos de *picking* a través de cámaras, localizando y clasificando los diferentes productos dentro de una línea de producción.

Imagen 10-Visión artificial para pcking y packing

- **Seguridad**

Es muy utilizada la visión por computador para la seguridad. Permite rebajar notablemente los costes de los servicios de seguridad, ya que mediante las imágenes se detectan comportamientos sospechosos.

Los nuevos avances han hecho posible que seamos capaces de reconocer y distinguir fiablemente características humanas de otras no humanas. Eso nos permite delimitar o controlar perfectamente los accesos de humanos a distintos lugares. También nos permite detectar rápidamente movimientos anómalos en maquinarias, como pueden ser robots, que puedan desembocar en desastres.

También existen aplicaciones de reconocimiento facial para controlar el acceso de personas a lugares. Como también se emplean técnicas de visión en lugares con tráfico, con el objetivo de controlarlo y prevenir peligros.

3. Fundamentos teóricos

En este tercer apartado, haremos una recopilación de los fundamentos teóricos más importantes, relacionados con las técnicas y los procedimientos utilizados en el proyecto.

3.1. Robótica

El desarrollo de los fundamentos teóricos relacionados con la robótica, se van a referenciar tanto en [1] como en [2].

3.1.1. Cinemática

Atendiendo a [2], la cinemática del robot se refiere al estudio del movimiento del extremo del robot, sin considerar las causas que lo generaron, con respecto a un sistema de referencia que normalmente es la base. Fundamentalmente, la cinemática busca una expresión que relacione la posición y orientación del extremo del robot, con las respectivas posiciones angulares de sus articulaciones, en función del tiempo.

Esa relación cinemática se puede encontrar planteándonos dos problemas distintos, el problema de la cinemática directa y el problema de la cinemática inversa. La cinemática directa implica el hallazgo de la posición y orientación del extremo del robot respecto al sistema de referencia de la base, si conocemos las coordenadas articulares y los parámetros geométricos del brazo. La cinemática inversa en cambio implica el cálculo de las coordenadas articulares del brazo, que consiguen que el extremo del robot esté en una posición y orientación determinadas.

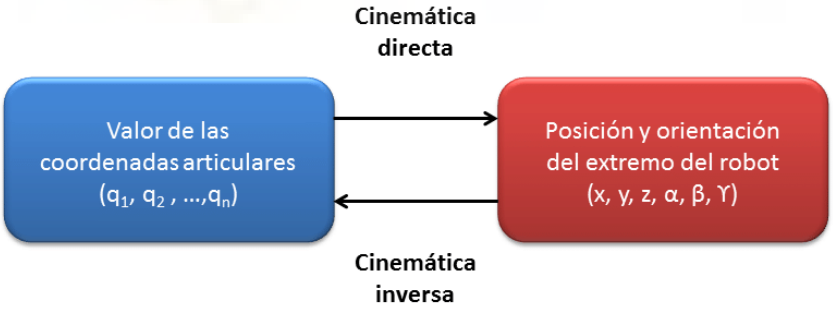


Imagen 11-Relación cinemática directa e inversa

3.1.1.1. Cinemática directa

Como hemos comentado anteriormente, la cinemática directa permite determinar la posición y orientación del extremo del robot, conocidas las posiciones articulares y los parámetros geométricos del brazo robótico. Para ello, se utilizan herramientas matemáticas como el álgebra vectorial y matricial, con el fin de trabajar con transformaciones homogéneas.

Según [1], un robot se puede entender como una cadena cinemática formada por eslabones, unidos entre sí por articulaciones. Estableciendo un sistema de referencia fijo en la base, y describiendo, con las transformaciones homogéneas, los sistemas de coordenadas entre las articulaciones, se puede reducir el problema cinemático a una sola matriz homogénea que permita definir la posición y orientación del extremo robot, en función de las coordenadas articulares.

Básicamente, se basa en encontrar las siguientes ecuaciones:

$$
\begin{aligned}
x &= \mathrm{f}_x(q_1, q_2, q_3, q_4, q_5, q_6) \\
y &= \mathrm{f}_y(q_1, q_2, q_3, q_4, q_5, q_6) \\
z &= \mathrm{f}_z(q_1, q_2, q_3, q_4, q_5, q_6) \\
\alpha &= \mathrm{f}_\alpha(q_1, q_2, q_3, q_4, q_5, q_6) \\
\beta &= \mathrm{f}_\beta(q_1, q_2, q_3, q_4, q_5, q_6) \\
\gamma &= \mathrm{f}_\gamma(q_1, q_2, q_3, q_4, q_5, q_6)
\end{aligned}
$$

Imagen 12-Ecuaciones cinemática directa

Donde, x, y, z, Alpha, beta y gamma, son los parámetros del extremo del robot en función de las posiciones articulares. Soluciones las cuales, se van a basar en el empleo de las matrices de transformación homogéneas.

Ciertamente, en algunos casos hallar estas relaciones es muy fácil, mediante consideraciones geométricas. Por ejemplo, para un sistema de dos grados de libertad:

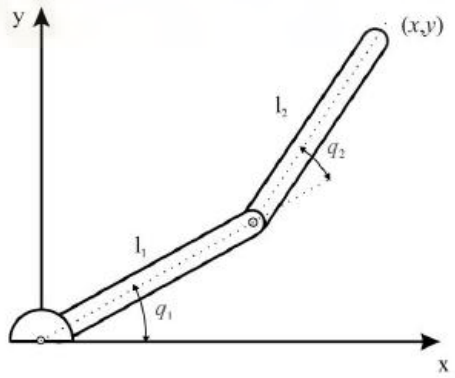


Imagen 13-Sistema de 2GDL

Podemos obtener esa relación como:

$$x = l_1 cos(q_1) + \ l_2 \cos(q_1 + q_2)$$

$$y = l_1 sen(q_1) + \ l_2 \mathrm{sen}(q_1 + q_2)$$

Pero para problemas más complejos, donde aumenta el número de grados de libertad se necesita un método sistemático para plantear las ecuaciones de la cinemática directa. En esos casos entra en juego el algoritmo de *Denavit-Hartenbeger*, que podemos ver en el apartado 4.1.2 de [1].

El objetivo es encontrar una matriz homogénea total, que relacione las coordenadas del extremo del robot con las coordenadas articulares, y para ello debemos utilizar la siguiente expresión.

$$T = {}^{0}A_1\,{}^{1}A_2 \cdots {}^{n-1}A_n$$

Imagen 14-Expresión matriz homogénea total

En la cual, la matriz "T" o matriz homogénea total, es el resultado del producto de las matrices de transformación de todas las articulaciones que existan en el robot. La problemática en este punto es, que hallar las matrices homogéneas de cada articulación es complejo y, además, cada una de esas matrices tiene doce elementos. Esto se simplifica utilizando el método de *Denavit-Hartenberg*.

El método de *Denavit-Hartenberg* propone hallar la matriz de transformación homogénea entre el eslabón i-1 y el eslabón i, utilizando cuatro matrices de transformación sucesivas. Y una vez tengamos esas cuatro matrices, solo tendremos que multiplicarlas para encontrar la matriz de transformación total. Las cuatro transformaciones consecutivas son:

1) Una rotación en torno al eje z_{i-1} un ángulo θ_i
2) Una translación a lo largo del eje z_{i-1} una cantidad d_i
3) Translación a lo largo del eje x_i una distancia a_i
4) Rotación en torno a X_i un ángulo α_i

Esas cuatro transformaciones tienen la siguiente forma.

$$^{i-1}T_a = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 & 0 \\ \sin(\theta_i) & \cos(\theta_i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha_i) & -\sin(\alpha_i) & 0 \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Imagen 15-Cuatro matrices de transformación

Una vez tengamos estas cuatro ecuaciones, siguiendo el método de *Denavit-Hartenberg* podemos obtener la matriz total de transformación de una articulación en concreto.

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

Imagen 16-Expresión matriz de transformación entre dos eslabones

Entendiendo ese procedimiento de transformaciones sucesivas, ya podríamos encontrar las matrices de transformación entre todos los eslabones del robot y, por tanto, podríamos aplicar el algoritmo completo de *Denavit-Hartenberg.*

A grandes rasgos, para utilizar este algoritmo, primero colocaremos todos los ejes de coordenadas sobre sus respectivas articulaciones, siguiendo el criterio de

colocación de los ejes de *Denavit-Hartenberg*, criterio el cual lo podemos consultar en [2]. Luego hallamos los parámetros de *Denavit-Hartenberg*, correspondientes a cada transformación entre articulaciones. Con esos parámetros, podremos hallar las matrices de homogéneas de transformación entre ejes, siguiendo el método de la imagen 15 Cuando calculemos las matrices de cada una de las articulaciones, podríamos calcular la matriz "T" de transformación, como se expone en la imagen 14.

Una vez tengamos obtenida la matriz "T", ya tendríamos desarrollada la cinemática directa de la aplicación robótica en cuestión. Ya que, con esa matriz podríamos calcular directamente las coordenadas del extremo del robot, si conocemos las posiciones articulares.

3.1.1.2. Cinemática inversa

Como bien sabemos, la cinemática inversa consiste en hallar las coordenadas articulares del robot, que consigan que el extremo del robot esté en una posición y orientación especificadas. Como argumenta [2], este procedimiento depende de la tipología del brazo y de su número de grados de libertad.

Existen varias formas de abordar la cinemática inversa de un robot. Entre otros, se han desarrollado procedimientos genéricos programables, en los cuales a partir de los parámetros de *Denavit-Hartenberg*, se pueden obtener los valores articulares que consiguen la posición y orientación deseada. Pero éstos, son métodos iterativos que en muchos casos necesitan una velocidad de procesamiento muy grande, lo cual puede causar problemas de capacidad de computación, además de no generar respuestas en tiempo real, lo que sería un problema en el control del robot.

Generalmente, para solucionar la cinemática inversa no se usan técnicas iterativas, sino que se buscan relaciones matemáticas. Si tenemos relaciones matemáticas que definan la cinemática inversa, podremos encontrar soluciones en tiempo real, que nos servirán para controlar el robot. Además, normalmente la cinemática inversa tiene múltiples soluciones; con esas relaciones matemáticas, podremos elegir cual es la solución que de verdad se adapta nuestra aplicación, simplemente imponiendo restricciones en el uso de la relación. Los métodos que podemos utilizar para resolver el problema cinemático inverso son, el método algebraico y el método geométrico.

- **Método algebraico**

El método algebraico se basa, en buscar una solución a partir de las matrices de transformación homogénea. Por tanto, el objetivo será, una vez conocida la matriz homogénea total "T", hallar las coordenadas articulares.

El método algebraico suele ser de aplicación bastante compleja. Además, en robots con muchos grados de libertad la complejidad aumenta exponencialmente, por ejemplo, en un robot de seis grados de libertad, tendríamos doce ecuaciones, de las

cuales tendríamos que elegir con que seis operar. Esa elección se debe realizar con sumo cuidado.

- **Método geométrico**

El método geométrico se basa en las relaciones trigonométricas extraídas directamente de la geometría del brazo. Debemos encontrar las suficientes relaciones geométricas como para poder resolver la cinemática directa. Aun así, este método solo es recomendado para robots de pocos grados de libertad o para simplemente hallar las coordenadas de las primeras articulaciones. Debido a que es bastante complejo encontrar suficientes relaciones geométricas en robots de más de tres grados de libertad.

Por eso mismo, existe un procedimiento geométrico, llamado desacoplo cinemático que permite hallar con mayor facilidad las coordenadas de todas las articulaciones.

- **Desacoplo cinemático**

Con los métodos comentados hasta ahora, se pueden hallar las coordenadas de las tres primeras articulaciones. Pero se complica el hallazgo de las coordenadas del resto de las articulaciones. La mayoría de robots industriales, suelen tener tres grados de libertad más, a parte de los tres primeros. Normalmente, las tres últimas articulaciones del robot conforman la muñeca del robot, articulaciones en las cuales sus ejes se cruzan.

La posición del final del final de la muñeca, cambia respecto a la posición donde empieza la muñeca, pero la orientación es la misma, en ambos lugares. Por tanto, si calculamos la orientación de las primeras articulaciones, hasta llegar a la muñeca, estaremos calculando la orientación final del extremo del robot. De eso mismo se aprovecha el desacoplo cinemático. En resumen, el desacoplo cinemático parte el problema en dos, uno el de hallar la orientación y otro, el de hallar la posición.

Primero se omitirá la parte del robot correspondiente a la muñeca, calculándose los calores de las tres primeras variables articulares, como si no existiera la muñeca. Con eso, ya sabemos la orientación final que tendrá el extremo del robot. A partir de ese dato de orientación y de las coordenadas de las primeras articulaciones, calcularemos los valores del resto de las articulaciones, que conforman la muñeca.

3.1.2. Planificación de trayectorias

Referenciándonos en [2], el control cinemático en la robótica plantea varios conceptos:

- *Path planning* (planificación de trayectorias)
- *Trajectory generation* (generación de referencias)
- *Trajectory tracking* (control de la trayectoria)

La planificación de trayectorias, o como se le suele llamar por su vocablo inglés *path planning*, es una técnica que consiste en describir el movimiento deseado del manipulador, con una secuencia de puntos, que permita al robot ir desde un punto inicial hasta un objetivo deseado. En esa ruta, el proceso de planificación debe asegurarse de evitar colisiones y de tener en cuenta las restricciones que se impongan sobre las articulaciones del robot.

La generación de referencias es el proceso mediante el cual se calculan las trayectorias que debe seguir cada articulación como función del tiempo. Ese cálculo debe tener en cuenta, el cumplimiento de alguna restricción de tiempo, alcanzar la máxima manejabilidad durante el movimiento y minimizar el costo de movimiento.

Por su parte, el control de la trayectoria, como su nombre indica consiste en llevar un seguimiento sobre el robot de la correcta realización de la trayectoria. Y en caso de que existan errores en la realización, aplicar las acciones correctoras oportunas.

Para ir de un punto a otro, se pueden utilizar múltiples tipos de trayectorias. Pero con la intención de establecer unos cánones que puedan seguir sobre todo los fabricantes de robots, existen tres tipos de trayectorias diferenciadas.

- **Trayectorias punto a punto**

Cada articulación del robot, realiza la trayectoria desde un punto inicial hasta el punto objetivo, independientemente del estado de las demás articulaciones. Las trayectorias punto a punto se pueden dar de dos formas distintas, movimientos eje a eje y movimiento simultáneo de ejes.

En los movimientos eje a eje, sólo se mueve una articulación en cada instante de tiempo, lo cual hace que el tiempo total de la trayectoria aumente notablemente, lo que hace que este movimiento no sea utilizado en la robótica industrial.

En el movimiento simultáneo de ejes, las articulaciones inician su trayectoria de movimiento al mismo tiempo y a máxima velocidad. El tiempo total de ejecución de la trayectoria se disminuye notablemente, pero el consumo aumenta, además de que todas las articulaciones no terminan el movimiento al mismo tiempo. Ciertamente, este tipo de movimiento tampoco es usado en la robótica industrial.

- **Trayectorias coordinadas o isócronas**

En este tipo de trayectorias las articulaciones se mueven siempre al mismo tiempo, el movimiento de todas las articulaciones comienza y finaliza en el mismo instante de tiempo. Todas las articulaciones se ajustan a la articulación que tenga la trayectoria más larga, la cual trabajará a su máxima velocidad. Este tipo de trayectorias evita los posibles problemas que puede causar, que unas articulaciones se muevas mientras otras ya han terminado su movimiento.

- **Trayectorias continuas**

Estas trayectorias siguen movimientos en el espacio de tares o en el espacio cartesiano, movimientos que son conocidos por el usuario. Esta trayectoria la sigue el extremo del robot y puede consistir, por ejemplo, en trayectorias rectilíneas a velocidad y orientación constante, en trayectorias circulares entre tres puntos o en trayectorias circulares con reorientación.

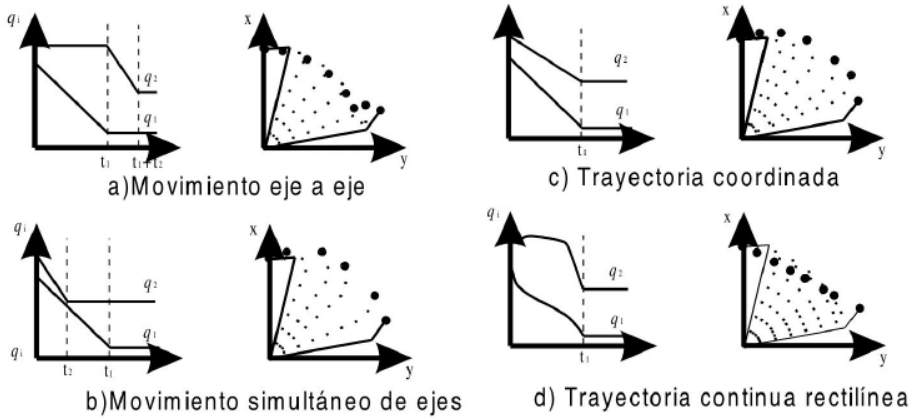


Imagen 17-Tipos de trayectorias

Además, también existen diferentes algoritmos de planificación de trayectorias, y dependiendo de la aplicación que necesitemos, podremos utilizar un algoritmo u otro. Vamos a comentar tres de los algoritmos más importantes:

- **Algoritmo A* o *A-star***

Este algoritmo permite obtener un camino, libre de obstáculos, que conecta el punto inicial con el final, en base a una serie de pesos. Es eficiente en la búsqueda de trayectorias óptimas en entornos estáticos. Se puede resumir en un proceso de búsqueda rápido. Es muy usada en robots móviles y en sistemas de logística.

- **Algoritmo D* o *Dynamic A-star***

El algoritmo D* o *Dynamic A-star,* tiene una gran capacidad de replanificación eficiente, lo que hace sea un algoritmo muy adecuado para entornos dinámicos, en los que puedan aparecer nuevos obstáculos o puedan moverse los presentes. Son de uso en aplicaciones industriales como sistemas de seguridad o robots colaborativos, que necesitan adaptabilidad en tiempo real.

- **Algoritmo RRT o *Rapidly-exploring Random Trees***

El algoritmo RRT se adapta muy bien a entornos complejos y desconocidos, lo que permite que sea utilizado principalmente para la generación de trayectorias en tiempo real. Este algoritmo se basa en la generación de un árbol de configuraciones que se expande explorando desde el punto de origen.

3.1.2.1. Planificación de trayectorias y cinemática UR5

Todas las operaciones relacionadas con la planificación de trayectorias y el control cinemático del robot UR5, en las aplicaciones relacionadas con este proyecto, se realizarán por medio del software *MoveIt*. Éste es un paquete de ROS, que facilita la planificación de trayectorias, la cinemática y el control del robot, entre otros. Del software *MoveIt* hablaremos más profundamente en el apartado 4.2.

3.1.3. Espacio de trabajo

El espacio de trabajo de un robot es un dato clave a la hora de saber si ese robot es compatible en la aplicación que queremos implementar. El espacio de trabajo es el volumen espacial que puede abarcar el extremo del robot. Viene determinado por la geometría del robot, así como por las limitaciones en las articulaciones del mismo.

Aunque todos los puntos a los que debemos alcanzar en nuestra aplicación, estén dentro del espacio de trabajo, hay que considerar otros posibles problemas que pueden aparecer en la implementación de nuestra aplicación. Hay que tener en cuenta que no todos los puntos del espacio de trabajo, tienen que ser alcanzables por el robot con cualquier configuración articular, habrá ciertas posiciones dentro del espacio de trabajo, la cuales se podrán alcanzar solamente con un número de configuraciones limitadas. Además, se debe tener en cuenta la presencia de puntos singulares en el espacio de trabajo, que nos pueden perturbar la trayectoria de nuestra aplicación.

Por tanto, para elegir si el robot es aceptable para un uso concreto según su espacio de estado, hay que tener varias cosas en cuenta. Las restricciones articulares y los puntos singulares hay que tenerlos muy en cuenta, así como los objetos que estén presentes cerca del robot y puedan representar un obstáculo.

3.1.3.1. Espacio de trabajo del UR5

Según su manual de usuario [4], el espacio de trabajo del robot UR5 ocupa 850mm desde la junta de la base. La representación que aparece en dicho manual, del espacio de trabajo del UR5 es la siguiente.

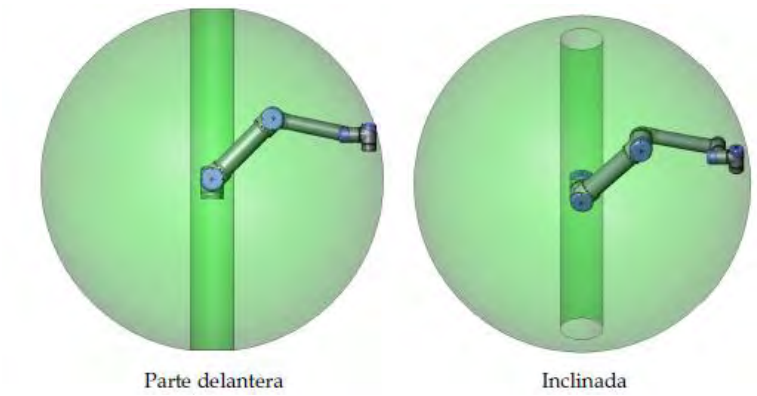


Imagen 18-Espacio de trabajo UR5

Como podemos ver, el espacio de trabajo lo conforma una esfera de radio 850mm. Atravesando la esfera hay un cilindro, este cilindro indica que esa área de dentro del cilindro, no forma parte del espacio de trabajo.

En el manual de usuario [4], no vienen especificadas las dimensiones del cilindro que está fuera del espacio de trabajo. Simplemente, dicho manual nos comenta que evitemos acercar la herramienta al volumen cilíndrico en la medida de los posible, ya que esto provocaría que el robot trabaje de forma ineficiente y dificultando la realización de la evaluación de riesgos.

3.1.4. Parámetros de carga

La carga útil de un robot, es la carga o peso máximo que puede soportar un robot dentro del espacio de trabajo del mismo, en su elemento terminal del manipulador. Así como el espacio de trabajo, esta característica es una de las más importantes a la hora de seleccionar un robot que se adapte a nuestra aplicación. Por ejemplo, para operaciones donde los objetos a transportar sean muy pesados, tendremos que escoger robots con una alta carga útil, como puede ser un robot de paletizado.

3.1.4.1. Parámetros de carga del UR5

En el manual de usuario del UR5 [4], nos dan el parámetro de carga con un diagrama de carga máxima frente a la compensación del centro de gravedad. Donde la compensación del centro de gravedad, según el propio manual es, “la distancia entre el centro de la brida de salida de emergencia y el centro de gravedad.

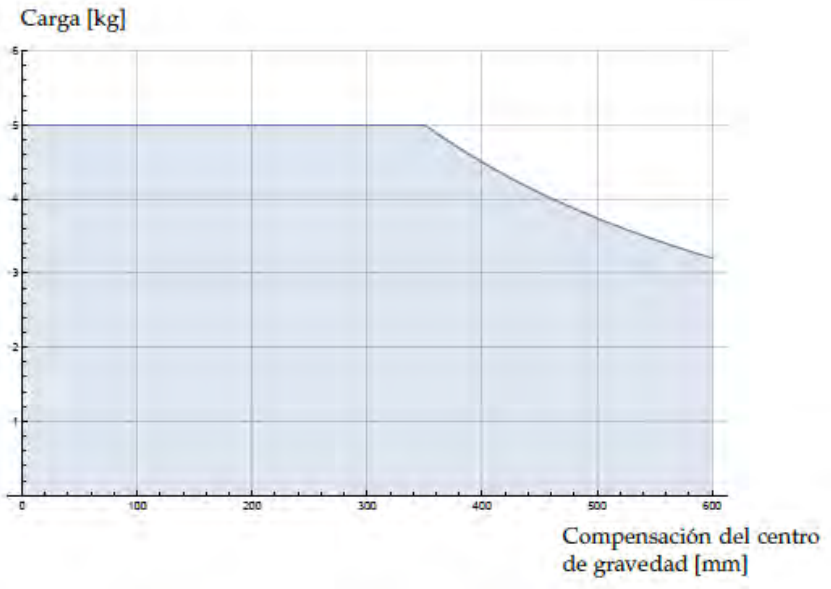


Imagen 19-Diagrama parámetros de carga

Podemos observar en el diagrama, que cuando el efector final está cerca de la base del robot, puede aguantar hasta 5 kg de peso. Pero aun así debemos tener cuidado

si utilizamos el UR5, porque si alejamos el centro de gravedad de la brida de emergencia, lo que quiere decir que estiramos el brazo, a más de 350 mm, el peso que soporta el robot baja, llegando a soportar 3.2 kg cuando tenemos el brazo estirado.

Por tanto, al planificar una aplicación con este robot, debemos tener en cuenta que, si el brazo se va a estirar bastante, deberíamos utilizar objetos con peso menor a 3.2 kg para prevenir daños en el brazo robótico.

3.2. Visión por computador

Todos los fundamentos teóricos que se van a desarrollar a continuación, están referenciados en [3].

3.2.1. Espacios de color y sus transformaciones

Los espacios o modelos de color son sistemas de coordenadas tridimensionales, en los que cada punto representa un color específico, y tienen el propósito de representar los colores de alguna forma estándar. La amplia mayoría de estos modelos se han desarrollado para aplicaciones específicas. Por ejemplo, uno de los espacios más utilizados como es el espacio de color RGB, se utiliza principalmente en la adquisición de imágenes. Mientras que el espacio XYZ es útil si nos interesa medir un color. Otros espacios de color como HSI y HSV son útiles a la hora de segmentar objetos por color.

Existen muchos espacios de color y cada uno tiene ventajas e inconvenientes, por tanto, tenemos que elegir cual es el espacio de color que mejor nos convenga en cada momento. Para ello existen operaciones que transforman una imagen representada en un espacio de color a su equivalente en otro espacio. Debido a que las imágenes se adquieren normalmente en el modelo RGB, las conversiones se suelen realizar desde el modelo RGB a cualquier otro deseado. Por ejemplo, la operación matricial que convierte del espacio RGB al espacio XYZ es la siguiente.

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 2.739 & -1.145 & -0.424 \\ -1.119 & 2.029 & 0.033 \\ 0.138 & -0.333 & 1.105 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Imagen 20-Conversión de RGB a XYZ

Para convertir una imagen en modelo RGB a modelo HSI se debe realizar lo siguiente.

$$I = \frac{1}{3}(R + G + B)$$

$$S = 1 - \frac{3}{R + G + B}[\min(R, G, B)]$$

$$H = \cos^{-1}\left\{\frac{1/2[(R - G) + (R - B)]}{[(R - G)^2 + (R - B)(G - B)]^{1/2}}\right\}$$

Imagen 21-Conversión de RGB a HSI

Para realizar el algoritmo de visión artificial de este proyecto, nos conviene trabajar con la imagen en el espacio de color HSV. La representación tridimensional del modelo HSV es la siguiente.

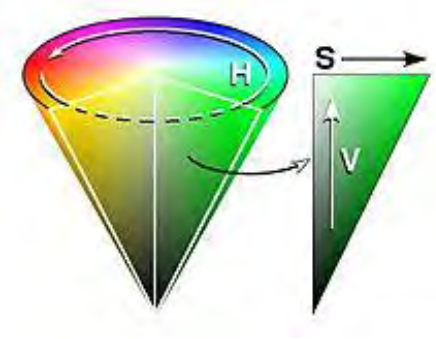


Imagen 22-Representación tridimensional HSV

La ventaja principal de este espacio de color, es que el canal H o *Hue*. En este canal se encuentran escalados todos los tonos de color perceptibles. Esa condición es perfecta para detectar objetos por color. En apartados posteriores, se mostrará como utilizaremos esa condición para detectar los distintos objetos.

En *Python*, gracias a la librería llamada *OpenCV*, se pueden realizar operaciones de conversión entre espacios de color utilizando el siguiente código de programación.

cv2.cvtColor(src, code[, dst[, dstCn]])

Donde *src* es la imagen que queremos transformar y con *code* indicamos la transformación que queremos realizar. Los otros dos parámetros son condicionales, en nuestro caso no los utilizaremos.

3.2.2. Segmentación

El objetivo de la segmentación es dividir la imagen en regiones con significado, para obtener una mejor comprensión de estas. Existen varias maneras de segmentar una imagen, en el caso que nos atañe en este proyecto utilizaremos la umbralización.

La funcionalidad de la umbralización consiste en separar los objetos por un valor de gris característico, llamado umbral. El objetivo de las operaciones de umbralización es encontrar el umbral que mejor separe un objeto del fondo o incluso varios objetos entre sí. Hay operaciones que utilizan un solo umbral, pero también existen otras que utilizan varios umbrales para separar dos o más de dos zonas de gris.

Para realizar el algoritmo de visión de este proyecto, necesitamos segmentar la imagen por color. Existe una función de *OpenCv*, con la cual podemos segmentar por color.

cv2.inRange (src, lowerb, upperb)

Dicha función trabaja en el espacio de color HSV, ya que en este espacio se puede segmentar fácilmente el color accediendo al canal *Hue*.

Esta función realiza la segmentación de los distintos objetos de la imagen, utilizando el rango de valores de los píxeles que corresponda al color que se quiere segmentar.

En esa función insertamos los parámetros *src* que es la imagen a segmentar, *lowerb* es el límite inferior del rango de colores que pertenecen al objeto que queremos segmentar y *upperb* corresponde al límite superior.

3.2.3. Visión 3D

Como se expone en [3], el problema central de la visión 3D es, a partir de una imagen o de una secuencia de imágenes de un objeto, tomadas de forma monocular o policular, comprender cuales son los objetos de la escena, su ubicación y sus propiedades tridimensionales. Por tanto, la funcionalidad central de la visión 3D es conseguir que, a partir de una imagen bidimensional con coordenadas en pixeles, poder averiguar cuáles son las coordenadas tridimensionales de los objetos, en unidades métricas.

Existen varias técnicas o métodos para obtener la percepción tridimensional de las imágenes, algunos de estos son métodos activos, los cuales modifican de alguna forma la escena o la cámara, y otros son pasivos. Pero todos estos métodos se basan en el principio de la Triangulación Espacial. Este principio se fundamenta en la geometría de un triángulo, y nos dice que, si conocemos tres de los parámetros, podemos conocer cualquier otro parámetro del triángulo.

$$L1\cos(a)+L2\cos(b)=D$$

$$L1\,\mathrm{sen}(a)=L2\,\mathrm{sen}(b)$$

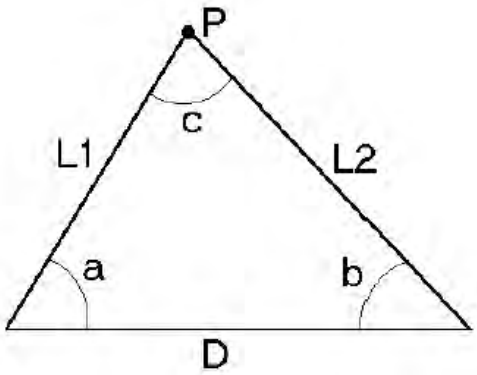


Imagen 23-Triangulación espacial

Existen múltiples técnicas de recuperación de la estructura 3D de la escena, por ejemplo, tenemos métodos acticos como la técnica del tiempo de vuelo o la proyección de luz estructurada. También tenemos técnicas que utilizan una sola cámara, y la van

moviendo, como es el flujo óptico y otras que utilizan un par de cámaras, como es la técnica del par estereoscópico.

En la aplicación de visión artificial, que hemos desarrollado en este proyecto, tenemos un sistema con una cámara. El modelo matemático que utilizaremos para obtener, a partir de la imagen bidimensional adquirida de la escena, la pose tridimensional de los objetos de esa escena, es el modelo *Pin-Hole*.

3.2.3.1. Modelo de Lente *Pin-Hole*

El modelo *Pin-Hole* no es el modelo que se sigue en la construcción de las cámaras actuales, pero su modelo matemático sí que es capaz de representar el funcionamiento de la mayoría de las cámaras sea cual sea su diseño. El modelo de proyección de Lente *Pin-Hole* sigue la siguiente representación.

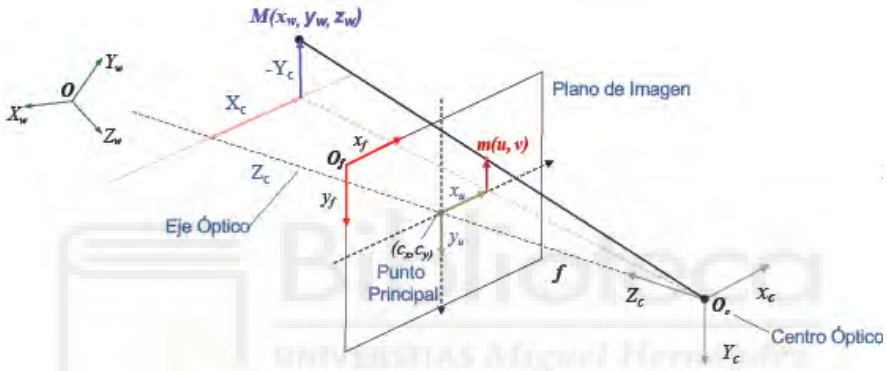


Imagen 24-Representación gráfica modelo Pin-Hole

Los sistemas de coordenadas implicados en el modelo *Pin-Hole* son los siguientes:

- **Coordenadas del mundo:** $(O; X_w, Y_w, Z_w)$
- **Coordenadas de la cámara:** $(O_c; X_c, Y_c, Z_c)$
- **Coordenadas centrales de la imagen:** $(C_x, C_y; x_u, y_u)$
- **Coordenadas centrales de imagen normalizadas:** $(f = 1)(C_x, C_y; x_n, y_n)$
- **Coordenadas laterales de la imagen:** $(O_f; x_f, y_f)$ ó (u, v)

Con la representación gráfica del modelo *Pin-Hole* y los sistemas de coordenadas descritos, se pueden especificar las ecuaciones de proyección. Por triángulos semejantes, podemos obtener las dos ecuaciones de proyección del modelo *Pin-Hole*.

$$\frac{x_u}{f} = \frac{X_c}{Z_c}$$
$$\frac{y_u}{f} = \frac{Y_c}{Z_c}$$

Imagen 25-Ecuaciones proyectivas Pin-Hole

Ahora podemos hallar la relación entre las coordenadas proyectivas centrales de la imagen en función de las coordenadas proyectivas de la cámara.

$$\begin{bmatrix} \lambda x_u \\ \lambda y_u \\ \lambda \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$

Imagen 26-Relación coordenadas centrales imagen y coordenadas cámara

Se introduce un parámetro landa a la relación anterior, porque las coordenadas proyectivas están definidas a un factor de escala. Como las coordenadas están definidas a un factor de escala, podemos multiplicar la expresión por la distancia focal y las coordenadas se mantendrán igual.

$$\begin{bmatrix} n x_u \\ n y_u \\ n \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$

Imagen 27-Relación coordenadas centrales imagen y coordenadas cámara

Esta expresión también nos da la matriz de proyección.

$$\tilde{A}_0 \cong [A_0 \quad 0] \qquad A_0 \cong \begin{bmatrix} f & 0 & 0 \\ 0 & f & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Imagen 28-Matriz de proyección

Podemos normalizar las coordenadas, considerando que f = 1. Estas coordenadas del centro de la imagen normalizadas tendrían la siguiente forma.

$$\begin{bmatrix} \lambda x_n \\ \lambda y_n \\ \lambda \end{bmatrix} = [I \quad 0] \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} \qquad \begin{aligned} x_n &= \frac{X_c}{Z_c} \\ y_n &= \frac{Y_c}{Z_c} \end{aligned}$$

Imagen 29-Coordenadas centrales imagen normalizadas

Estas coordenadas normalizadas son útiles a la hora de modelar la distorsión. En muchos sistemas de visión, es necesario modelar esa distorsión de la imagen. La distorsión sigue el siguiente modelo.

Radial Tangencial Prismática

$$\begin{aligned} D_x(x_n, y_n) &= k_1 r^2 x_n + k_2 r^4 x_n + k_3 r^6 x_n + 2p_1 x_n y_n + p_2(2x_n^2 + r^2) + s_1 r^2 + s_2 r^4 \\ D_y(x_n, y_n) &= k_1 r^2 y_n + k_2 r^4 y_n + k_3 r^6 y_n + 2p_2 x_n y_n + p_1(2y_n^2 + r^2) + s_1 r^2 + s_2 r^4 \end{aligned}$$

Imagen 30-Modelado de la distorsión

Y ahora, podemos hallar las coordenadas centrales de la imagen distorsionadas, y con ello estaremos introduciendo la distorsión a las coordenadas del modelo *Pin-Hole*.

$$x_d = x_u + f \cdot D_x(x_n, y_n)$$
$$y_d = y_u + f \cdot D_y(x_n, y_n)$$

Imagen 31-Coordenadas centrales imagen distorsionadas

Ahora veremos el resto de relaciones entre los sistemas de coordenadas, con el objetivo de obtener las ecuaciones proyectivas de una imagen. Primero, atendiendo a la representación de la imagen 24, la relación entre las coordenadas del mundo y las coordenadas de la cámara es la siguiente.

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} R_c^w & t_c^w \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

Imagen 32-Relación entre coordenadas de la cámara y del mundo

La siguiente etapa es pasar de coordenadas de cámara a coordenadas centrales de la imagen. Como este caso es una proyección, ya que pasamos de coordenadas 3D a coordenadas 2D, utilizaremos la matriz de proyección, de la imagen 28, para realizar la transformación.

$$\begin{bmatrix} n\,x_u \\ n\,y_u \\ n \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$

Imagen 33-Relación entre coordenadas de la cámara y centrales de la imagen

Una vez tenemos las coordenadas centrales de la imagen, podemos añadir la distorsión debida a la deformación de la imagen. Para ello, calculamos las coordenadas centrales de la imagen distorsionadas, según lo dispuesto en la imagen 31. El último paso, es trasformar las coordenadas centrales en mm, a coordenadas laterales en pixeles teniendo en cuenta las dimensiones que tengan los píxeles de nuestro sensor.

Para realizar esta transformación, modelaremos una transformación euclídea en la que se realice un desplazamiento *Cx* y *Cy*, acompañado por un escalado *Kx* y *Ky*. Esta transformación vendrá definida de la siguiente manera.

$$\begin{aligned} x_f &= K_x x_d + C_x \\ y_f &= K_y y_d + C_y \end{aligned} \Rightarrow \begin{bmatrix} n\,x_f \\ n\,y_f \\ n \end{bmatrix} = \begin{bmatrix} K_x & 0 & C_x \\ 0 & K_y & C_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} n\,x_d \\ n\,y_d \\ n \end{bmatrix}$$

Imagen 34-Relación entre coordenadas centrales de la imagen y las laterales de la imagen

Ya tenemos todas las ecuaciones proyectivas. Ahora podemos combinar estas, con la intención de encontrar la relación matricial entre las coordenadas del mundo y

las coordenadas laterales de la cámara. Esta relación, sin tener en cuenta la distorsión, será la siguiente.

$$\begin{bmatrix} n\,x_f \\ n\,y_f \\ n \end{bmatrix} = \begin{bmatrix} K_x f & 0 & C_x \\ 0 & K_y f & C_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ 0 \\ 1 \end{bmatrix}$$

Imagen 35-Relación entre las coordenadas del mundo y las laterales de la imagen

Este es el modelo proyectivo matemático *Pin-Hole*. Nos permite tener una relación para pasar directamente de coordenadas de la imagen a coordenadas del mundo de forma proyectiva. Considerando $f_x = K_x * f \;\; y \;\; f_y = K_y * f$, podemos considerar que la matriz de proyección de cámara o de parámetros intrínsecos de la cámara, A, es la siguiente.

$$A = \begin{bmatrix} f_x & 0 & C_x \\ 0 & f_y & C_y \\ 0 & 0 & 1 \end{bmatrix}$$

Imagen 36-Matriz de parámetros intrínsecos

Tendríamos así, el modelo compacto proyectivo completo, que nos representa la proyección de un punto 3D en un punto 2D proyectivo.

$$\widetilde{m} \cong A \; [R_c^w \quad t_c^w] \; \widetilde{M}$$

Imagen 37-Modelo proyectivo Pin-Hole compacto

Tanto la matriz A, de parámetros intrínsecos de la cámara, como la matriz de rotación y translación, serán muy importantes a la hora de calibrar la cámara. Esto se debe a qué, hallando esos valores, podremos transformar un punto de la imagen en píxeles a un punto 3D en coordenadas del mundo.

Por último, si desarrollamos la matriz de proyección.

$$\begin{bmatrix} n\,x_f \\ n\,y_f \\ n \end{bmatrix} = \begin{bmatrix} K_x f\,\bar{r}_1 + C_x\,\bar{r}_3 & K_x f\,t_x + C_x\,t_z \\ K_y f\,\bar{r}_2 + C_y\,\bar{r}_3 & K_y f\,t_y + C_y\,t_z \\ \bar{r}_3 & t_z \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

Imagen 38-Matriz de proyección desarrollada

Esta expresión puede ser útil a la hora de calibrar, cuando nos planteemos como obtener los coeficientes internos de la proyección, teniendo varias equivalencias entre puntos de la imagen y puntos del mundo.

Si añadimos la distorsión el modelo ya no va a ser lineal, pero aun así vamos a mantener una expresión linealizada. Para ello, primero hallamos las coordenadas normalizadas.

$$\tilde{q} \cong A^{-1} \cdot \tilde{m}_u \cong \begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix}$$

Imagen 39-Coordenadas centrales imagen normalizadas

Y con las coordenadas normalizadas, calculamos las coordenadas con la distorsión.

$$\left.\begin{array}{c} x_n^d = x_n(1 + k_1r^2 + k_2r^4 + k_3r^6) + 2p_1x_ny_n + p_2(2x_n^2 + r^2) + s_1r^2 + s_2r^4 \\ y_n^d = y_n\,(1 + k_1r^2 + k_2r^4 + k_3r^6) + 2p_2x_ny_n + p_1(2y_n^2 + r^2) + s_1r^2 + s_2r^4 \\ r^2 = x_n^2 + y_n^2 \end{array}\right\} \quad \tilde{q}^d \cong \begin{bmatrix} x_n^d \\ y_n^d \\ 1 \end{bmatrix}$$

Imagen 40-Coordenadas centrales distorsionadas

Y con esas coordenadas de distorsión, podríamos obtener la relación con las coordenadas 2D, de la siguiente manera.

$$\tilde{m} \cong A \cdot \tilde{q}^d$$

Imagen 41-Relación con las coordenadas 2D distorsionadas

3.2.3.2. Calibración cámara

La calibración consiste en la determinación de los parámetros que permiten modelar la proyección de un punto 3D en un punto de imagen. En este proceso, se deben encontrar tanto los parámetros intrínsecos como los extrínsecos. Los intrínsecos son, los factores de escala *Kx* y *Ky*, la distancia focal f, el punto principal *Cx* y *Cy*, y la distorsión *Dx* y *Dy*. Los parámetros extrínsecos hacen referencia al vector de translación y la matriz de rotación de la ecuación de proyección del modelo *Pin-Hole*.

La calibración nos permite, a través de un punto de imagen, obtener el rayo de proyección de los puntos 3D y para ello, el proceso calibración sigue una serie de etapas. La primera etapa es, plantear las ecuaciones del sistema, que son las que hemos visto en el apartado anterior, el 3.2.3.1. Seguidamente se obtendrán, para ciertos puntos 3D de la escena, cuáles son sus proyecciones en la imagen. Y la última etapa, es la determinación de los parámetros de calibración.

Cabe recordar, que el modelo proyectivo que utilizaremos en la calibración, es el modelo *Pin-Hole*. Más concretamente, la expresión que utilizaremos para estimar los parámetros y que relaciona las coordenadas del mundo, con las coordenadas laterales de la imagen, es la siguiente.

$$\begin{bmatrix} n\,x_f \\ n\,y_f \\ n \end{bmatrix} = \begin{bmatrix} K_x f\,\bar{r}_1 + C_x\,\bar{r}_3 & K_x f\,t_x + C_x\,t_z \\ K_y f\,\bar{r}_2 + C_y\,\bar{r}_3 & K_y f\,t_y + C_y\,t_z \\ \bar{r}_3 & t_z \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

Imagen 42-Relación coordenadas mundo y laterales de la imagen

Una vez, tenemos claro que vamos a usar el modelo *Pin-Hole*, debemos tomar tantos datos como sean necesarios, para realizar la calibración. Buscaremos puntos de imagen, de los cuales conozcamos sus coordenadas 3D.

En la práctica, normalmente se usan patrones de calibración para facilitar esa tarea, normalmente patrones blancos y negros con esquinas fácilmente reconocibles. Un ejemplo de patrón de calibración sería el siguiente.

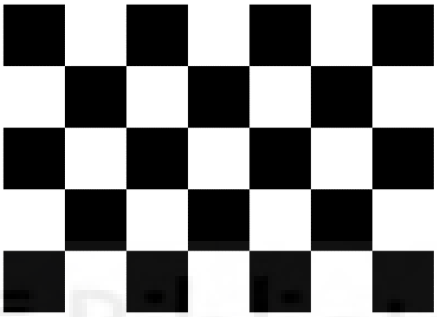


Imagen 43-Ejemplo de patrón de calibración

Una vez obtengamos las matrices de puntos del patrón tanto en 3D como sus relativos en 2D, nos quedará hallar los parámetros de calibración. Realmente, no sabremos a ciencia cierta si esos parámetros son del todo correctos o no. Para saber si hemos realizado bien la calibración y, por tanto, son correctos los parámetros calculados, se suele utilizar el error de reproyección.

El cálculo de ese error consiste en, dado un punto en el que se conoce tanto su coordenada 3D como su coordenada de imagen, se debe hallar la coordenada 2D utilizando los parámetros de calibración y luego compararla con la coordenada 2D real. A más cercano de cero sea ese error, mejor calibrada estará la cámara.

$$e = \sum_{i} \left(\hat{x}_f^{(i)} - x_f^{(i)}\right)^2 + \left(\hat{y}_f^{(i)} - y_f^{(i)}\right)^2$$

Imagen 44-Error de reproyección

Existen múltiples técnicas y métodos de calibración, los métodos relacionados con la robótica se caracterizan por ser rápidos y autónomos, trabajar con imágenes con menor resolución y por verse afectados por muchos factores que de forma sistemática causan errores.

En robótica se utilizan diversos métodos, como puede ser el método de transformación lineal directa, la restricción de alineamiento radial. Pero por encima de todos, hoy en día el método más utilizado es el método de Zhang [5].

La idea básica del método de Zhang es realizar la calibración utilizando múltiples vistas, de un mismo patrón plano. Se capturarán imágenes sucesivas de un mismo patrón moviendo y girando éste, delante de la cámara. Con esas sucesivas imágenes, se buscarán varias correspondencias, que consiste en encontrar un mismo punto 3D concreto, en las imágenes 2D. A partir de dichas correspondencias se calcularán primero los parámetros intrínsecos de la cámara y, en segundo lugar, se calculará la posición y rotación de esos todos los planos respecto a la cámara.

El inconveniente principal de este método, es que aumenta el tiempo de procesamiento, ya que implica realizar dos estimaciones separadas. En cambio, tiene diversas ventajas como simplificar el elemento de calibración, también fuerza el desacople de los parámetros intrínsecos y incorpora más puntos de control distribuidos en más planos.

Las etapas a seguir en el método de Zhang, son las siguientes.

- **Cálculo de la homografía entre imagen y objeto**
- **Cálculo de parámetros intrínsecos con múltiples vistas**
- **Cálculo de los parámetros extrínsecos para cada vista**
- **Optimización global considerando distorsión**

El cálculo de la homografía consiste básicamente en hallar las relaciones entre los puntos de los patrones 3D y los puntos de los patrones de las imágenes 2D. Si colocamos el eje de coordenadas del mundo, sobre el patrón, podremos asumir que *Zw* siempre va a ser cero.

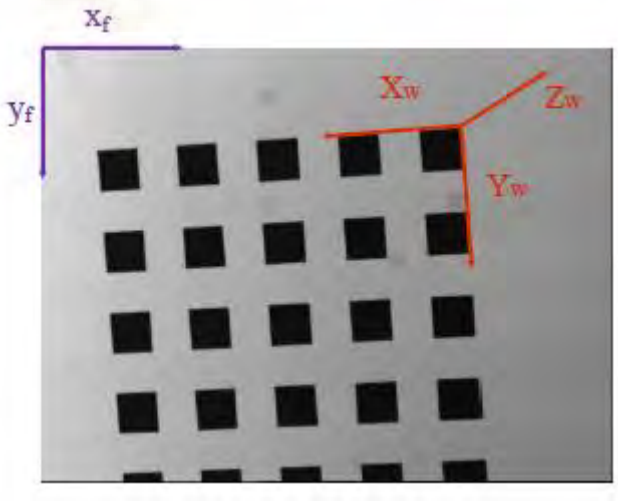


Imagen 45-Patrón de calibración con los ejes implicados

El modelo sin distorsión quedaría de la siguiente manera.

$$\begin{bmatrix} n\,x_f \\ n\,y_f \\ n \end{bmatrix} = \begin{bmatrix} K_x f & 0 & C_x \\ 0 & K_y f & C_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ 0 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} n\,x_f \\ n\,y_f \\ n \end{bmatrix} = \begin{bmatrix} K_x f & 0 & C_x \\ 0 & K_y f & C_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ r_{31} & r_{32} & t_z \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix}$$

Imagen 46-Modelo Pin-Hole sin distrosión

Podemos unir las dos matrices de parámetros, en una sola matriz "H", que será la matriz de homografía.

$$\begin{bmatrix} n \cdot x_f \\ n \cdot y_f \\ n \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ 1 \end{bmatrix}$$

Imagen 47-Relación homográfica

La matriz "H" tiene la forma, $H = \lambda A\ [r_1\ \ r_2\ \ t]$. Una vez tenemos establecida la matriz de homografía y la relación homográfica, podemos empezar con el proceso de calibración.

Primero se procedería con el cálculo de la homografía, que consiste en calcular, para una serie de puntos 3D conocidos, su respectiva posición en un conjunto de imágenes.

De la relación homográfica de la imagen 47, sacamos dos restricciones.

$$(h_{31}X_w + h_{32}Y_w + h_{33}) \cdot x_f = h_{11}X_w + h_{12}Y_w + h_{13}$$
$$(h_{31}X_w + h_{32}Y_w + h_{33}) \cdot y_f = h_{21}X_w + h_{22}Y_w + h_{23}$$

Imagen 48-Restricciones homografía

Por tanto, sabemos que para cada coordenada 3D, obtenemos dos ecuaciones. Como sabemos, en la matriz "H" de homografía tenemos nueve incógnitas, pero como dicha matriz viene definida a un factor de escala concreto, solo tendremos ocho grados de libertad en la matriz. Eso significa que solo necesitaremos ocho ecuaciones, para encontrar los parámetros de la matriz homográfica, lo que implica la necesidad de conocer al menos cuatro correspondencias de puntos. Aunque tenemos que saber que, a más correspondencias, más exacto es el cálculo de la homografía.

Una vez formulemos al menos ocho de las ecuaciones posibles, podemos crear un sistema de ecuaciones, en el que las incógnitas son los valores de la homografía, y los datos son las correspondencias entre los valores de las coordenadas del mundo y de la imagen. En formato matricial, el sistema tendría la siguiente forma.

$$\mathbf{A}_{xy} = \begin{bmatrix} -X_w^{(1)} & -Y_w^{(1)} & -1 & 0 & 0 & 0 & X_w^{(1)}x_f^{(1)} & Y_w^{(1)}x_f^{(1)} & x_f^{(1)} \\ 0 & 0 & 0 & -X_w^{(1)} & -Y_w^{(1)} & -1 & X_w^{(1)}y_f^{(1)} & Y_w^{(1)}y_f^{(1)} & y_f^{(1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \end{bmatrix}$$

$$\mathbf{h} = [h_{11}, h_{12}, h_{13}, h_{21}, h_{22}, h_{23}, h_{31}, h_{32}, h_{33}]^T$$

$$\mathbf{A}_{xy} \cdot \mathbf{h} = 0$$

Imagen 49-Sistema ecuaciones homografía

Y este sistema de ecuaciones lo podemos resolver con mínimos cuadrados, realizando la descomposición.

$$SVD(A_{xy}) \rightarrow A_{xy} = UDV^T \rightarrow h = v_9$$

Donde *U* es una matriz ortogonal de vectores ortogonales, *D* es una matriz diagonal de valores singulares y *V* una matriz de vectores singulares. La solución será el vector *h* que cumpla ser el noveno vector singular de la matriz *V*, el cual está asociado al último valor singular de la matriz D. De este modo, encontramos el vector de solución que es lo más ortogonal posible a A_{xy}. Una vez hecho esto, ya hemos completado el cálculo de la homografía.

Una vez se ha calculado la homografía, se puede pasar al cálculo de los parámetros intrínsecos. Si recordamos, la expresión de homografía era $H = \lambda A\ [r_1\ \ r_2\ \ t]$. Que se puede expresar también matricialmente.

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} = \lambda \begin{bmatrix} K_x f & 0 & C_x \\ 0 & K_y f & C_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & t_x \\ r_{21} & r_{22} & t_y \\ r_{31} & r_{32} & t_z \end{bmatrix}$$

Imagen 50-Expresión matricial de H

Si organizamos de forma compacta esta expresión, obtendremos lo siguiente.

$$[h_1\ \ h_2\ \ h_3] = \lambda A\ [r_1\ \ r_2\ \ t] \ \rightarrow \ \ [r_1\ \ r_2\ \ t] = \lambda^{-1}A^{-1}[h_1\ \ h_2\ \ h_3]$$

A partir de esta expresión, para desacoplar los parámetros intrínsecos de los extrínsecos, estableceremos las siguientes condiciones.

$$\text{Se cumple: } \begin{cases} \mathbf{r}_1^T\mathbf{r}_2 = 0 \ \Rightarrow \ \mathbf{h}_1^T A^{-T}A^{-1}\mathbf{h}_2 = \mathbf{h}_1^T K^{-1}\mathbf{h}_2 = 0 \\ \mathbf{r}_1^T\mathbf{r}_1 = \mathbf{r}_2^T\mathbf{r}_2 \ \Rightarrow \ \begin{cases} \mathbf{h}_1^T A^{-T}A^{-1}\mathbf{h}_1 = \mathbf{h}_2^T A^{-T}A^{-1}\mathbf{h}_2 \\ \mathbf{h}_1^T K^{-1}\mathbf{h}_1 = \mathbf{h}_2^T K^{-1}\mathbf{h}_2 \end{cases} \\ \text{Con } K = A \cdot A^T \end{cases}$$

Imagen 51-Condiciones desacoplo parámetros

De esas condiciones, sacamos dos restricciones lineales por cada vista, y por tanto necesitamos tres vistas. Las dos restricciones por cada vista, son las siguientes.

$$\begin{cases} \mathbf{h}_1^T K^{-1} \mathbf{h}_2 = 0 \\ \mathbf{h}_1^T K^{-1} \mathbf{h}_1 = \mathbf{h}_2^T K^{-1} \mathbf{h}_2 \end{cases}$$

Imagen 52-Restricciones calculo parámetros

Seguidamente, con el conjunto de restricciones formado por las tres vistas como mínimo, determinaríamos la matriz *K*. Una vez conocida *K*, podemos atender a su expresión.

$$K = AA^T = \begin{bmatrix} K_x f & 0 & C_x \\ 0 & K_y f & C_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} K_x f & 0 & 0 \\ 0 & K_y f & 0 \\ C_x & C_y & 1 \end{bmatrix} = \begin{bmatrix} K_x^2 f^2 + C_x^2 & C_x C_y & C_x \\ C_x C_y & K_y^2 f^2 + C_y^2 & C_y \\ C_x & C_y & 1 \end{bmatrix}$$

Imagen 53-Expresión matriz K

Primero tendríamos que calcular $A\ y\ A^T$ con la descomposición de Cholesky; $Cholesky(K) = AA^T$. Una vez hallada la descomposición, podríamos hallar los parámetros intrínsecos, los cuales se calcularían como se indica a continuación.

$$f_x = K_x f$$

$$f_y = K_y f$$

$$C_x, C_y$$

Por último, se puede proceder al cálculo de los parámetros extrínsecos. Conocemos la matriz *A* y la matriz *H*, por tanto, podemos utilizar la siguiente expresión.

$$[\mathbf{r}_1 \quad \mathbf{r}_2 \quad \mathbf{t}] = \lambda^{-1} A^{-1} [\mathbf{h}_1 \quad \mathbf{h}_2 \quad \mathbf{h}_3]$$

Imagen 54-Expresión para hallar parámetros extrínsecos

Para calcular el factor de escala, landa, podemos forzar que la norma de r_1 sea uno.

$$\left|\left|r_1\right|\right| = 1 \rightarrow \ \lambda = 1/||\mathrm{A}^{-1} h_1||$$

Con el factor de escala calculado, podemos hallar r_1 y r_2 con la expresión de la imagen 54. Y ya solo nos faltaría $r_3 = r_1 x r_2$.

La cuarta y última etapa de la calibración, es la estimación de la distorsión con una optimización no lineal a posteriori. Una vez calculado el modelo lineal, iremos calculando el error de proyección e iremos optimizando los valores de los coeficientes de la siguiente expresión.

$$\left.\begin{aligned} x_f &= C_x + fK_x x_n + fK_x[k_1 r_n^2 x_n + k_2 r_n^4 x_n + k_3 r_n^6 x_n + 2p_1 x_n y_n + p_2(2x_n^2 + r_n^2)] \\ y_f &= C_y + fK_y y_n + fK_y[k_1 r_n^2 y_n + k_2 r_n^4 y_n + k_3 r_n^6 y_n + 2p_2 x_n y_n + p_1(2y_n^2 + r_n^2)] \end{aligned}\right\}$$

$$r_n^2 = x_n^2 + y_n^2 \quad ; \quad x_n = x_c/z_c \quad ; \quad y_n = y_c/z_c$$

Imagen 55-Estimación de la distorsión

Y con esto concluiríamos el proceso de calibración, que nos permite saber cómo se genera la proyección de un punto en el espacio, sobre la imagen.

4. Herramientas de software

4.1. ROS

Robot Operating System [6] es un *middleware*, ya que provee la infraestructura de comunicaciones necesaria para conectar componentes o aplicaciones de *software*, para el desarrollo de software para robots. ROS cubre la necesidad de comunicación entre los múltiples procesos involucrados en un sistema robótico y, además, dichos procesos pueden estar ejecutándose en el mismo computador o no.

Este *software*, fue desarrollado en 2007 por el Laboratorio de Inteligencia Artificial de *Stanford*, para dar soporte al proyecto llamado, *STAIR2*. Desde 2008, el desarrollo de ROS continuó principalmente en *Willow Garage*, un instituto de investigación robótica. Desde 2013 ha sido administrado por *Open Source Foundation*, ahora llamada*, Open Robotics*.

Aunque ROS sea un meta-sistema operativo para robots, provee los servicios necesarios de cualquier sistema operativo al uso, como puede ser, la abstracción de hardware, el control de dispositivos de bajo nivel, la implementación de funcionalidad de uso común, el paso de mensajes entre procesos y el mantenimiento de paquetes. Además, las aplicaciones desarrolladas se organizan en paquetes, los cuales contienen todos los datos, archivos y código del proyecto en cuestión.

ROS está basado en una arquitectura de grafos, donde el procesamiento toma lugar en los nodos, lo cuales pueden recibir, mandar y multiplexar mensajes de sensores, control, estados, planificaciones y actuadores, entre otros. Hay varios mecanismos, mediante los cuales, los distintos nodos de un sistema de ROS, se comunican.

Mecanismos de comunicación de ROS:

- **Mediante *Topics***

Esta comunicación consiste, en el paso de mensajes mediante un Publicador o Subscriptor anónimo, a través de buses nominales, denominados *R0S Topics*, que permiten una comunicación asíncrona entre el nodo publicador que emite un mensaje y el conjunto de nodos subscriptores que lo reciben.

Básicamente, un *topic* es un bus nominal que pasa mensajes de un nodo a otro. Por su parte, los nodos pueden ser Publicadores o Subscriptores. Un nodo Publicador, enviará una información concreta, utilizando un mensaje, a un *topic.* En dicho *topic* se almacenará esa información, además de otras informaciones que ya existiera previamente, y un nodo Subscriptor podrá acceder a esa información recibiéndola como mensaje. Varios nodos Publicadores pueden publicar información en un *topic*, y a su vez, varios nodos Subscriptores pueden comunicarse a la vez con un mismo *topic,* para acceder a la información contiene.

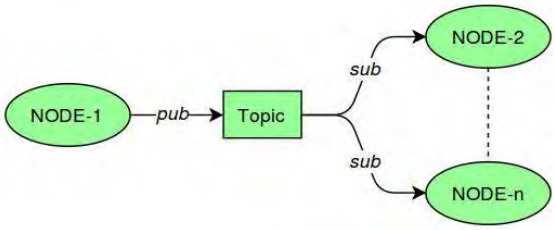


Imagen 56- Esquema comunicación con topics

- **Comunicación tipo Cliente/Servidor**

En este tipo de comunicación, existe un nodo cliente que llama, mediante una llamada remota de procedimientos, a el nodo servidor, el cual le responde.

- **Cliente/Servidor con Acciones**

Esta comunicación sigue el funcionamiento Cliente/Servidor pero ofrece, además, la posibilidad de realizar interrupciones en la comunicación. Estas comunicaciones se realizarán mediante unas herramientas llamadas Acciones, que permiten la monitorización del progreso, y que incluye la posibilidad de interrupción de la llamada.

La inicialización del ecosistema de ROS, se da cuando se ejecuta el comando *roscore*. *Roscore* es un servicio que permite que los diferentes nodos sean visibles y además se habiliten las comunicaciones entre dichos nodos.

Con la ejecución de *roscore*, se inicializan varias herramientas de ROS. Una es el *ROSMaster*, el cual es un servicio que permite que los diferentes nodos se registren en el sistema y monitoriza el sistema de comunicación de los nodos, mediante registros. Otra herramienta es el *Ros Parameter Server,* el cual sirve de almacenamiento de datos, de los diferentes nodos del sistema. *Roscore* también inicializa el nodo llamado *Rosout*.

Podemos utilizar las funcionalidades *ROS*, vía comandos del terminal, así como ejecutar programas que se encuentren en paquetes del espacio de trabajo de *ROS*. Existen bastantes comandos para realizar acciones utilizando *ROS*, desde el terminal; los comandos que han tenido un mayor uso en la realización de este proyecto son los siguientes.

- **roscore:** Este comando inicia el *ROSMaster.*
- **rosrun:** Permite ejecutar programas del espacio de trabajo de ROS. La sintaxis es la siguiente.

```
$ rosrun nombre_paquete nombre_ejecutable
```

- **roslaunch:** Este comando lo utilizaremos para lanzar los archivos que contengan, los modelos de simulación de *Gazebo.* Los archivos que se lanzarán con este comando, tienen formato, *.launch.* La sintaxis será la siguiente.

```
$ roslaunch nombre_paquete fichero.launch
```

- **rosnode:** Muestra información sobre los nodos. Este comando se puede utilizar con algún subcomando para realizar tareas concretas. Por ejemplo, el siguiente comando muestra la lista de todos los nodos que estén en ejecución y por tanto estén presentes en el entorno de *ROS*.

```
$ rosnode list
```

- **rostopic:** Por su parte, este comando muestra la información de los *topics* del entorno de ROS. También se puede combinar con algún subcomando, para mostrar información concreta. Por ejemplo, al ejecutar el siguiente comando, se mostrarán por el terminal los valores que se están almacenando en un *topic* concreto, en tiempo real.

```
$ rostopic echo nombre_topic
```

ROS ofrece una serie de herramientas, que facilitan notablemente el desarrollo de aplicaciones robóticas. Entre esas herramientas que ofrece ROS, las que han sido de utilidad en la realización del proyecto son las siguientes.

- **Lenguaje *URDF***
- **Visor *Rviz***
- **Simulador *Gazebo* (Integrada como paquete de ROS)**

Entraremos más a fondo a explicar esas tres herramientas, en los próximos apartados.

ROS es un *software* de código abierto, esto causa que a menudo necesite modificaciones, mejoras o incluso subsanación de errores. Eso implica que existan muchas versiones de este *software* y que se vean lanzamientos de nuevas versiones anualmente. Para cada versión de *Ubuntu LTS*, solo existe una versión de ROS compatible. Esto causa que se tenga que decidir incluso la versión del sistema operativo con la que trabajar, en función de la versión de *ROS* que más utilidad tenga en el proyecto en cuestión.

En este proyecto, se utilizará la versión *ROS Kinetic*, cuya fecha de lanzamiento fue el 23 de mayo de 2016. Debido a esto, la versión de *Ubuntu* que usaremos es la 16.04 *LTS*.

En cuanto a los lenguajes de programación que se utilizan para desarrollar aplicaciones con *ROS*, los más utilizados son *C++* y *Python*. Se puede usar la librería *roscpp*, para programar código *C++* en *ROS*. Para programar archivos de ros en lenguaje Python, debemos usar la librería *rospy*. En este proyecto, toda la programación se ha realizado en lenguaje Python.

Hablaremos de Python y de todas las librerías que hemos utilizado, en apartados posteriores.

4.1.1. Configuración del espacio de trabajo de ROS

Ante empezar a desarrollar *software* en *ROS*, necesitamos configurar nuestro espacio de trabajo donde vamos tener todos los ficheros relacionados con el proyecto. Los espacios de trabajo de *ROS*, comúnmente llamados *Catkin Workspace,* simplemente son espacios de trabajo estandarizados, que se suelen utilizar cuando se trabaja con *ROS*, para organizar los ficheros de los proyectos.

Para crear este *workspace* desde el terminal de *Ubuntu*, debemos ejecutar los siguientes comandos, ubicándonos en el directorio *home.*

```
$ mkdir new_catkin_ws
$ cd new_catkin_ws
$ mkdir src
$ catkin_make
```

Con esto ya tendríamos creado el espacio de trabajo donde tendremos los distintos modelos URDF, los distintos ficheros de código y el resto de archivos que se necesita en un proyecto de ROS. Como podemos ver, el nombre de nuestro espacio de trabajo es, *new_catkin_ws.* En cuanto al comando *catkin_make,* este es utilizado para construir entornos de trabajo y para compilar paquetes.

Ahora, para ser capaces de utilizar los paquetes de este entorno de trabajo desde cualquier terminal, tenemos dos opciones. La primera opción es ejecutar el siguiente comando en cada terminal, en el cual queramos ejecutar archivos del entorno.

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

La segunda opción es colocar ese mismo comando, al final del archivo. *bashrc* del sistema operativo, ya que las sentencias que se encuentren en este archivo, se ejecutan cada vez que se inicia dicho sistema operativo.

4.2. MoveIt!

A la hora de comandar a un brazo robótico con el fin de coger un objeto o ir a una posición específica, se necesita planificar la secuencia de posiciones que debe seguir cada articulación en cada instante de tiempo; este procedimiento sabemos que se llama, planificación de trayectorias. Esa planificación es una tarea realmente compleja, y en *ROS*, la herramienta principal para realizar esas tareas de cálculo cinemático y planificación es *MoveIt.*

MoveIt, genera en el entorno de *ROS*, un nodo llamado *move_group.* Este nodo sirve como interfaz de usuario, ya que proporciona acciones y servicios, para comunicarse tanto con los controladores, para comandar acciones al robot, como con los distintos sensores que pueda haber en la simulación, para controlar el estado del robot.

La arquitectura de alto nivel del nodo primario, *move_group,* que genera *MoveIt* es la siguiente.

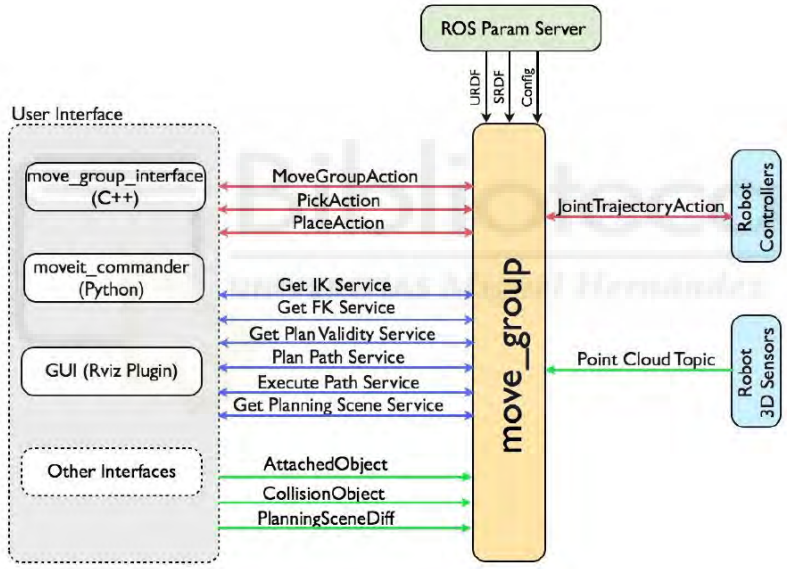


Imagen 57-Arquitectura alto nivel de move_group

En el recuadro de la izquierda, podemos ver los componentes de la interfaz de usuario, que son las formas en las que un usuario puede acceder al nodo *move_group*. Por tanto, podemos distinguir tres formas de acceder a las acciones o servicios que nos permiten comunicarnos con ese nodo.

- ***move_group_interface (C++):*** Este paquete nos permitirá acceder mediante acciones o servicios, a las funcionalidades del nodo *move_group,* programando en lenguaje C++.

- ***moveit_commander (Python):*** Este paquete por su parte, nos permite acceder a esas acciones o servicios, con un interfaz de programación en Python.

- ***GUI (Rviz):*** El sistema llamado, *Graphic User Interface,* nos permite realizar la planificación de movimiento a través del *software* Rviz. Esta herramienta ofrece una visualización 3D del modelo del robot sobre el que se está actuando y, además, ofrece una interfaz gráfica, llamada *MottionPlanning,* que es la que nos permite mover el robot a la posición deseada del robot. Todas las acciones que realicemos sobre la interfaz gráfica, y que impliquen un movimiento del robot, se visualizarán sobre el visor 3D de *Rviz.*

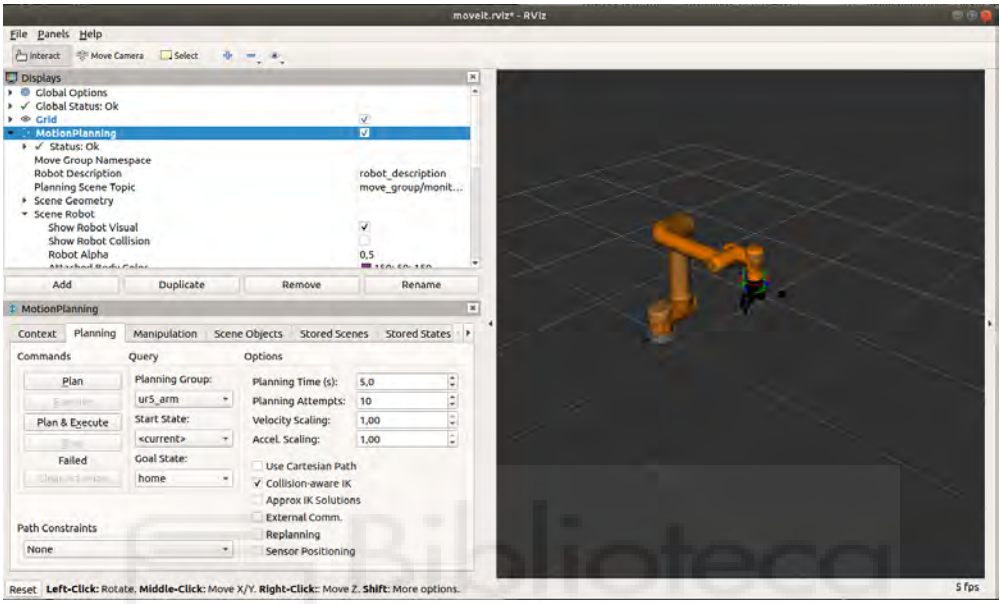


Imagen 58-Interfaz gráfica Rviz

El nodo *move_group,* obtiene la información de configuración del *ROS Parameter Server,* como podemos ver en la imagen 57. Este nodo, puede recibir tres tipos de información.

El primer tipo de información es un modelo *URDF*; este modelo viene almacenado dentro del parámetro *robot_description,* en el almacenamiento de *ROS Parameter Server* y gracias a él se obtiene el modelo *URDF* del robot. El segundo tipo de información es un modelo SRDF; este modelo viene almacenado en el parámetro *robot_description_semantic* y contiene el modelo semántico, *SRDF*, del robot. El tercer tipo de información es la configuración general de *MoveIt;* esta configuración incluye límites en las articulaciones, planificación de trayectorias, cinemática y otro tipo de configuraciones.

El nodo *move_group,* se comunica con el entorno del robot mediante *topics* o *actions.* Se pude comunicar mediante *topics,* con el entorno del robot, con la intención de recibir información del estado del robot o para recibir información de sensores 3D, presentes en la simulación.

También, *move_group* se puede comunicar con los controladores del robot mediante *actions.* La interfaz que permite comunicarse mediante *actions* se llama,

FollowJointTrajectoryAction. Con esa interfaz, los controladores del robot reciben las instrucciones de movimientos procedentes de *move_group.*

MoveIt, también nos permite planificar el entorno del robot.

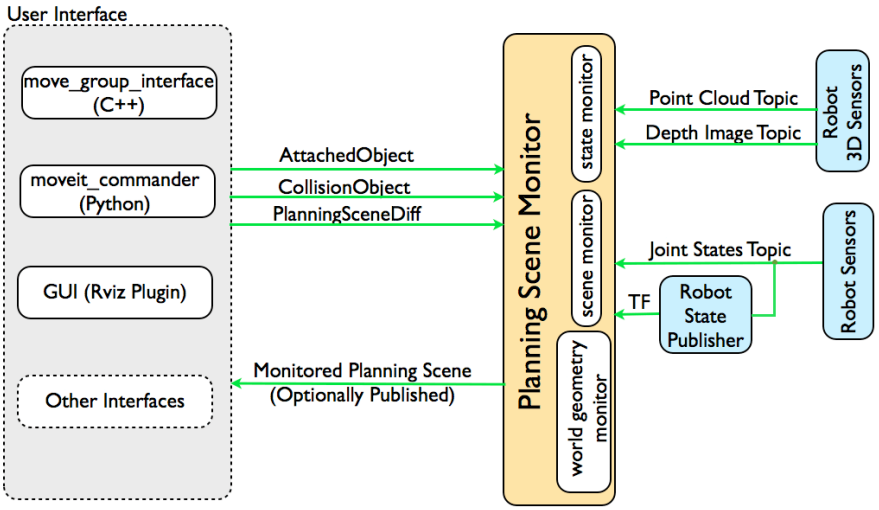


Imagen 59-Arquitectura planificación escenario

La planificación de la escena, la gestiona el *Planning Scene Monitor.* La gestión de esta planificación se lleva a cabo gracias a varias comunicaciones. Este nodo se comunica con el *topic,* llamado *joint_states,* que contiene el estado del robot. *Planning Scene Monitor* también se comunica con las entradas de usuario usando el *topic, planning_scene.* También utiliza el *World Geometry Monitor,* el cual utiliza la información de los sensores que estén en el entorno y de las entradas del usuario, y a partir de eso construye la geometría del entorno.

4.3. Gazebo

Gazebo es una herramienta de simulación que tiene un motor de simulación dinámica potente, gráficos de alta calidad, buenas interfaces gráficas y de programación. Esta herramienta permite realizar simulaciones de robots articulados en entornos complejos, realistas y tridimensionales. Realmente, esta aplicación no forma parte de las aplicaciones originarias de ROS, pero ha sido integrada como paquete de ROS, gracias al llamado *gazebo_ros_pkgs.*

Las características del *software Gazebo,* más destacables son las siguientes.

- Es un *software* libre.
- Nos da la posibilidad de crear entornos de simulación, en los cuales, podemos cambiar las características físicas de las colisiones con el suelo y los demás obstáculos, además de poder modificar la influencia de la gravedad en los tres ejes.
- Capacidad de utilizar el lenguaje *URDF*, para desarrollar y simular modelos de robots propios.

- Ofrece una simulación realista de los cuerpos rígidos. Eso permite que los robots, puedan interactuar con los distintos objetos del mundo y que los objetos puedan deslizarse, rodar y chocar tanto con como por las demás superficies de la simulación.
- Es compatible con *ROS*, se pueden controlar las simulaciones de *Gazebo* por medio de las *API's* de *ROS*. Además, es posible ejecutar *Gazebo* desde *ROS*.
- *Gazebo* contiene diversas extensiones para incluir sensores al modelo del robot y simular su funcionamiento.

Para simular la dinámica de los objetos, en el espacio tridimensional, *Gazebo* utiliza dos motores de simulación. El primero, y en el que está basada la arquitectura, es el motor *Ode Dynamics Engine, ODE*. El motor *ODE* proporciona una implementación rápida y estable de los cálculos físicos, esto permite una simulación precisa y eficiente. El segundo motor que utiliza *Gazebo* es el llamado, *Bullet Physics.* Este segundo motor es conocido principalmente por su eficiencia y su capacidad de manejar interacciones complejas entre objetos.

La integración de *Gazebo* con *ROS* se pude llevar a cabo utilizando una estructura de paquetes. que facilitan la utilización de *Gazebo* desde el entorno de ROS. La estructura se llama *simulator_gazebo,* y los paquetes que incluye son los siguientes.

- ***gazebo:*** Este paquete contiene la versión más reciente de *Gazebo,* que es compatible con *ROS*. Al utilizarlo, se crea un nodo más en la arquitectura de *ROS*, el cual posee sus propios *topics* y servicios, para interactuar con él, desde ROS.

- ***gazebo_msgs:*** Contiene varios tipos de mensajes y de servicios, los cuales permiten que, desde ROS, se pueda interactuar con *Gazebo*.

- ***gazebo_plugins:*** Este paquete contiene las extensiones que permiten simular diferentes tipos de sensores en *Gazebo*.

- ***gazebo_tools:*** El uso y ejecución de este paquete genera un nodo llamado *gazebo_models.* Este nodo es el que nos permite enviar o eliminar modelos URDF, en el simulador.

- ***gazbeo_worlds:*** En este caso, el paquete contiene archivos de configuración estándares del entorno y modelos *URDF*, de varios objetos.

Podemos afirmar que, *MoveIt* es la herramienta de *ROS* que se encarga del cálculo cinemático y de la planificación de trayectorias, correspondientes a las acciones que se quieran realizar sobre el robot. *Gazebo* no nos ofrece la capacidad de cálculo y planificación de *MoveIt*. En cambio, la ventaja que ofrece *Gazebo* es, su protocolo de comunicaciones con *ROS*, ya que ofrece unas características muy similares a lo que sería el proceso de comunicación que se establecería entre *ROS* y un robot real.

Por tanto, cuando se quiera comandar alguna tarea para el robot desde *ROS*, la herramienta *MoveIt,* realizará los cálculos y la planificación pertinentes para que el robot logre llegar a su destino. A su vez, *ROS* se comunicará con *Gazebo* para actuar sobre la simulación del robot, y esta simulación modelada en *Gazebo* responderá de forma similar a un robot real, interactuando con los objetos que tenga en su entorno y realizando las tareas y rutinas que se le han especificado desde ROS.

4.4. Lenguaje URDF

El lenguaje *URDF*, *Unified Robot Description Format*, es un lenguaje de modelaje de robots. Este lenguaje, se utiliza con ficheros en formato *XML* y contiene la información de los eslabones y articulaciones de la cadena cinemática que forma el robot. Este tipo de modelos *URDF*, también contiene información sobre la ubicación de los sensores, la apariencia visual de cada parte del robot, o las características dinámicas de las articulaciones y de los eslabones.

Los simuladores de *ROS*, como *Rviz*, usan modelos URDF para generar los modelos robóticos que se van a simular. Los modelos *URDF*, cuando entran en el ecosistema de ROS se almacenan como parámetros, habitualmente llamados *robot_description*, en la herramienta *Param Server.*

Los modelos *URDF* se escriben en ficheros del tipo *xacro,* que es un formato de archivo que hace que este tipo de modelos sean más fáciles de comprender, para facilitar su generación y mantenimiento.

Para el modelaje de robots, también existe un modelo semántico llamado *Semantic Robot Description Forma, SRDF.* El formato *SRDF*, describe la información semántica sobre el robot, que no esté en el fichero *URDF*.

Básicamente, los modelos *URDF* funcionan como un árbol de *links,* conectados entre sí mediante *joints.* Un *link* representa un componente físico del robot, mientras que un *joint* representa como se mueve un *link* respecto a otro *link,* además de definir su posición. Hay bastantes tipos de *joint,* y la forma relativa en que se mueven dos *links* entre sí, dependerá del tipo de *joint* utilizado. Los tipos más habituales son los siguientes.

- ***Revolute*****:** Define un movimiento rotacional con restricciones de ángulo máximo y mínimo.
- ***Continuous*****:** Define un movimiento rotacional sin restricciones.
- ***Prismátic*****:** Define un movimiento lineal con límites de posición máxima y mínima.
- ***Fixed*****:** El *link* hijo está rígidamente conectado al *link* padre.

La sintaxis *URDF* está estructurada mediante etiquetas. Todo modelo *URDF* debe empezar con el uso de la etiqueta ***<robot>***. Esta etiqueta es la raíz de todo el modelo del robot, y sirve para especificar el nombre del modelo.

<robot name="robot_name">

. . .

Modelo del robot con el resto de etiquetas.

. . .

</robot>

Dentro de esa etiqueta, se encuentran todas las etiquetas que describen el modelo del robot. Las dos etiquetas principales dentro de una descripción *URDF* son, ***<link>*** y ***<joint>***.

La etiqueta *<link>* marca el comienzo del modelado de un *link,* además de especificar el nombre de dicho *link.* Las características que podemos definir de un *link,* las podemos especificar dentro de su etiqueta, por medio de las siguientes etiquetas.

1) ***<visual>*:** Esta característica influirá en la forma que visualicemos el robot en los simuladores. Podemos modificar las características de este, utilizando las siguientes etiquetas.
   a) ***<geometry>*:** *box/cilynder/sphere* especificando las medidas o *mesh.*
   b) ***<origin>*:** Para especificar un *offset*, desde el origen del *link.*
   c) ***<material>*:** Especifica el color del *link.*

2) ***<collision>*:** Características en el cálculo físico de las colisiones.
   a) ***<geometry>*:** Especifica el área del que abarcan las posibles colisiones del *link.*
   b) ***<origin>*:** *Offset* en el área considerada de colisión.
3) ***<inertial>*:** Usado para determinar cómo los *links* responden a fuerzas externas.
   a) ***<mass>*:** Masa del *link*
   b) ***<origin>*:** Centro de masas
   c) ***<inertia>*:** Especifica la matriz de rotación inercial.

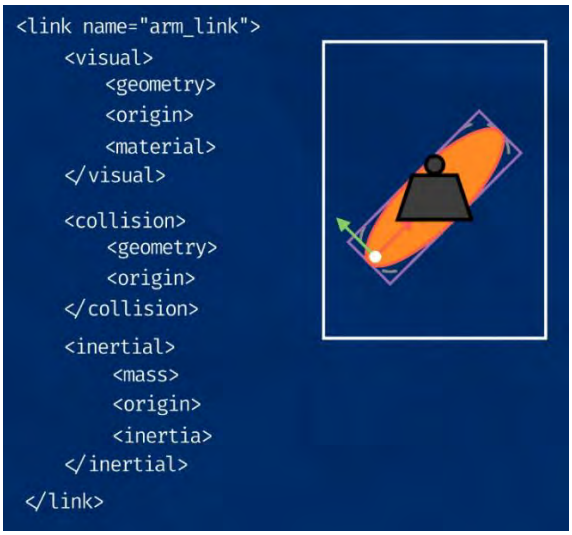


Imagen 60-Ejemplo de definición de un link

La otra etiqueta principal de un *URDF* es la etiqueta ***<joint>***. Estas etiquetas definen la localización de los *links,* y como es el movimiento relativo entre ellos. Dentro de esta, se utilizan las siguientes etiquetas.

1) ***<name>*:** Se especifica el nombre del *joint.*
2) ***<type>*:** Se especifica el tipo de *joint.*
3) ***<parent>*:** *Link* padre.
4) ***<child>*:** *Link* hijo.
5) ***<origin>*:** Relación entre los dos *links* antes de arrancar el movimiento.
6) ***<axis>*:** A lo largo o alrededor de que eje moverse.
7) ***<limits>*:** Limitaciones en las físicas del robot. Dentro de estos límites se pueden especificar los límites de posición máximos y mínimos, los límites de velocidad y los límites de esfuerzo.

```
<joint name="arm_joint" type="revolute">
    <parent link="slider_link"/>
    <child link="arm_link"/>
    <origin xyz="0.25 0 0.15" rpy="0 0 0"/>
    <axis xyz="0 -1 0"/>
    <limit lower="0" upper="${pi/2}" velocity="100" effort="100"/>
</joint>
```

Imagen 61-Ejemplo definición de un joint

4.5. Python

El *software ROS*, permite utilizar varios lenguajes de programación. Por ejemplo, podemos programar los diferentes archivos de un proyecto de *ROS* utilizando *java*, si introducimos la librería *rosjava. ROS* también incluye una interfaz que permite la comunicación y el control de robots mediante *Matlab.* Pero sin lugar a dudas, los lenguajes de programación más utilizados para programar los ficheros del ecosistema de *ROS* son, *C++* y *Python*.

El funcionamiento de *ROS*, es independiente del lenguaje de programación utilizado, por ejemplo, cada uno de los nodos que se generen en *ROS*, pueden estar escritos en distintos lenguajes de programación. Eso refuerza la reutilización de ficheros de programación en otros proyectos o incluso, integrar ficheros de diferentes aplicaciones, escritos de *C++,* a aplicaciones escritas en *Python*.

Las aplicaciones de este proyecto, se han programado completamente en *Python.* Toda la programación y la visualización de los distintos ficheros del ecosistema de *ROS* se ha realizado con *Visual Studio Code,* que es un editor de textos y depurador de código. Como estamos utilizando la versión *ROS Melodic,* se ha utilizado *Python 2.7,* que es una versión de ese lenguaje compatible con dicha versión de *ROS*.

En la realización de las diferentes aplicaciones del proyecto, se han utilizado varias librerías de *Python.* Las librerías que se han integrado en el proyecto, las comentaremos a continuación.

- ***rospy***

Esta librería proporciona una serie de comandos y funciones para poder trabajar con las funcionalidades de *ROS*, utilizado lenguaje *Python.*

- ***sys***

Permite manipular varios aspectos del proceso de ejecución del programa.

- ***numpy***

Esta librería permite simplificar todo lo relacionado con las operaciones matemáticas y el tratamiento de *arrays,* en *Python.*

- ***math***

La librería *math* es una biblioteca predefinida en *Python,* que contiene una serie de constantes y una serie de funciones matemáticas.

- ***moveit_commander***

Al igual que la librería *rospy*, la librería *moveit_commander* es importantísima para trabajar con las funcionalidades de *ROS*, ya que esta librería nos permite interactuar con la herramienta *MoveIt.* Con esta librería podremos controlar el movimiento del robot y la planificación de trayectorias.

- ***moveit_msgs.msg***

Contiene la estructura con los posibles mensajes que se pueden utilizar para comunicarse con el sistema *MoveIt.*

- ***geometry_msgs.msg***

Esta librería contiene tipos de mensajes utilizados para representar geometría y transformaciones en el espacio tridimensional. La utilizaremos, por ejemplo, para establecer la posición del extremo del robot, en la cinemática inversa.

- ***std_msgs.msg***

Por su parte, esta librería contiene definiciones de mensajes estandarizados, con una gran variedad de tipos de datos básicos.

- ***sensor_msgs.msg***

Los tipos de mensajes que contiene esta estructura, representan datos de sensores. En nuestro caso, dentro de la librería utilizaremos el tipo de mensaje llamado, *CompressedImage,* para recibir y manipular la información de la cámara.

- ***shape_msgs.msg***

Esta librería contiene mensajes que representan diferentes tipos de formas tridimensionales.

- ***cv2***

Esta es la librería *OpenCV*, la cual contiene múltiples operaciones de procesamiento de imagen y de visión artificial.

- ***pynput***

Proporciona la capacidad de detectar entradas de dispositivos externos como el teclado o el ratón. En nuestro caso se utilizará para registrar las entradas por teclado, de la aplicación de control manual del robot.

- ***time***

Esta librería es estándar en *Python,* sirve para el manejo del tiempo.

5. Entornos de simulación y software realizado

En este apartado se van a mostrar todo el software que se ha desarrollado y utilizado, para implementar los objetivos de aplicación que se han propuesto. Primero hablaremos de los modelos *URDF* que se han utilizado y de las simulaciones que se han creado. Seguidamente, se mostrarán los códigos que se han desarrollado, en lenguaje *Python,* para comandar el robot en la realización de las distintas aplicaciones.

5.1. Simulación creada en gazebo

5.1.1. UR5

La intención de este proyecto, es utilizar el robot *UR5*, de la empresa *Universal Robots,* para usarlo junto con el planificador de *MoveIt,* con el objetivo de realizar todas las aplicaciones del proyecto. Por tanto, lo primero que debemos plantearnos para poder controlar el robot UR5 en una simulación de *Gazebo* es, como podemos obtener la simulación del robot UR5.

Antes de empezar a desarrollar la simulación, tenemos que tener nuestro entorno de trabajo de *ROS* configurado tal y como se muestra en el apartado 4.1.1. Una vez lo tenemos creado y configurado, podemos empezar a añadir los archivos de simulación a dicho entorno. Como comentábamos anteriormente, lo primero que realizaremos es obtener el modelo de simulación del robot *UR5*. Este modelo lo obtendremos del repositorio de *GitHub* llamado, *ROS-Industrial.*

El repositorio *ROS-Industrial,* contiene una serie de paquetes, herramientas y *drivers,* para trabajar con varios *hardware* industriales. Este repositorio está pensado para dar soporte a los profesionales e investigadores de la industria de la robótica. Proviene al usuario de un proceso fácil mediante el cual trabajar con robots industriales, usando la arquitectura de ROS.

Este repositorio contiene varios meta-paquetes, que corresponden a varias marcas de uso común, de robots industriales. En nuestro caso, de este repositorio, clonaremos el paquete de *Universal Robots*, paquete que contiene múltiples directorios, que nos dan la posibilidad de realizar simulaciones con varios robots de dicha empresa.

Para clonar dicho paquete, en nuestro espacio de trabajo, y ser capaz de utilizarlo en simulación ejecutaremos los siguientes comandos por terminal. Estos comandos los ejecutaremos ubicándonos en el directorio *home.*

```
$ cd new_catkin_ws/src

$ git clone -b kinetic-devel https://github.com/ros-
industrial/universal_robot.git

$ cd new_catkin_ws

$ catkin_make
```

Observando los comandos anteriores, nos podemos dar cuenta de que hemos clonado el paquete, en la versión kinetic-*devel.* Ésta es la versión del paquete que estaba preparada para *ROS Kinetic.* Aunque en este proyecto utilicemos la versión *ROS Melodic,* utilizaremos el paquete *kinetic-devel,* porque que la versión *melodic-devel* de dicho paquete, nos daba problemas de compatibilidad, a la hora de introducir la pinza al extremo del robot.

Una vez clonado el paquete en nuestro entorno, si hemos activado el espacio de trabajo, podremos lanzar la simulación del robot *UR5*. Para lanzar la simulación en G*azebo* del robot UR5, ejecutaremos el siguiente comando.

```
$ roslaunch ur_gazebo ur5.launch
```

Si ejecutamos ese comando, podremos visualizar el siguiente escenario de *Gazebo.*

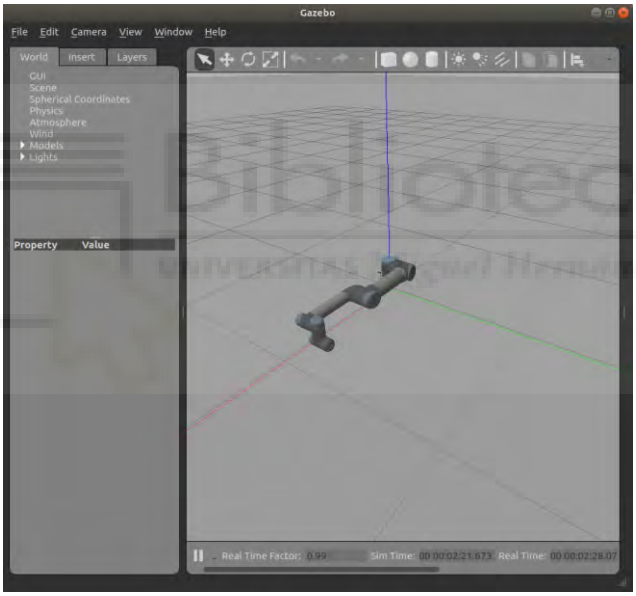


Imagen 62-Simulación robot UR5 en Gazebo

Esta es la simulación en *Gazebo* del robot UR5. El meta-paquete de *Universal Robot,* también contiene los archivos de configuración necesarios, para arrancar el planificador *MoveIt.* Para configurar los nodos de *MoveIt,* que permiten la planificación de movimiento con el robot simulado, ejecutaremos el siguiente comando.

```
$ roslaunch ur5_moveit_config ur5_moveit_planning_execution.launch
sim:=true
```

Con este comando, tendríamos configurados todos los nodos pertenecientes al planificador *MoveIt*. Con esto, ya podríamos mover el robot simulado en *Gazebo*, con cualquiera de las interfaces que nos ofrece el nodo *move_group.* En este caso, no utilizaremos la interfaz de *move_group* utilizando *Python,* porque la utilizaremos más tarde, a partir del apartado 5.2. En este caso lanzaremos la interfaz *Rviz*, desde el terminal, y veremos que ya podemos comandar el robot, debido a que se han configurado correctamente los nodos de *MoveIt,* y por tanto podemos mover el robot.

```
$ roslaunch ur5_moveit_config moveit_rviz.lauch config:=true
```

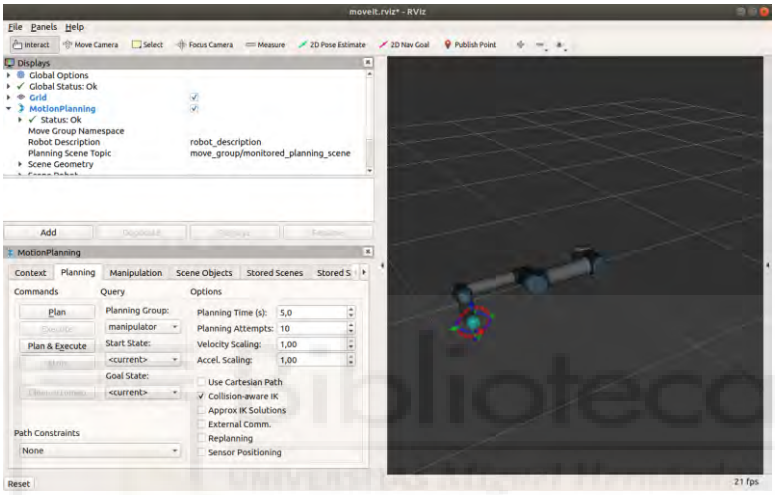


Imagen 63-Robot UR5 en el visor 3D de Rviz

A la derecha de la imagen está el visor 3D, que ofrece *Rviz* para visualizar el estado del robot. A la izquierda de la imagen 63, podemos ver una ventana llamada *MotionPlanning*, desde esta ventana podemos mover al robot. Una de las formas de mover el robot es, escogiendo en la casilla *Goal State* una de las posiciones predefinidas del robot. Por ejemplo, escogiendo *Goal State*: *up,* podremos visualizar lo siguiente en *Rviz*.

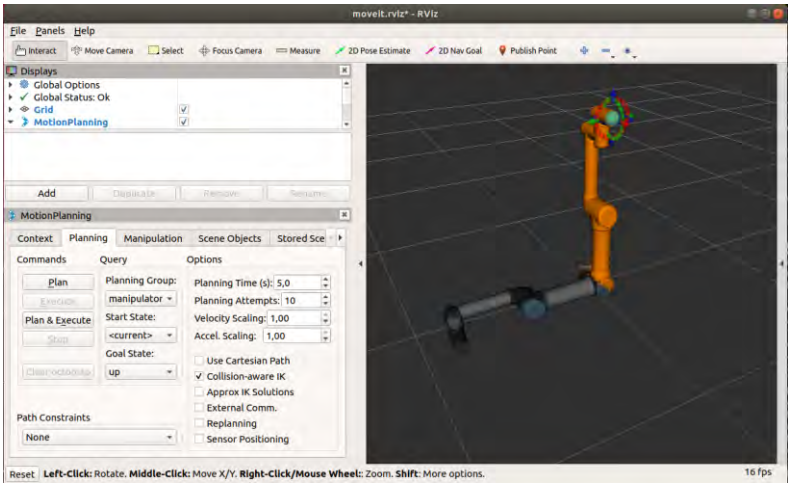


Imagen 64-Mover el UR5 con Rviz utilizando Goal State

En esta imagen, observamos como *Rviz* muestra una previsualización, en color naranja, de cómo será la nueva posición del robot. Si pulsamos a *Plan&Execute*, se ejecutará el cambio de posición en la simulación de *Gazebo.* El resultado de la ejecución en *Gazebo* sería la siguiente.

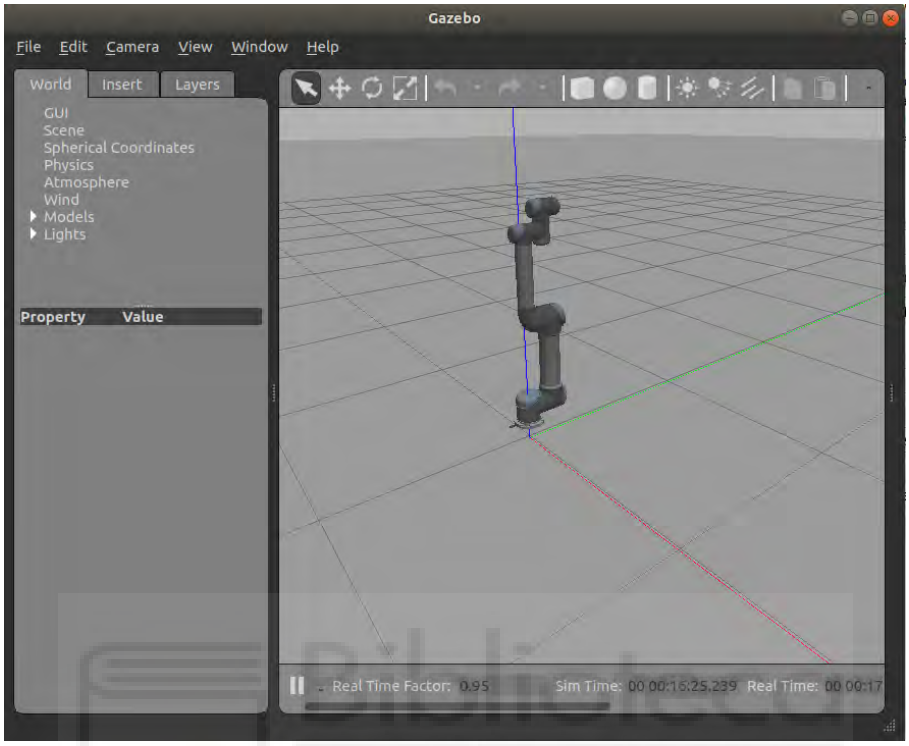


Imagen 65-Nueva posición del UR5 utilizando Rviz

También podemos mover el robot, desde la pestaña *Joints,* dentro de la ventana *MotionPlanning.* En este caso, seleccionaremos la posición angular de cada articulación.

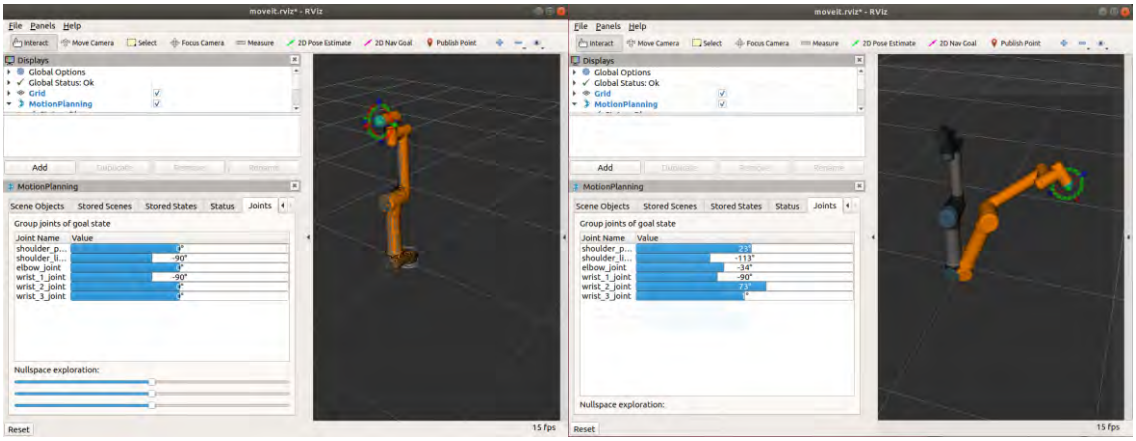


Imagen 66- Mover el UR5 con Rviz utilizando la pestaña Joints

Podemos ver en la imagen 66, que se han modificado las posiciones angulares de las articulaciones. Si pulsamos *Plan&Execute*, se ejecutaría este cambio de posición en *Gazebo.*

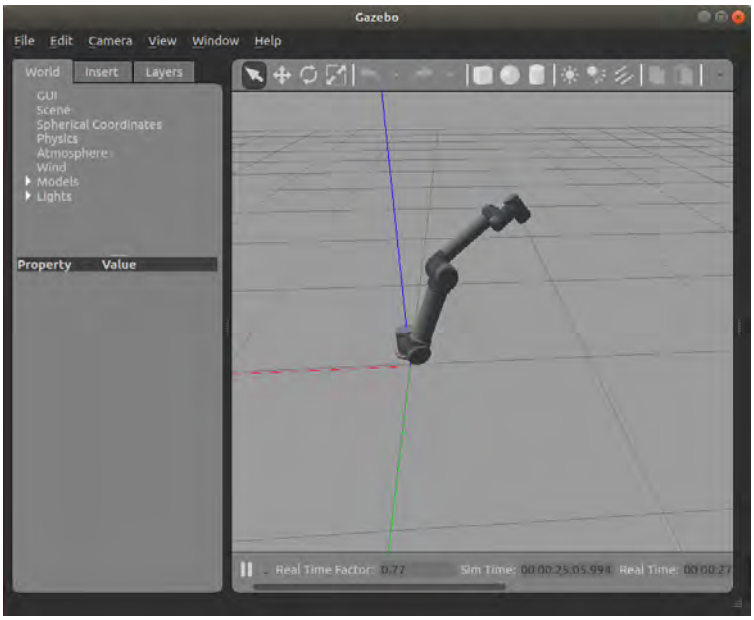


Imagen 67-Resultado del movimiento en la simulación

Ahora mismo ya tenemos una simulación del robot UR5 totalmente funcional, lo siguiente que tenemos que plantearnos es, como introducir una pinza en el extremo del robot, para con ella, poder manipular objetos.

5.1.2. Pinza

Para conseguir introducir la pinza a la simulación de *Gazebo*, primero tenemos que preparar los ficheros *URDF* necesarios, para realizarlo. Lo primero que hay que destacar es, que la pinza que vamos a colocar en el extremo del robot, es una simulación de una pinza de la empresa *Robotiq.* Una vez sabemos esto, tenemos que clonar un repositorio que contiene la descripción de los atributos físicos de dicha pinza de *Robotiq.*

```
$ cd new_catkin_ws/src/
$ git clone https://github.com/filesmuggler/robotiq.git
```

Ahora ya tenemos el repositorio que describe la pinza, por tanto, necesitamos un modelo *URDF*, que combine la descripción del robot *UR5* y la pinza. Ese modelo *URDF*, lo ubicamos en el directorio, */new-catkin_ws/src/universal_robot/ur_description/urdf/*. Para obtener ese modelo y ubicarlo donde hemos comentado, ejecutaremos lo siguiente.

```
$ cd new-catkin_ws/src/universal_robot/ur_description/urdf/
$ wget
https://raw.githubusercontent.com/utecrobotics/ur5/master/ur5_description/urdf/ur5_robotiq85_gripper.urdf.xacro
```

Tanto el repositorio que describe la pinza, como este modelo *URDF* que acabamos de copiar, pueden estar desactualizados o simplemente debemos adaptarlos a los archivos de nuestro espacio de trabajo. Para ello debemos hacer los siguientes cambios.

Dentro del el modelo *URDF*, que acabamos de clonar, *ur5_robotiq85_gripper.urdf.xacro,* debemos modificar la línea 4 para poner los siguiente.

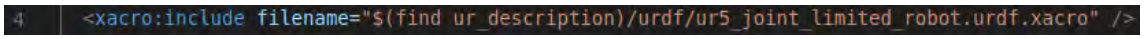


Imagen 68-Línea 4 archivo xacro

Con esto logramos que el modelo *URDF* de la pinza, busque e incluya el modelo del robot *UR5*. Seguidamente, tenemos que cambiar las interfaces de control para el robot y para la pinza, cambiaremos la interfaz *PositionJointInterface,* por *EffortJointInterface.*

Dentro del repositorio de robotiq, concretamente en el directorio robotiq_description/urdf/robotiq_85_gripper.transmission.xacro, modificaremos la línea 9 y 14, para dejarlas de la siguiente manera.

Imagen 69-Línea 9 archivo xacro

Imagen 70-Línea 14 archivo xacro

Con esto, ya tenemos cambiada la interfaz de la pinza. Para cambiar la del brazo robótico, iremos al paquete de *Universal Robots*, concretamente al directorio llamado *ur_description/urdf/ur5_joint_limited_robot.urdf.xacro* y cambiaremos la línea 5 por lo siguiente.

Imagen 71-Línea 5 archivo xacro

En este momento, ya tenemos tanto la descripción del brazo y la descripción de la pinza con las correcciones necesarias, y el modelo *URDF*, que conjuga ambas descripciones. Pero para poder controlar el robot con la pinza, necesitamos un paquete más, que contenga la configuración de *MoveIt* necesaria, para el cálculo de la cinemática y la planificación del movimiento del conjunto de robot y pinza juntos.

Para crear ese paquete, podríamos usar el asistente de *MoveIt.* Este asistente, nos permite seleccionar el modelo *URDF*, del conjunto robot y pinza, y en ese modelo podemos establecer el cálculo de las colisiones, para cada articulación el tipo,

seleccionar cuantos grupos de planificación queremos, predefinir poses del robot, entre otras cosas. Al crear un paquete con la configuración de *MoveIt,* se nos genera en él un ejecutable, tanto para simular el robot en *Gazebo,* como para manipularlo desde *Rviz*.

El resultado de la creación de ese paquete, nos da la configuración de *MoveIt* necesaria para mover el robot. El resultado que obtendríamos de la creación de ese paquete, podemos obtenerlo clonando el paquete *ur5_gripper_moveit_config* del repositorio llamado, darial. Esto lo podemos hacer ejecutando lo siguiente.

```
$ cd new_catkin_ws/src/
$ git clone https://github.com/darial/ur5_gripper_moveit_config
```

Una vez tenemos ese paquete, en nuestro entorno de trabajo, podemos lanzar tanto la simulación de *Gazebo,* como la interfaz de planificación *Rviz,* con el siguiente comando.

```
$ roslaunch ur5_gripper_moveit_config demo_gazebo.launch
```

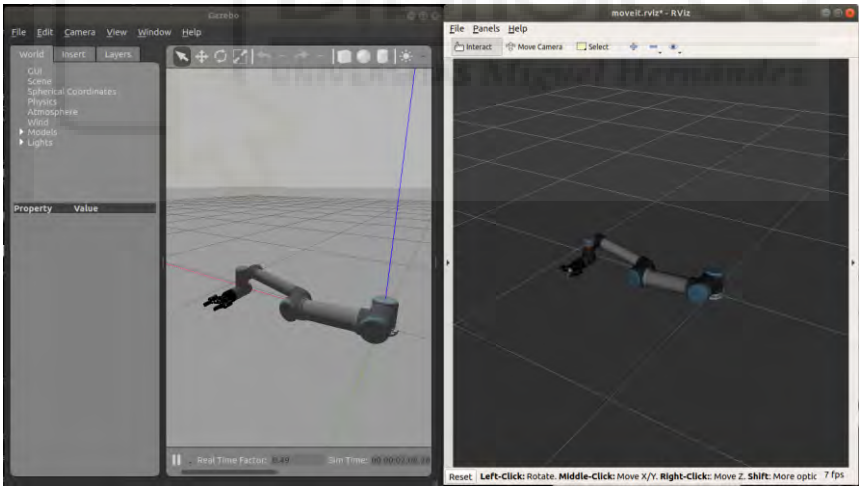


Imagen 72-Simulación del robot con la pinza adherida

En esta imagen, podemos observar la simulación en *Gazebo* del robot *UR5*, con la pinza incorporada en el extremo del robot. Podemos mover esta simulación, mediante la interfaz *Rviz.* En este caso, hay dos grupos de planificación, el primero que engloba solo al brazo robótico sin la pinza y el segundo que engloba solo al brazo. Seleccionamos que grupo mover, por medio de la casilla llamada, *Planning Group.* El movimiento del brazo robótico es idéntico a como lo hemos hecho en el apartado 5.1.1.

En el caso del movimiento de la pinza, debemos seleccionar *Planning Group: gripper.* Una vez seleccionado, podemos mover la pinza de la misma forma que lo hicimos en el apartado 5.1.1. con el brazo. Podemos seleccionar una posición

predefinida en *Goal State* y pulsar *Plan&Execute.* Otra manera, es utilizando la pestaña *Joints.*

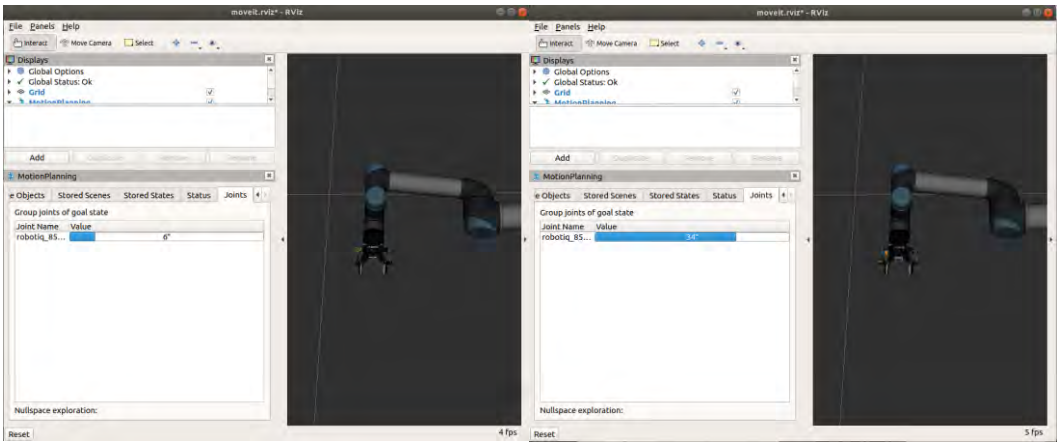


Imagen 73-Mover la pinza a través de Rviz

Si pulsamos *Plan&Execute,* visualizaremos los resultados en *Gazebo.*

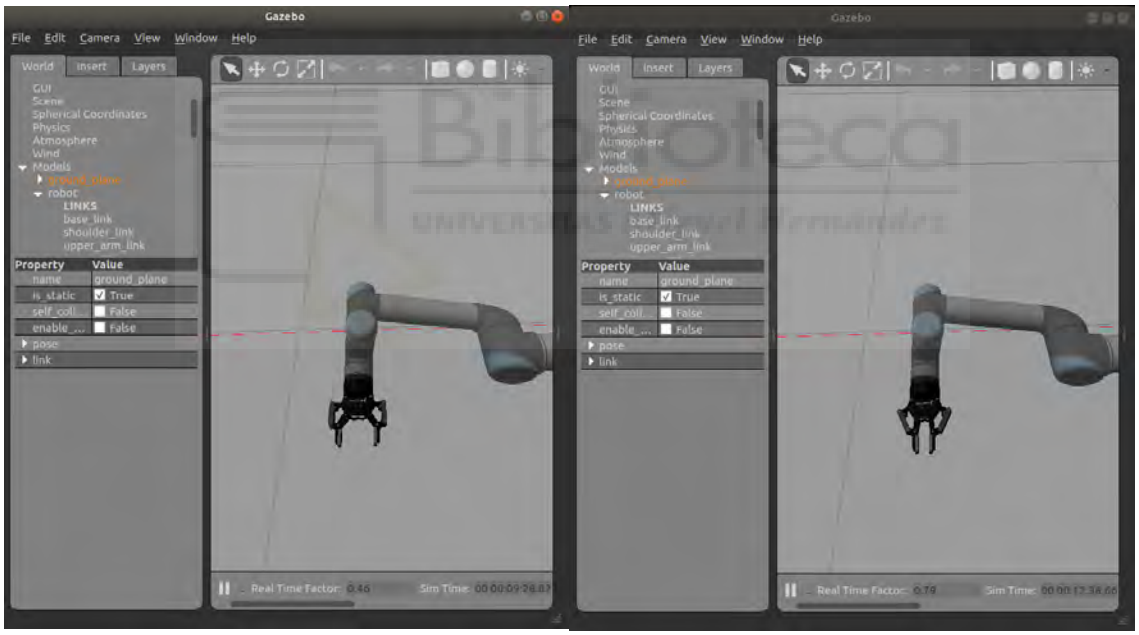


Imagen 74-Movimiento de la pinza en la simulación

5.1.3. Cámara

Para utilizar técnicas de visión artificial, necesitaremos añadir una cámara a la simulación, la cual nos permita tomar imágenes, para procesarlas y aplicar los algoritmos pertinentes. Esto lo conseguiremos gracias a que *Gazebo* soporta una serie de complementos para sus simulaciones, como por ejemplo sensores, y que además pueden ser conectados con ros. Uno de estos complementos que son soportados, es la simulación de una cámara.

El primer paso, es añadir un eslabón más al modelo *URDF* del robot, el cual representará la cámara. En nuestro caso, utilizaremos simplemente un bloque blanco que simulará ser el objeto cámara. Para ello, crearemos un *link,* que represente simplemente un bloque blanco, y después crearemos un *joint,* que fije ese bloque blanco a el último eslabón del robot. Con esto conseguiremos que la cámara se mueva conjuntamente a el último eslabón del robot.

Por tanto, lo primero que haremos es añadir el un *link* y un *joint* extra, al modelo URDF del brazo robótico. Concretamente, los añadiremos en el siguiente directorio: *universla_robot/ur_description/urdf/ur5_joint_limited_robot.urdf.xacro*, ya que este archivo contiene el modelo URDF del brazo robótico.

El *link*, que define el bloque blanco como parte del robot será el siguiente.

```
<!-- Camera -->
<link name="camera_link">
    <collision>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="${camera_width} ${camera_width} ${camera_width}"/>
        </geometry>
    </collision>

    <visual>
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <geometry>
            <box size="${camera_width} ${camera_width} ${camera_width}"/>
        </geometry>
        <material name="white"/>
    </visual>

    <inertial>
        <mass value="1e-5" />
        <origin xyz="0 0 0" rpy="0 0 0"/>
        <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6" />
    </inertial>
</link>
```

Imagen 75-Definición del link de conforma la cámara

El *joint*, que fijará la cámara al último eslabón del robot será de la siguiente manera.

```
<!-- Adding Camera to the robot end effector-->
<joint name="camera_joint" type="fixed">
    <origin xyz="0 0.1 0.15" rpy="0 0 1.57079633"/>
    <parent link="wrist_3_link"/>
    <child link="camera_link"/>
</joint>
```

Imagen 76-Definición del joint que une la cámara con el robot

El parámetro, *camera_width*, fija las dimensiones del cubo que representa la cámara.

```
<xacro:property name="camera_width" value="0.02" />
```

Imagen 77-Dimensiones cámara

Ahora que tenemos el bloque añadido a la simulación, tenemos que pasar a añadir a la descripción, la extensión de *Gazebo*, que le da al bloque la funcionalidad de recibir imágenes y hacerlas accesible mediante un *topic* de *ROS*. Existe un convenio de uso de la funcionalidades de *Gazebo*, que dice que cuando se usen complementos o extensiones de ese software, tenemos que definirlas en un fichero en formato, *.gazebo.*

Por tanto, lo primero que haremos es crear, en el mismo directorio donde está el modelo del brazo, un archivo llamado *ur5.gazebo.xml*. Dentro de este archivo escribiremos lo siguiente.

```
<?xml version="1.0"?>
<robot>
  <!-- camera -->
    <gazebo reference="camera_link">
        <sensor type="camera" name="camera1">
            <update_rate>30.0</update_rate>
            <camera name="head">
                <horizontal_fov>1.35</horizontal_fov>
                <image>
                    <width>320</width>
                    <height>240</height>
                    <format>R8G8B8</format>
                </image>
                <clip>
                    <near>0.02</near>
                    <far>300</far>
                </clip>
                <noise>
                    <type>gaussian</type>
                    <!-- Noise is sampled independently per pixel on each frame.
                        That pixel's noise value is added to each of its color
                        channels, which at that point lie in the range [0,1]. -->
                    <mean>0.0</mean>
                    <stddev>0.007</stddev>
                </noise>
            </camera>
            <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
                <alwaysOn>true</alwaysOn>
                <updateRate>0.0</updateRate>
                <cameraName>myur5/camera1</cameraName>
                <imageTopicName>image_raw</imageTopicName>
                <cameraInfoTopicName>camera_info</cameraInfoTopicName>
                <frameName>camera_link</frameName>
                <Cx>160</Cx>
                <Cx>120</Cx>
                <hackBaseline>0.07</hackBaseline>
                <distortionK1>0.0</distortionK1>
                <distortionK2>0.0</distortionK2>
                <distortionK3>0.0</distortionK3>
                <distortionT1>0.0</distortionT1>
                <distortionT2>0.0</distortionT2>
            </plugin>
        </sensor>
    </gazebo>
</robot>
```

Imagen 78-Defínción parámetros cámara

De esta descripción podemos destacar una de las primeras líneas, en el comando *<gazebo reference="">*; dentro de ese parámetro especificamos el *link* al que se le van a dar las propiedades de cámara. Además de ese parámetro, hay muchos más en la descripción que sirven para definir qué tipo de cámara simularemos y sus propiedades. Cabe destacar que, en el parámetro, *<imageTopicName>,* se especifica el nombre del *topic,* donde se ubicará la información de la imagen captada. En el parámetro *<cameraInfoTopicName>*, se especifica el nombre del *topic*, que contendrá propiedades de la cámara, y será muy importante para obtener los parámetros intrínsecos de la cámara.

Solo faltaría incluir el siguiente comando en el archivo llamado, *ur5_joint_limited_robot.urdf.xacro*.

```
<xacro:include filename="$(find ur_description)/urdf/ur5.gazebo.xml" />
```

Imagen 79

Añadiendo este comando, incluimos la definición de la configuración de la cámara, en el archivo donde hemos definido el bloque, que simula ser la cámara. Ahora si ejecutamos el siguiente comando, deberíamos ver en la simulación, la cámara.

```
$ roslaunch ur5_gripper_moveit_config demo_gazebo.launch
```

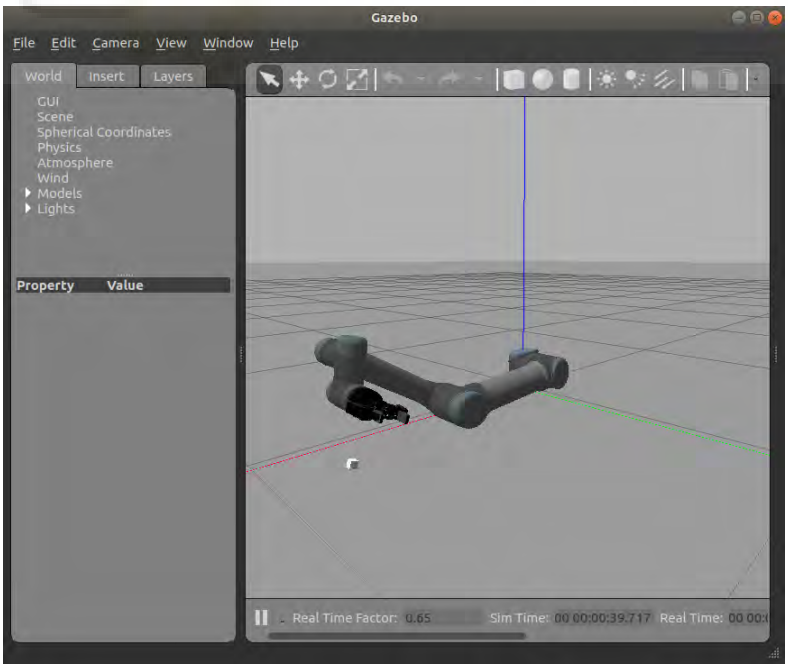


Imagen 80-Simulación en Gazebo con la cámara

Como vemos en la imagen 80, se ha añadido a la simulación el cubo blanco, el cual simula ser la imagen. Con ese cubo, captaremos las imágenes que nos permitirán

aplicar el algoritmo de visión. El proceso de recibir imágenes de la cámara, lo veremos en el apartado 5.2.

5.1.4. Entorno del robot

Hasta ahora, en el apartado 5 hemos visto cómo conseguir una simulación del robot UR5 en *Gazebo*, además, hemos incorporado al robot, tanto una pinza con la que manipular objetos como una cámara, para poder aplicar algoritmos de visión. Por tanto, ya tenemos el modelo del robot como todos los complementos que necesitamos. Pero en esa simulación, también tenemos que añadir más objetos como, por ejemplo, las piezas que queremos manipular con el robot, un soporte donde colocar el robot, entre otras cosas. En este apartado, vamos a explicar todos los objetos, que hemos añadido en el entorno del robot, y que serán necesarios para realizar la aplicación de *pick and place.*

Lo primero que se va a añadir a la simulación de *Gazebo*, es un pedestal, sobre el cual colocaremos el robot. Básicamente, pondremos el robot encima del pedestal, para que este esté posicionado a cierta altura y así pueda manipular mejor las distintas piezas.

El planteamiento, consiste en que el robot quede fijo al pedestal y a su vez, ese pedestal quede fijo a la superficie. Por tanto, el pedestal lo modelaremos en el mismo modelo URDF del brazo robótico, como un *link* más de la simulación. Y con dos *joints*, fijaremos la base del robot al pedestal, y el pedestal a la superficie.

El pedestal lo modelaremos como un cilindro. El código, que define el modelo URDF del pedestal es el siguiente.

```
<link name="pedestal">
    <inertial>
        <origin xyz="0 0 0.5" rpy="0 0 0"/>
        <mass value="20"/>
        <inertia ixx="200"  ixy="200"  ixz="200" iyy="200" iyz="200" izz="200" />
    </inertial>
    <visual>
        <origin xyz="0 0 0.5" rpy="0 0 0"/>
        <geometry>
            <cylinder radius="0.1" length="1"/>
        </geometry>
        <material name="Gray">
        <color rgba="0.5 0.5 0.5 0" />
        </material>
    </visual>
    <collision>
        <origin xyz="0 0 0.5" rpy="0 0 0"/>
        <geometry>
            <cylinder radius="0.1" length="0.5"/>
        </geometry>
    </collision>
</link>
```

Imagen 81-Defínción del link pedestal

Creamos un pedestal, con altura 1m y radio 0.1m. La altura la tendremos en cuenta a la hora de establecer la posición a la que se genera el robot.

Para unir este soporte, tanto con la superficie de la simulación, que en este caso es el *link* llamado *world,* como con la base del robot, necesitamos crear los siguientes dos *joints.*

```
<joint name="world_joint" type="fixed">
    <parent link="world" />
    <child link = "pedestal" />
    <origin xyz="0.0 0.0 0.0" rpy="0.0 0.0 0.0" />
</joint>

<joint name="base_joint" type="fixed">
    <parent link="pedestal" />
    <child link = "base_link" />
    <origin xyz="0.0 0.0 1" rpy="0.0 0.0 0.0" />
</joint>
```

Imagen 82-Los dos joints que fijan el pedestal al suelo y la base del robot al pedestal

Con el *link* y los *joints* definidos, ya tenemos el pedestal incluido en la simulación. Para establecer los coeficientes de fricción en la superficie del pedestal, y darle el color gris oscuro, introduciremos el código que veremos a continuación.

```
<gazebo reference="pedestal">
    <mu1>0.2</mu1>
    <mu2>0.2</mu2>
    <material>Gazebo/DarkGrey</material>
</gazebo>
```

Imagen 83-Definición parámetros pedestal

Ahora mismo, podemos lanzar la simulación de *Gazebo,* con el mismo comando que lo hicimos en el apartado anterior.

```
$ roslaunch ur5_gripper_moveit_config demo_gazebo.launch
```

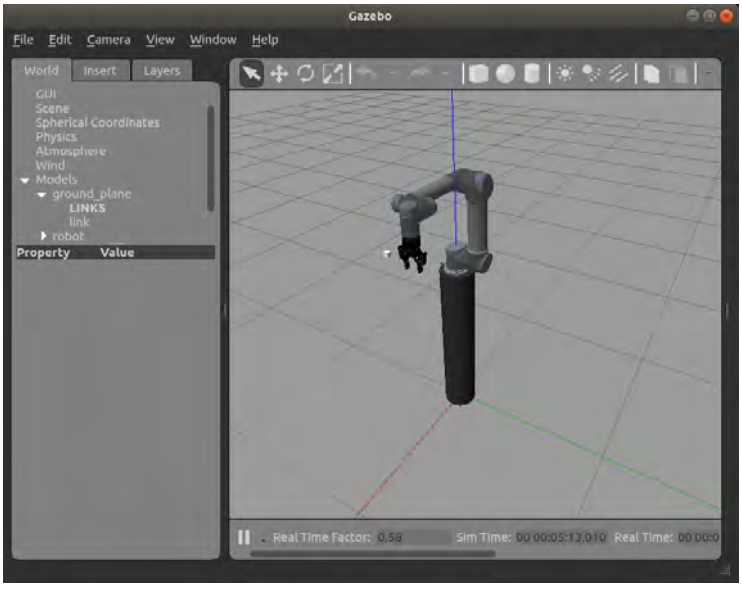


Imagen 84-Simulación del robot con el pedestal

Para completar el espacio de simulación de *Gazebo*, primero añadiremos tres soportes, que simplemente serán tres cubos que vienen predefinidos en el simulador como *box*, y le aumentaremos los valores de fricción de su superficie. Añadiremos estos soportes, desde la interfaz del simulador.

También añadiremos las piezas, que manipulará el robot. Como antes, serán cubos, los cuales los podemos introducir desde la interfaz de *Gazebo.*

Por último, introduciremos a la simulación una mesa, sobre la cual estarán las piezas a manipular. En *Gazebo*, no vienen predefinidas mesas, por tanto, tenemos que buscar su modelo en un repositorio.

Utilizaremos el repositorio, *https://github.com/osrf/gazebo_models.git*. De él, utilizaremos el objeto llamado *Cafe Table.*

La simulación completa, quedaría de la siguiente manera.

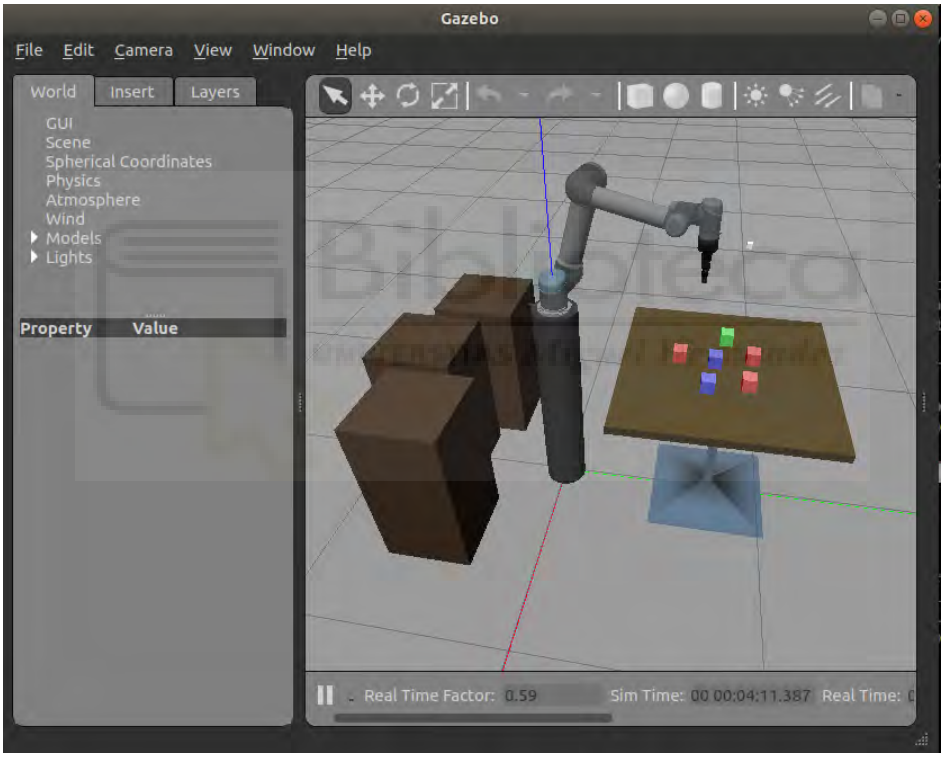


Imagen 85-Entorno de simulación de la aplicación de pick and place

5.2. Software creado en Python

En esta sección, hablaremos de todo el código que se ha desarrollado utilizando *Python*. Entre otras cosas, se comentarán las comunicaciones, dentro del sistema de ROS, que se han realizado para obtener la imagen de la cámara y también se explicará el uso que se le ha dado a la interfaz que ofrece el nodo *move_group*, para comandar el robot por medio de *Python.*

5.2.1. Nodo de captación y procesamiento de imagen

Antes de realizar tanto la aplicación manual como el *pick and* place, en las que movemos el robot, tenemos que plantearnos el procedimiento mediante el cual, primero captaremos la imagen del escenario donde se ubiquen las piezas a manipular y luego, el procedimiento mediante el cual apliquemos un algoritmo a la imagen, para clasificar las piezas.

Lo primero que se va a mostrar son, las librerías que se han integrado en el programa de captación y procesamiento de imagen.

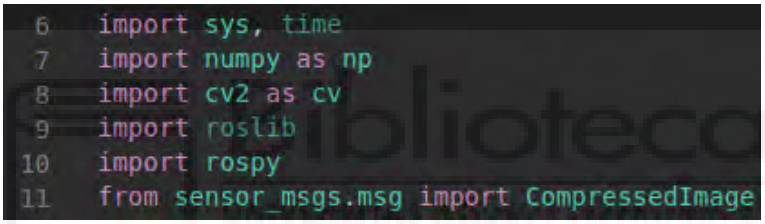


Imagen 86-Librerías programa procesamiento imagen

Si recordamos la sección 5.1.3, en esa sección mostramos como se añade la cámara a la simulación de *Gazebo.* Para ello, entre otras cosas, teníamos que describir un complemento de *Gazebo* del tipo cámara, como podemos ver en la imagen 78. Uno de los parámetros que teníamos que definir se llamaba, *<image_topic_name>*; ese parámetro define el nombre del *topic* que contiene la información de las imágenes recogidas por la cámara.

Por tanto, para la captación de la imagen, necesitamos crear un nodo que sea *subscriber* del *topic* llamado *image_raw*.

Este nodo, de captación y procesamiento de la imagen, lo vamos a crear en un archivo de programación a parte y programándolo dentro de una clase de *Pyhton*, con el objetivo de poder acceder a los datos que resulten del procesamiento de la imagen, desde cualquier otro programa, como veremos en los siguientes apartados.

Para saber el nombre completo, del *topic* al que tenemos que acceder para captar la imagen, ejecutaremos la simulación del robot junto a la cámara y observaremos los *topics* que surgen, y que tienen relación con *image_raw*.

```
$ rostopic list
```

```
/myur5/camera1/camera_info
/myur5/camera1/image_raw
/myur5/camera1/image_raw/compressed
/myur5/camera1/image_raw/compressed/parameter_descriptions
/myur5/camera1/image_raw/compressed/parameter_updates
/myur5/camera1/image_raw/compressedDepth
/myur5/camera1/image_raw/compressedDepth/parameter_descriptions
/myur5/camera1/image_raw/compressedDepth/parameter_updates
/myur5/camera1/image_raw/theora
/myur5/camera1/image_raw/theora/parameter_descriptions
/myur5/camera1/image_raw/theora/parameter_updates
/myur5/camera1/parameter_descriptions
/myur5/camera1/parameter_updates
```

Imagen 87-Lista de topics que se generan cuando se simula la cámara

/myur5/camera1/image_raw/compressed es el *topic*, que almacena la imagen. Por tanto, ya sabemos que tenemos que inicializar un nodo que sea *subscriber* de ese *topic* en concreto. El tipo de datos, que se pueden encontrar dentro ese *topic* son del tipo, *CompressedImage*. Podemos consultar el tipo de dato de un *topic*, con el siguiente comando.

```
$ rostopic info /myur5/camera1/image_raw/compressed
```

Ahora que sabemos todo lo anterior, podemos obtener la imagen que capta la cámara, iniciando un nodo que se suscriba a ese *topic* en concreto. A continuación, mostraremos, la programación que conlleva esa captación.

```
91  def main(args):
92      """Initializes and clearup ros node"""
93      ic = image_read()
94      rospy.init_node('image_read', anonymous=True)
95      try:
96          rospy.spin()
97      except KeyboardInterrupt:
98          print('Shutting down the ROS Image Reader Node')
99          cv.destroyAllWindows()
100
101 if __name__ == '__main__':
102     main(sys.argv)
```

Imagen 88-Inicialización del nodo de procesamiento de imagen

```
15  class image_read:
16      def __init__(self):
17          # Define the subscriber topic
18          self.subscriber = rospy.Subscriber(("/myur5/camera1/image_raw/compressed"),
19                                             CompressedImage, self.callback, queue_size=1)
20
21      def callback(self, ros_data):
```

Imagen 89-Subscripción del nodo al topic en cuestión

En estas imágenes podemos ver la comunicación de un nodo *subscriber*, con el *topic* que estábamos comentando anteriormente. Una vez se inicializa el nodo, se ejecuta continuamente la subscripción y, por tanto, se llama continuamente a la función llamada, *callback*. La información obtenida del *topic*, que será la imagen tomada por la cámara, se almacenará en el parámetro *ros_data.*

Ya hemos realizado la captación de la imagen. Dentro de la función *callback*, es donde realizaremos el procesamiento de dicha imagen. Si quisiéramos ver la imagen que hemos captado podríamos, dentro de *callback*, ejecutar lo siguiente.

```
np_arr = np.fromstring(ros_data.data, np.uint8)
frame = cv.imdecode(np_arr, cv.IMREAD_COLOR)

cv.imshow('image', frame)
```

Imagen 90-Visualización de la imagen de la cámara

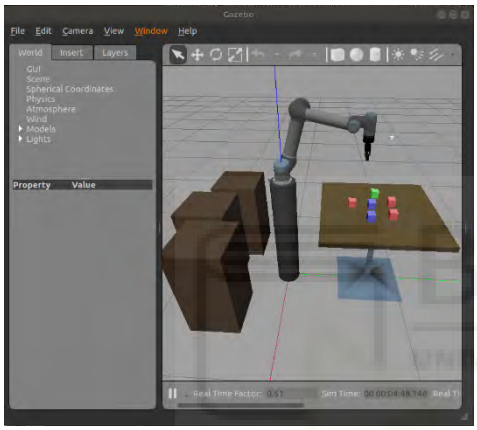


Imagen 91-Posición robot

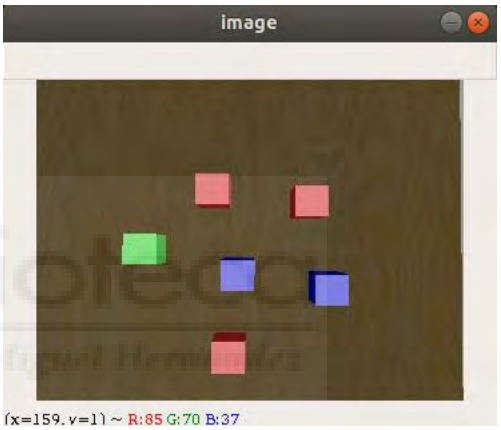


Imagen 92-Imagen visualizada por la cámara

Ahora ya vamos a pasar a explicar el algoritmo de visión por computador, que se le ha aplicado a la imagen llamada *frame*, para clasificar los objetos por color. La primera parte del algoritmo, consiste en segmentar las regiones de la imagen, que contengan los colores que queremos detectar. La parte del código que realiza esta segmentación por colores es la siguiente.

```
49          # ROJO
50          redBajo1 = np.array([0, 100, 20], np.uint8)
51          redAlto1 = np.array([8, 255, 255], np.uint8)
52          redBajo2 = np.array([175, 100, 20], np.uint8)
53          redAlto2 = np.array([179, 255, 255], np.uint8)
54
55          # AZUL
56          blueBajo = np.array([100, 100, 20], np.uint8)
57          blueAlto = np.array([125, 255, 255], np.uint8)
58
59          # VERDE
60          greenBajo = np.array([45, 100, 20], np.uint8)
61          greenAlto = np.array([95, 255, 255], np.uint8)
```

Imagen 93-Rangos de segmentación

```
66          hsv = cv.cvtColor(frame, cv.COLOR_BGR2HSV)
67          maskRed1 = cv.inRange(hsv, redBajo1, redAlto1)
68          maskRed2 = cv.inRange(hsv, redBajo2, redAlto2)
69          maskRed = cv.bitwise_or(maskRed1, maskRed2)
70          maskBlue = cv.inRange(hsv, blueBajo, blueAlto)
71          maskGreen = cv.inRange(hsv, greenBajo, greenAlto)
```

Imagen 94-Segmentación de la imagen por color

Como vemos en la imagen 93, para segmentar por color, primero tenemos que especificar el rango de valores, de los canales *hsv*, que queremos segmentar para cada color. Seguidamente debemos pasar el *frame* original al espacio de color *hsv,* como podemos ver en la imagen 94.

Luego, con la función *cv.inRange()* segmentamos los colores de la imagen. Por ejemplo, la variable *maskBlue*, será una imagen binaria en la que solo estarán activos los píxeles en los que exista el color azul.

El resto de el algoritmo de visión se encapsula en una función llamada, dibujar.

```
24      def dibujar(mask, color):
25          n = 0
26          coord = []
27          contours, hiterachy = cv.findContours(mask, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_NONE)
28
29          for c in contours:
30              area = cv.contourArea(c)
31              if area > 100:
32                  n = n + 1
33                  m = cv.moments(c)
34                  if m["m00"] == 0:
35                      m["m00"] = 1
36                  x = int(m["m10"] / m["m00"])
37                  y = int(m["m01"] / m["m00"])
38                  cv.circle(frame, (x, y), 3, color, -1)
39                  font = cv.FONT_HERSHEY_SIMPLEX
40                  cv.putText(frame, "(" + str(x) + ", " + str(y) + ")", (x + 28, y), font, 0.5, color, 1, cv.LINE_AA)
41                  convexhull = cv.convexHull(c)
42                  cv.drawContours(frame, [convexhull], 0, color, 3)
43                  coord.append(x)
44                  coord.append(y)
45          return n, coord
```

Imagen 95-Función dibujar

Esta función recibe dos parámetros, el código RGB del color que se va a detectar y una de las máscaras de segmentación, de la imagen 94, correspondiente al color que se va a detectar. Utilizando la máscara de segmentación, se detectan los contornos de todas las piezas del color de la imagen y, además, se calcula la posición, en píxeles, en la que se encuentran todas las piezas. Sobre la imagen original, *frame*, se dibuja tanto el contorno, como la posición de cada pieza.

Esta función devuelve, la variable n, que representa el número de piezas que existen de cada color y la variable coord, que contiene las coordenadas x e y, de todas las piezas que existan de ese color en concreto.

La llamada a la función dibujar se realizará de la siguiente manera.

```
77          n_b, piec_blue = dibujar(maskBlue, (255, 0, 0))
78          coordenates.append(piec_blue)
79          n_g, piec_green = dibujar(maskGreen, (0, 255, 0))
80          coordenates.append(piec_green)
81          n_r, piec_red = dibujar(maskRed, (0, 0, 255))
82          coordenates.append(piec_red)
83
84          cv.imshow('frame', frame)
```

Imagen 96-Ejecución función dibujar

Al ejecutarse esa parte del programa, donde se llama a la función dibujar, se podrá visualizar lo siguiente.

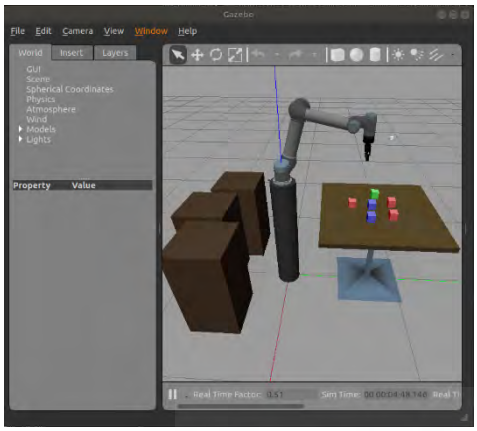


Imagen 97-Posición robot

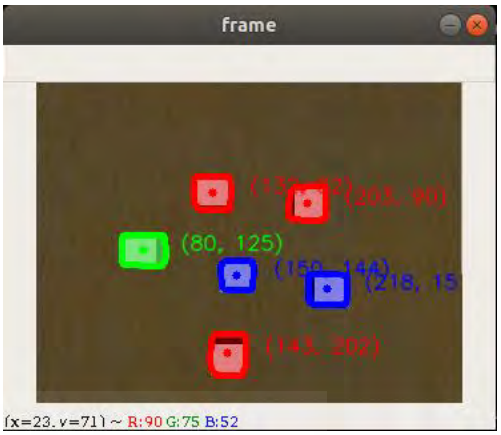


Imagen 98-Resultados algoritmo de visión

Ya tenemos realizado el algoritmo que clasifica las piezas según su color y averigua su posición en la imagen.

Ahora debemos programar la forma de que, desde otro nodo de ROS, podamos obtener la información que nos da este algoritmo.

Si nos fijamos en la imagen 96, las variables *n_b, n_g* y *n_r* almacenan el número de piezas azules, verdes y rojas respectivamente. La variable llamada *coordenates*, de la imagen 95, es un vector de vectores que contiene las coordenadas de las piezas en la imagen. Aprovechando que estamos programando este nodo por clases, haremos la siguiente función, dentro de la clase *image_read.*

```
87      def rev_coord(self):
88          global coordenates, n_b, n_g, n_r
89          return n_b, n_g, n_r, coordenates
```

Image 99-Función de la clase image_read para recibir la información del algoritmo

Creando un objeto de la clase *image_read*, desde otro nodo de ROS y ejecutando la función *rev_coord*, recibiremos tanto las coordenadas de las piezas, como el número de piezas de cada color, que existen en la imagen.

5.2.2. Aplicación manual de control del robot

El objetivo principal de aplicación de este proyecto, es el de realizar la aplicación de *pick and place*. Pero antes de realizar dicha aplicación, era necesario aprender todas las funcionalidades del nodo *move_group*, que nos pudieran ser útiles a la hora de mover el robot. Por tanto, con el fin de buscar una mayor familiarización con la interfaz, *moveit_commander* de *Python*, se propuso como otro objetivo el realizar una aplicación manual de control del robot.

Este programa, en el momento de ejecutarse, nos debe ofrecer una interfaz de usuario que contendrá una serie de opciones para manipular el robot. Con esto, logramos englobar todas las funcionalidades del nodo *moveit_commander* que podemos necesitar en el proyecto, en una sola aplicación.

Empezaremos por mostrar las librerías que se han añadido al programa de esta aplicación, las cuales las podremos ver a continuación.

```
3  import sys
4  import copy
5  import rospy
6  import numpy as np
7  import math as m
8  import moveit_commander
9  import moveit_msgs.msg
10 import geometry_msgs.msg
11 from math import pi
12 from std_msgs.msg import String
13 from moveit_commander.conversions import pose_to_list
14 from moveit_msgs.msg import DisplayTrajectory
15 import cv2 as cv
16 from pynput import keyboard
17 from read_camera_image_mult import image_read
```

Imagen 100-Librerías programa modo manual

Hay breves argumentos sobre las funcionalidades de esas librerías en la sección 4.5 de la memoria.

Luego de ver las librerías que se han añadido al programa, debemos inicializar tanto el nodo en el cual se va a ejecutar el programa como la interfaz *moveit*_commander, para poder utilizar sus funcionalidades para mover el robot. Ambas inicializaciones estarán en el *main* del programa.

```
135     # initialize moveit_comander and rospy node
136     moveit_commander.roscpp_initialize(sys.argv)
137     rospy.init_node('move_group_python_interface', anonymous=True)
```

Imagen 101-Inicialización nodo y moveit_commander

Como hemos se ha expuesto en el apartado 5.2.1, el procesamiento de la imagen se ha realizado en una clase llamada *image*_read y dentro de esa clase había una función que nos da la información que necesitamos de dicha imagen; esa función

se llama *rev_coord().* Por tanto, también se creará un objeto de la clase *image_read*, con el que podremos acceder a los datos obtenidos de las imágenes.

```
132     obj_img = image_read()
```

Imagen 102-Creación objeto de la clase image_read

Una vez tenemos iniciados tanto el nodo de la aplicación manual, como la interfaz de *move_group* para *Python*, debemos crear dos objetos de la clase *MoveGroupCommander.* Dicha clase está dentro de la interfaz *moveit_commander*, y es la que contiene todas las funciones que nos permitirán mover al robot.

```
142     # create a MoveGroupCommander object
143     # we will move the arm or the gripper with these objects
144     group_name_1 = "ur5_arm"
145     group_name_2 = "gripper"
146     move_group_interface_arm = moveit_commander.move_group.MoveGroupCommander(group_name_1)
147     move_group_interface_gripper = moveit_commander.move_group.MoveGroupCommander(group_name_2)
```

Imagen 103-Creación objetos MoveGroupCommander

Mediante el uso de esos objetos, seremos capaces de mover tanto el brazo robótico como la pinza. Respecto a los nombres, *ur5_arm* es el nombre que se le ha dado al modelo URDF del brazo, y *gripper* es el que se le ha dado al modelo de la pinza.

Después de las inicializaciones y de la creación de los objetos, estarán los condicionales que nos permiten escoger el modo de operación con el que queremos actuar en la simulación.

```
156     while True:
157         print('-----------------------------------------------')
158         print('OPERATING MODES:')
159         print('1: Direct kinematic')
160         print('2: Direct kin - Manual mode')
161         print('3: Inverse kinematic - MoveJ')
162         print('4: Open gripper')
163         print('5: Close gripper')
164         print('6: Go Home')
165         print('7: Go camera position')
166         print('8: See the object coordenates')
167         print('9: Close program')
168         print('-----------------------------------------------')
169         try:
170             num = int(input('\nSelect one mode: '))
171         except:
172             print(rospy.logerr('Bad mode selection'))
173             exit()
174 
175         if num == 1:
176             print("\nDIRECT KINEMATIC")
177             direct_kin()
```

Imagen 104-Condicional selector parte 1

```
178         elif num == 2:
179             print("\nMANUAL MODE")
180             print("*******************************")
181             print("CONTROLS:")
182             print("Positive rotation of joints: 1, 2, 3, 4, 5, 6")
183             print("Negative rotation of joints: q, w, e, r, t, y")
184             print("Open the gripper: press 'o'")
185             print("Close the gripper: press 'c'")
186             print("Position AllZeros: press 'a'")
187             print("Position Home: press 's'")
188             print("See the current joint values: press 'd'")
189             print("Close the manual mode: press 'f'")
190             print("*******************************")
191             current_joint_values = move_group_interface_arm.get_current_joint_values()
192             new_values = current_joint_values
193             with keyboard.Listener(
194                     on_press=on_press) as listener:
195                 listener.join()
196             print("Manual mode closed\n")
197         elif num == 3:
198             print('\nINVERSE KINEMATIC')
199             inverse_kin()
200         elif num == 4:
201             print("OPENING THE GRIPPER")
202             move_group_interface_gripper.go(gripper_open, wait=True)
203             move_group_interface_gripper.stop()
```

Imagen 105-Condicional selector parte 2

```
204         elif num == 5:
205             print("CLOSING THE GRIPPER")
206             move_group_interface_gripper.go(gripper_close, wait=True)
207             move_group_interface_gripper.stop()
208         elif num == 6:
209             print("MOVING TO HOME POSITION")
210             joint_goals = [0.0, -1.5447, 1.5447, -1.5794, -1.5794, 0.0]
211             move_group_interface_arm.go(joint_goals, wait=True)
212             move_group_interface_arm.stop()
213         elif num == 7:
214             print("MOVING TO THE POSITION WHERE WE PROCESS PICTURES")
215             joint_goals = [1.3909180384024973, -1.2970095837866964, 1.4559604820238743, -1.75, -1.573950195686849, -0.1819244873697139]
216             move_group_interface_arm.go(joint_goals, wait = True)
217             move_group_interface_arm.stop()
218         elif num == 8:
219             coord = []
220             _, _, _, coord = obj_img.rev_coord()
221             print("Coordenadas piezas azules: ({})\nCoordendas piezas verdes: ({})\nCoordenadas piezas rojas: ({})".format(coord[0], coord[1], coord[2]))
222         elif num == 9:
223             print('Finishing the program')
224             exit()
225         else:
226             print(rospy.logerr('Bad mode selection'))
227             exit()
```

Imagen 106-Condicional selector parte 3

Vemos en estas imágenes que hay un condicional en la función *main*, el cual en el momento en el que ejecutemos el programa, nos permitirá elegir las acciones que queremos aplicar sobre el robot, en función del valor introducido por teclado.

Al ejecutarse el programa, por terminal visualizaremos lo siguiente.

```
-------------------------------------------------
OPERATING MODES:
1: Direct kinematic
2: Direct kin - Manual mode
3: Inverse kinematic - MoveJ
4: Open gripper
5: Close gripper
6: Go Home
7: Go camera position
8: See the object coordenates
9: Close program
-------------------------------------------------

Select one mode:
```

Imagen 107-Selector de modo de operación sobre el robot

A continuación, vamos a ver, las funcionalidades de cada uno de estos modos de funcionamiento.

- **Selección modo 4/5**

Esta selección se desarrolló con la intención de aprender cómo mover la pinza y averiguar si podríamos especificar una posición predefinida de pinza abierta o cerrada, con la intención de utilizarlas en el apartado 5.2.3 en la aplicación de *pick and* place.

Al mostrarse la interfaz de la imagen 107 por terminal, si seleccionáramos el modo 4 o el 5, se ejecutarían las siguientes instancias del condicional *if*, de las imágenes 105 y 106.

```
201         elif num == 4:
202             print("OPENING THE GRIPPER")
203             move_group_interface_gripper.go(gripper_open, wait=True)
204             move_group_interface_gripper.stop()
205         elif num == 5:
206             print("CLOSING THE GRIPPER")
207             move_group_interface_gripper.go(gripper_close, wait=True)
208             move_group_interface_gripper.stop()
```

Imagen 108-Funciones de abrir y cerrar pinza

El objeto *move_group_interface*_gripper, nos sirve para acceder a diversas funciones que actúan sobre la pinza. En concreto, si utilizamos la función go, como vemos en la línea 203 del código, podemos mover la pinza. Para mover la pinza con esa función hasta una disposición concreta, dentro de la función *go* especificaremos la posición a la que se va a llevar la pinza; en este caso se han predefinido las posiciones *gripper_open* y *gripper_close* que son las posiciones de pinza abierta y pinza cerrada correspondientemente.

```
128   gripper_open = [0.1]
129   gripper_close = [0.25]
```

Imagen 109-Posiciones pinza predefinidas

En el apartado 5.1, se mostrarán los movimientos de la pinza en simulación que ocurren al seleccionar estos dos modos y todos los demás.

- **Selección modo 6/7**

Tanto el modo 6 como el 7 se programaron con la intención de conocer, si utilizando la cinemática directa que nos ofrece *moveit_commander*, era posible predefinir ciertas posiciones con unas líneas de códigos, las cuales siempre que se ejecutarán llevarían al robot a la misma posición.

Si seleccionáramos el modo 6, en la interfaz de la imagen 107 se ejecutaría el siguiente código.

```
209         elif num == 6:
210             print("MOVING TO HOME POSITION")
211             joint_goals = [0.0, -1.5447, 1.5447, -1.5794, -1.5794, 0.0]
212             move_group_interface_arm.go(joint_goals, wait=True)
213             move_group_interface_arm.stop()
```

Imagen 110-Código que mueve al robot hasta Home

El objeto *move_group_interface_*arm nos sirve para aplicar funciones sobre el brazo robótico. En concreto, la función *go* de ese objeto sirve para mover el brazo hasta una posición predefinida. En este caso se va a mover el brazo usando cinemática directa, ya que la posición a la que irá el robot se define con el vector *joint_*goals, el cual define las posiciones angulares a las que irán todas las articulaciones.

Por tanto, en este caso al ejecutarse la función *go*, las articulaciones del robot se moverán hasta las posiciones angulares especificadas en el vector *joint_goals.* La posición a la que se dirige el robot al seleccionarse este modo, se ha llamado *Home*.

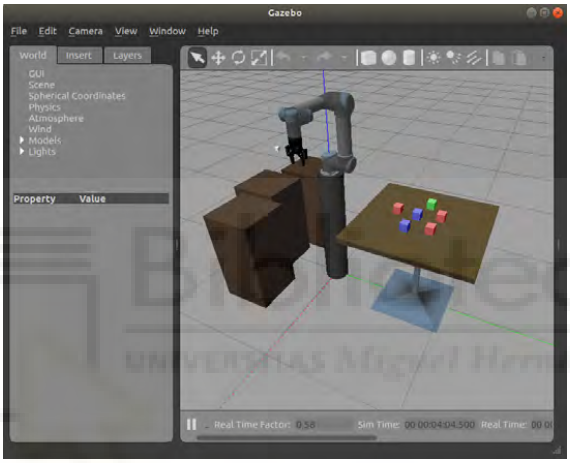


Imagen 111-Robot en posición Home

Como apreciamos en la imagen 106, si el valor condicional cumple que es igual a 7, el código que ejecutamos es idéntico al de la imagen 110, pero cambiando el vector *joint_goals*. Eso significa, que el modo 7 utilizará también cinemática directa para llevará al robot a la posición que marque dicho vector, pero será distinta a la del modo 6.

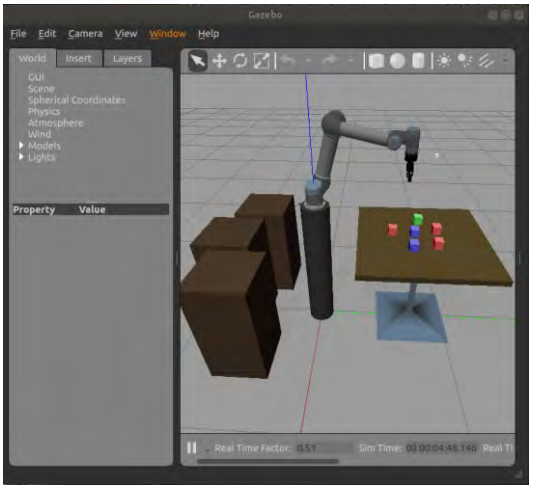


Imagen 112-Robot en posición del modo 7

En el caso de la imagen 112, el robot va a la posición en la cual se deben tomar las imágenes de las piezas de la mesa. Desde esta posición se detectarán las piezas en la aplicación de *pick and place* del apartado 5.2.3.

- **Selección modo 8**

Se incluyó el modo 8 en la aplicación manual para aprender cómo utilizar un objeto de la clase *image_read*, que hemos creado en la imagen 102, para acceder a los datos de las piezas.

En este modo simplemente se van a recibir las coordenadas, de las diferentes piezas que se encuentren en la mesa, desde el programa principal y se van a mostrar por pantalla. Nos va a servir como aprendizaje para, en la aplicación de *pick and place* del apartado 5.2.3, utilizarás para recoger las piezas.

Por tanto, como podemos observar en el código de la imagen 106, si seleccionáramos el modo 8, se obtienen las coordenadas de las piezas accediendo a la función *rev_coord()*, del objeto *obj_img* de la clase *image_read*.

Debemos tener en cuenta, que antes de detectar las coordenadas de las piezas de la mesa, el robot debe estar en la posición en la cual se pueden detectar dichas piezas, a la cual podemos mandar el robot si seleccionamos el modo 7. Dicho de otro modo, el robot y la cámara deben estar como en la imagen 97 y 98, para que funcione correctamente el modo 8.

En la sección 6.1 se verá un ejemplo de la información que se muestra por terminal al seleccionar este modo.

- **Selección modo 1**

Si seleccionamos el modo 1, se ejecutará la función *direct_kin(),* como podemos ver en la imagen 104.

```
19  def direct_kin():
20      current_joint_values = move_group_interface_arm.get_current_joint_values()
21      print('Current joint values:')
22      print(current_joint_values)
23      print('\nEnter new values:\n')
24      try:
25          joint_goals = [float(input("Enter joint " + str(i) + " value: ")) for i in range(6)]
26      except:
27          print("Error: bad joint_goals")
28          exit()
29      move_group_interface_arm.go(joint_goals, wait = True)
30      print("New goals for the robot: " + str(joint_goals))
31      move_group_interface_arm.stop
```

Imagen 113-Función direct_kin()

Esta función surgió para experimentar con la capacidad de cinemática directa de la interfaz de *move*_group. En la línea 20 del código de, se utiliza el objeto *move_group_interface_arm* para acceder a las funciones que actúan sobre el brazo. A través de dicho objeto, se accede a la función *get_current_joint_values()*, la cual nos da un vector con las posiciones angulares actuales de las articulaciones del robot.

Seguidamente, en la línea 25 pide al usuario las posiciones angulares que deben alcanzar las seis articulaciones del robot y las almacenará en el vector llamado *joint_goals*.

Utilizando el objeto *move_group_interface_arm* se accederá a la función go, especificando como parámetro el vector *joint*_goals. De este modo se mueve al robot a la posición especificada por el usuario utilizando la cinemática directa. Un ejemplo de lo que mostraría esta función sería el siguiente.

```
Select one mode: 1

DIRECT KINEMATIC
Current joint values:
[0.01776891360981292, -1.5531149572377263, 1.5483
128530586416, -1.5697156641054972, -1.57967276596
68202, -0.0005725050657083841]

Enter new values:

Enter joint 0 value: pi
Enter joint 1 value: 0
Enter joint 2 value: -pi/4
Enter joint 3 value: 0
Enter joint 4 value: pi/3
Enter joint 5 value: 0
```

Imagen 114-Interfaz direct_kin()

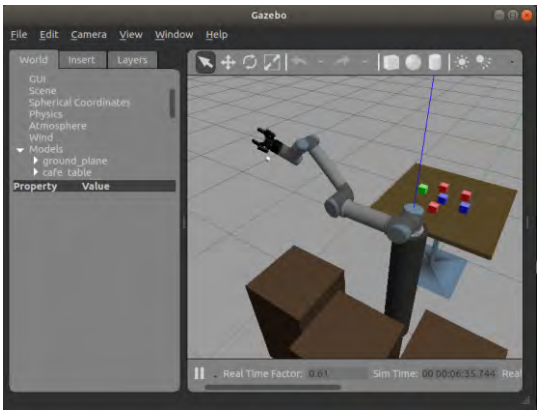


Imagen 115-Movimiento con direct_kin()

- **Selección modo 3**

Si seleccionáramos el modo 3, se ejecutaría la función *inverse_kin()*.

```
102 def inverse_kin():
103     current_pose = move_group_interface_arm.get_current_pose()
104     print("Current values of end effector pose:")
105     print(current_pose)
106     print('\nEnter new end effector values:\n')
107 
108     pose_target = geometry_msgs.msg.Pose()
109     pose_target.position.x = float(input('Enter position x: '))
110     pose_target.position.y = float(input('Enter position y: '))
111     pose_target.position.z = float(input('Enter position z: '))
112     z = int(input("Do you want to enter a quaternion?(Yes(1) or No(2)): "))
113     if z == 1:
114         pose_target.orientation.w = float(input('Enter value w: '))
115         pose_target.orientation.x = float(input('Enter value x: '))
116         pose_target.orientation.y = float(input('Enter value y: '))
117         pose_target.orientation.z = float(input('Enter value z: '))
118     elif z == 2:
119         pose_target.orientation = current_pose.pose.orientation
120 
121     move_group_interface_arm.set_pose_target(pose_target)
122 
123     move_group_interface_arm.go(wait=True)
124 
125     move_group_interface_arm.stop()
126     move_group_interface_arm.clear_pose_targets()
```

Imagen 116-Función inverse_kin()

Esta función tenía como objetivo investigar cómo comandar al robot mediante cinemática inversa. Utilizando la interfaz de *moveit_commander.*

Como se puede observar en la imagen, en este caso en la línea 103 se utiliza la función *get_current_pose()*, que en este caso no recibe el valor de las articulaciones si

no que recibe la pose actual del robot, es decir la posición y orientación del extremos del robot.

Para comandar el robot por cinemática inversa, se crea un mensaje del tipo Pose, que se llamará *pose*_target. Dentro de ese mensaje se puede especificar la posición x, y, z a la que queremos llevar el extremos robot y, además, se puede especificar con un cuaternión la orientación de dichos extremos.

Tanto la posición como la orientación se piden por teclado al usuario, y se almacenan en el mensaje *pose_target*. Luego, en la línea 121 utilizando *set_pose_targe(pose_target)*, se establece la pose a la que va a ir el robot cuando seguidamente se ejecuta la función *go().* Podemos ver un ejemplo de uso de la función a continuación.

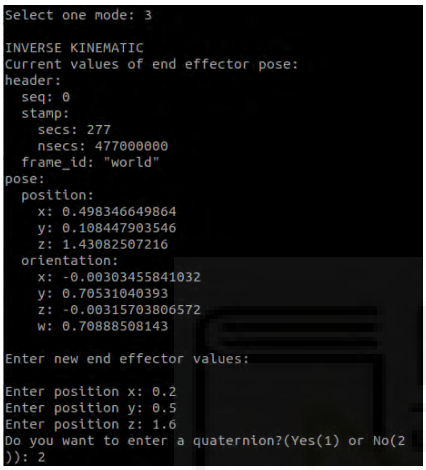


Imagen 117-Interfaz inverse_kin()

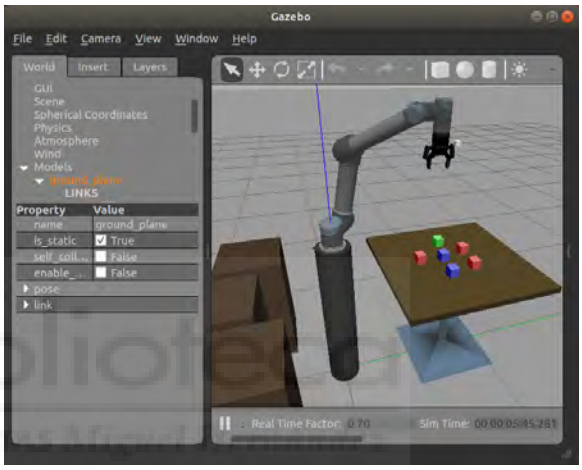


Imagen 118-Movimiento con inverse_kin()

- **Selección modo 2**

Por último, se ha de destacar el modo 2, el cual es un modo manual en el cual el usuario puede mover el robot, utilizando algunas teclas del teclado. Como se puede ver en la imagen 105, se utiliza la clase *Keyboard*, de la librería *pynput*, para registrar las teclas pulsadas. Al pulsar alguna tecla del teclado se ejecuta la función *on_press,* la cual podemos verla a continuación.

```python
def on_press(key):
    global new_values
    try:
        current_joint_values = move_group_interface_arm.get_current_joint_values()
        # print(current_joint_values)
        if key.char == "1":
            new_values[0] += 0.1121997376
            new_values[1:] = current_joint_values[1:]
        elif key.char == "2":
            new_values[0] = current_joint_values[0]
            new_values[1] += 0.1121997376
            new_values[2:] = current_joint_values[2:]
        elif key.char == "3":
            new_values[:1] = current_joint_values[:1]
            new_values[2] += 0.1121997376
            new_values[3:] = current_joint_values[3:]
        elif key.char == "4":
            new_values[:2] = current_joint_values[:2]
            new_values[3] += 0.1121997376
            new_values[4:] = current_joint_values[4:]
        elif key.char == "5":
            new_values[:3] = current_joint_values[:3]
            new_values[4] += 0.1121997376
            new_values[5] = current_joint_values[5]
        elif key.char == "6":
            new_values[:4] = current_joint_values[:4]
            new_values[5] += 0.1121997376
        elif key.char == "q":
            new_values[0] -= 0.1121997376
            new_values[1:] = current_joint_values[1:]
        elif key.char == "w":
            new_values[0] = current_joint_values[0]
            new_values[1] -= 0.1121997376
            new_values[2:] = current_joint_values[2:]
        elif key.char == "e":
            new_values[:1] = current_joint_values[:1]
            new_values[2] -= 0.1121997376
            new_values[3:] = current_joint_values[3:]
        elif key.char == "r":
            new_values[:2] = current_joint_values[:2]
            new_values[3] -= 0.1121997376
            new_values[4:] = current_joint_values[4:]
        elif key.char == "t":
            new_values[:3] = current_joint_values[:3]
            new_values[4] -= 0.1121997376
            new_values[5] = current_joint_values[5]
        elif key.char == "y":
            new_values[:4] = current_joint_values[:4]
            new_values[5] -= 0.1121997376
        elif key.char == "a":
            new_values = [0, 0, 0, 0, 0, 0]
        elif key.char == "s":
            new_values = [0, -1.4981833546922498, 1.5076458045811982, -0.0019684415915168785, 0, 0]
        elif key.char == "d":
            print('Current joint values:')
            print(current_joint_values)
        elif key.char == "o":
            move_group_interface_gripper.go(gripper_open, wait=True)
            move_group_interface_gripper.stop()
        elif key.char == "c":
            move_group_interface_gripper.go(gripper_close, wait=True)
            move_group_interface_gripper.stop()
        elif key.char == "f":
            sys.exit()
        move_group_interface_arm.go(new_values, wait = True)
        move_group_interface_arm.stop
    except AttributeError:
        print("You shouldn't press special characters")
```

Imagen 119-Función on_press()

Siempre que ejecutemos este modo, por terminal se mostrará la siguiente información.

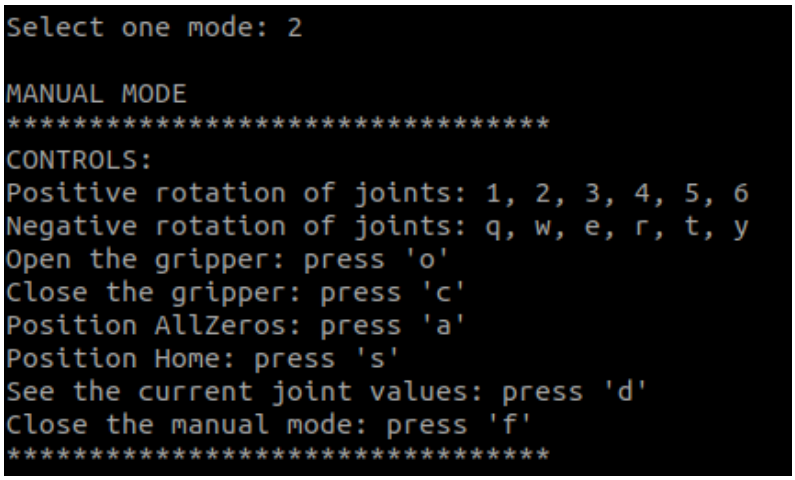


Imagen 120-Paleta de usuario para mover el robot con el modo 2

En el terminal, como vemos en la imagen 120, se muestran todas las teclas del teclado que se pueden utilizar y la funcionalidad que tiene cada una de estas. En el momento de la ejecución del modo 2, ya se estará registrando los valores de teclado que pulsemos, y todos los movimientos que realizará el robot al pulsar una tecla se realizarán en tiempo real en el momento que pulsemos dicha tecla.

Pulsando las teclas del 1 al 6, podremos mover las articulaciones desde la primera hasta la última respectivamente, en el sentido positivo de la articulación. Las teclas de la q a la y, servirán para lo mismo, pero en sentido negativo.

Las teclas “o” y “c”, abrirán y cerrarán la pinza respectivamente. La tecla “a”, llevará al robot a una posición predefinida que se llama *AllZeros*. Por su parte, al pulsar “s” se llevará al robot a la posición *Home.* La tecla “d”, mostrará los valores actuales de las articulaciones del brazo. Por último, este modo cerrará cuando se pulse la tecla “f”.

Cabe destacar, que lo remarcable de este modo de operación es, que el movimiento del robot se inferirá a las articulaciones del robot, directamente desde el teclado, por tanto, estaremos manejando el robot en tiempo real.

Se mostrarán ejemplos del funcionamiento de esta aplicación en el apartado 6.1.

5.2.3. Pick and place con clasificación por color

En el apartado 5.2.3 se mostrará todo el procedimiento que se ha llevado a cabo para realizar la aplicación final que cumple el objetivo principal de aplicación de este proyecto, desarrollar una aplicación de *pick and place* utilizando el robot UR5 y clasificando las piezas según su color.

Se debe tener en cuenta que el nodo de captación y procesamiento de la imagen ya se ha realizado en el apartado 5.2.1. En esta sección se va a desarrollar otro nodo distinto, que entre otras cosas va a acceder a la información del nodo de captación y

procesamiento, para utilizar dicha información con el objetivo de saber la ubicación de las piezas y su color.

5.2.3.1. Aplicación de pick and place

En este apartado se va a exponer, el desarrollo de la aplicación de *pick and* place, comandando el robot en simulación mediante la interfaz de *MoveIt* llamada *moveit_commander*, que nos permitía mover al robot en simulación utilizando código *Python.*

Las librerías que vamos a incluir en el programa de esta aplicación, son las siguientes.

```
3  import sys
4  import copy
5  import rospy
6  import numpy as np
7  import math as m
8  import moveit_commander
9  import moveit_msgs.msg
10 import geometry_msgs.msg
11 import shape_msgs.msg
12 from math import pi
13 from std_msgs.msg import String
14 from moveit_commander.conversions import pose_to_list
15 from moveit_msgs.msg import DisplayTrajectory
16 import cv2 as cv
17 from pynput import keyboard
18 from read_camera_image_mult import image_read
19 import time as t
```

Imagen 121-Librerías aplicación pick and place

Lo primero que debemos hacer, es inicializar tanto el nodo en el cual se va a ejecutar el programa con la aplicación de *pick and place*, como la interfaz *moveit_commander*, para poder utilizar sus funciones para manipular el robot. Esas dos inicializaciones se realizarán de forma idéntica a como ya se hicieron en el apartado 5.2.2 para la aplicación manual. Podemos ver como se codifica esas inicializaciones en la imagen 101.

Una vez iniciados tanto la interfaz como el nodo, debemos crear los dos objetos de la clase *MoveGroupCommander,* que están dentro de la librería *moveit*_commander.

```
262  # create a MoveGroupCommander object
263  # we will move the arm or the gripper with these objects
264  group_name_1 = "ur5_arm"
265  group_name_2 = "gripper"
266  move_group_interface_arm = moveit_commander.move_group.MoveGroupCommander(group_name_1)
267  move_group_interface_gripper = moveit_commander.move_group.MoveGroupCommander(group_name_2)
```

Imagen 122-Objetos para mover el robot

Como ya sabemos de la aplicación manual, con el objeto *move_group_interface_arm* accederemos a las funciones que actúan sobre el brazo y

con *move_group_interface_gripper* podemos utilizar las funciones que actúan sobre la pinza.

- **Desarrollo de las funciones de movimiento del robot**

El código de la aplicación se ha estructurado en funciones, lo cual facilitará su comprensión, la corrección de errores y además facilitará la reutilización del código para futuras aplicaciones. Existen, en el código, cinco funciones las cuales utilizan la cinemática directa para mover el robot.

```
38  def home_pos():
39      print("MOVING TO HOME POSITION...")
40      joint_goals = [0.0, -1.5447, 1.5447, -1.5794, -1.5794, 0.0]
41      move_group_interface_arm.go(joint_goals, wait=True)
42      move_group_interface_arm.stop()
43
44  def home_pos_inv():
45      print("MOVING TO HOME POSITION...")
46      joint_goals = [-3.14, -1.5447, 1.5447, -1.5794, -1.5794, 0.0]
47      move_group_interface_arm.go(joint_goals, wait=True)
48      move_group_interface_arm.stop()
```

Imagen 123-Funciones home_pos() y home_pos_inv()

Estas son dos de las cinco funciones que utilizan la cinemática directa. Especifican las posiciones angulares que se quieren alcanzar, mediante el vector *joint_goals*. Utilizando el objeto *move_group_interface_arm*, se ejecuta la función *go*, que mueve las articulaciones del robot hasta la posición especificada.

Nos podemos dar cuenta, de que cada vez que ejecutemos estas funciones desde cualquier lugar del programa, el robot se moverá hasta esas posiciones especificadas por los vectores. Por tanto, estamos predefiniendo posiciones del robot, a las que podremos mover el robot simplemente ejecutando dicha función.

Por ejemplo, si ejecutamos *home_pos*, el robot se movería hasta la siguiente posición.

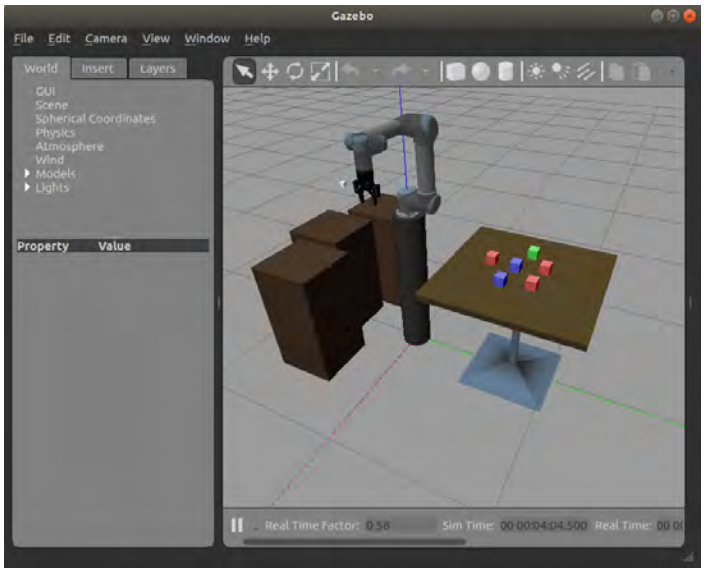


Imagen 124-Posición robot ejecutando home_pos()

Otra función que también mueve el robot, utilizando cinemática directa es la siguiente.

```
29  def pos_take_images():
30      global target_pose_correct_orien
31      print("MOVING TO CAMERA POSITION...")
32      images_goals = [1.3909180384024973, -1.2970095837866964, 1.4559604820238743, -1.75, -1.573950195686849, -0.1819244873697139]
33      move_group_interface_arm.go(images_goals, wait=True)
34      move_group_interface_arm.stop()
```

Imagen 125-Función pos_take_images()

Mediante el vector *images_goals*, se especifican las posiciones angulares de las articulaciones que llevan al robot hasta la posición en la cual se procesarán las imágenes. Por tanto, sabemos que cada vez que ejecutemos esta función desde cualquier lugar, el robot se moverá hasta la posición de tomar imágenes.

Al ejecutarse la función *pos_take_images()*, el robot se moverá hasta la siguiente posición.

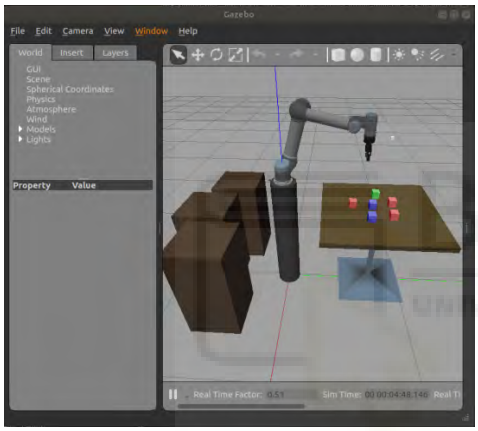


Imagen 126-Robot en posición de tomar imágenes

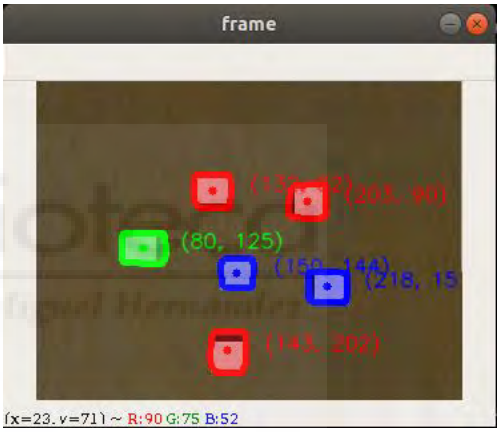


Imagen 127-Imagen de la cámara procesada

Como bien sabemos, podemos mover la pinza utilizando el objeto *move_group_interface_gripper*. Predefiniremos una posición de pinza abierta, *open_gripper*, y una posición de pinza cerrada, *close_gripper*. Estas posiciones de la pinza las utilizaremos para coger y dejar las piezas.

```
50  def close_gripper():
51      print("CLOSING THE GRIPPER...")
52      move_group_interface_gripper.go(gripper_close, wait=True)
53      move_group_interface_gripper.stop()
54
55  def open_gripper():
56      print("OPENING THE GRIPPER...")
57      move_group_interface_gripper.go(gripper_open, wait=True)
58      move_group_interface_gripper.stop()
```

Imagen 128-Función close_gripper() y open_gripper()

Ejecutando cualquiera de las dos funciones, abriremos o cerraremos la pinza para coger o soltar la pieza en cuestión.

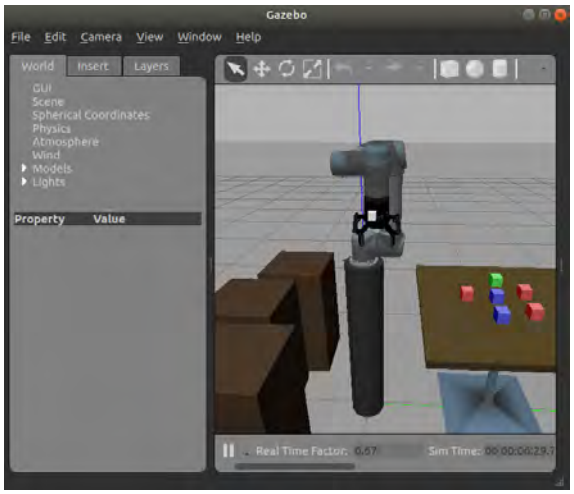


Imagen 129-open_gripper()

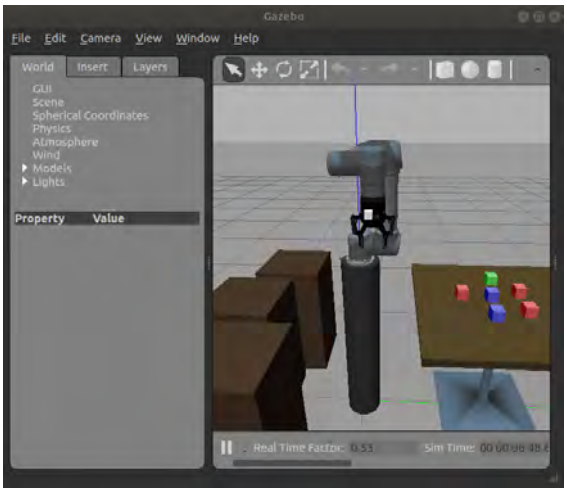


Imagen 130-close_gripper()

Como ya vimos en el apartado 5.2.2, utilizando la clase *MoveGroupCommander,* también se puede mover el robot utilizando cinemática inversa, es decir, especificando directamente la posición y orientación en coordenadas del mundo, a la que queremos llevar el extremo del robot.

Utilizando la cinemática inversa, se ha definido una función que aproximará la pinza del robot sobre una pieza en concreto que se quiera coger.

```
60  # Approximate the gripper over the piece
61  def approximation(x, y):
62      global target_pose, comp_orien, target_pose_correct_orien
63      target_pose.position.x = x
64      target_pose.position.y = y
65      target_pose.position.z = 1.1
66      if comp_orien > 2:
67          target_pose.orientation = target_pose_correct_orien.pose.orientation
68      move_group_interface_arm.set_pose_target(target_pose)
69      move_group_interface_arm.go(wait=True)
70      move_group_interface_arm.stop()
71      comp_orien = comp_orien + 1
```

Imagen 131-Función approximation(x, y)

La altura a la que se encuentran las piezas respecto al eje de coordenadas del mundo es conocido. En el momento que conozcamos la posición x e y, de una pieza, la función *approximation* nos servirá para colocar la pinza sobre dicha pieza. Esto nos servirá para aproximar la pinza a la pieza que se quiera coger, con el objetivo de, en un paso posterior simplemente bajar en el eje z la pinza y coger la pieza.

Por ejemplo, en el siguiente caso la posición de la única pieza colocada sobre la mesa es, (x=0.0071, y=0.7859). Si utilizamos la función *approximation*, pasándole como parámetros esa x e y, el robot se movería hasta la siguiente posición.

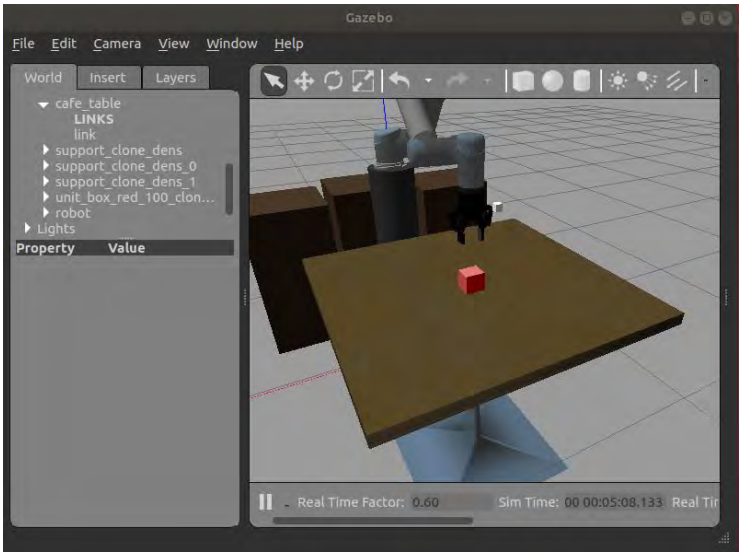


Imagen 132-Ejecución de approximation(x, y)

Como podemos ver en la imagen que, al ejecutar la función *approximation* la pinza se aproxima a la pieza posicionándose en la posición x e y especificadas.

A continuación, necesitamos una función que, al ejecutarse, baje la pinza y se prepare para coger la pieza. La función que realiza esa tarea, a través de la cinemática inversa, es la función llamada *take_piece*.

```
73  # Take the piece with the gripper after approximation
74  def take_piece():
75      global target_pose, comp_orien
76      target_pose.position.z = 0.96
77      move_group_interface_arm.set_pose_target(target_pose)
78      move_group_interface_arm.go(wait=True)
79      move_group_interface_arm.stop()
```

Imagen 133-Función take_piece()

Al ejecutar esa función, si previamente hemos ejecutado *approximation*, moveremos el robot a la siguiente posición.

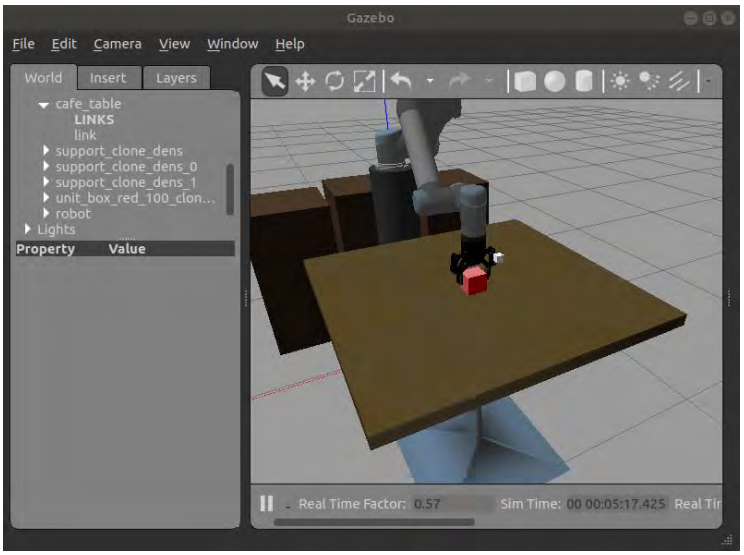


Imagen 134-Ejcución take_piece()

Como se puede observar, una vez llegados a esa posición, solo tendríamos que ejecutar la función *close_gripper()*, para coger la pieza.

En este momento, ya tenemos descritas las funciones que podrán ser utilizadas dentro del programa, para mover el robot a posiciones concretas. Pero antes de utilizarlas para programar el *pick and place*, necesitamos saber la ubicación de las piezas.

- **Transformación de coordenadas de la imagen a coordenadas del mundo.**

Del algoritmo de visión que hemos desarrollado en el apartado 5.2.1, podemos obtener las coordenadas de las piezas existentes en la mesa. El problema es, que las coordenadas de las piezas que recibimos del algoritmo de visión, vienen en coordenadas de la imagen, en píxeles. Por tanto, necesitamos una función que convierta las coordenadas en píxeles de la imagen a coordenadas respecto al eje del mundo, que es el eje respecto al cual se comanda el robot.

Para realizar esa conversión necesitamos analizar las expresiones del modelo *Pin-Hole*. Este modelo nos da las siguientes dos expresiones:

$$\begin{bmatrix} n\,x_f \\ n\,y_f \\ n \end{bmatrix} = \begin{bmatrix} f_x & 0 & C_x & 0 \\ 0 & f_y & C_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

El algoritmo de procesamiento de imágenes, nos proporciona las coordenadas x_f e y_f. Y tenemos arriba una expresión que relaciona estas coordenadas con las

coordenadas del mundo, y abajo tenemos otra que relaciona las coordenadas del mundo con las coordenadas de la cámara. Por tanto, podemos obtener la siguiente expresión.

$$\begin{bmatrix} n\,x_f \\ n\,y_f \\ n \end{bmatrix} = \begin{bmatrix} f_x & 0 & C_x & 0 \\ 0 & f_y & C_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$

De esta expresión, tenemos que saber que, la posición desde la que captemos las imágenes que tendrán importancia para ser procesadas, siempre será la misma. Por tanto, la posición Z_c de cualquiera de las piezas que se sitúen en la mesa, siempre será la misma. Por lo consiguiente, Z_c es conocida. Con eso, podremos desarrollar lo siguiente.

$$n = Z_c$$

$$X_c = \frac{(x_f - C_x)}{f_x} Z_c$$

$$Y_c = \frac{(y_f - C_y)}{f_y} Z_c$$

Con estas estas expresiones, en el momento en que tengamos las coordenadas de una pieza en píxeles de la imagen, podremos hallar las coordenadas de esta misma pieza respecto a la cámara.

Pero primero, tenemos que hallar los parámetros intrínsecos de la cámara, f_x, f_y, C_x, C_y. Si nos fijamos en la imagen 78, al *topic* que contiene la información sobre los parámetros de la cámara le hemos llamado *camera_info*. Podemos acceder a la información de ese *topic*, para saber los parámetros intrínsecos de la cámara. Para ello, con la simulación de *Gazebo* ejecutándose, podremos lo siguiente por terminal.

```
$ rostopic echo /myur5/camera1/camera_info
```

Ejecutando ese comando, obtendremos la siguiente información por terminal.

```
D: [0.0, 0.0, 0.0, 0.0, 0.0]
K: [199.89382069250044, 0.0, 160.0, 0.0, 199.89382069250044, 120.5, 0.0, 0.0,
1.0]
R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
P: [199.89382069250044, 0.0, 160.0, -13.992567448475032, 0.0, 199.893820692500
44, 120.5, 0.0, 0.0, 0.0, 1.0, 0.0]
```

Imagen 135-Matriz parámetros intrínsecos

La matriz K es la matriz de parámetros intrínsecos, por tanto, con esta matriz sabemos todos los valores de esos parámetros.

$$f_x = 199.893821,\ f_y = 199.893821,\ C_x = 160,\ C_y = 120{,}5.$$

Una vez tenemos estos parámetros intrínsecos, ya podemos obtener las coordenadas de la pieza respecto de la cámara, con las expresiones que hemos obtenido anteriormente, las cuales las veremos a continuación.

$$n = Z_c$$

$$X_c = \frac{(x_f - C_x)}{f_x} Z_c$$

$$Y_c = \frac{(y_f - C_y)}{f_y} Z_c$$

Con esto, ya podemos encontrar las coordenadas de una pieza respecto al eje de coordenadas de la cámara, pero para comandar el robot necesitamos conocer las coordenadas de una pieza respecto al eje de coordenadas del mundo. Para realizar la transformación entre esos ejes de coordenadas, vamos a utilizar la siguiente expresión.

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

Esta expresión relaciona las coordenadas respecto al eje de la cámara con las coordenadas respecto al eje del mundo, utilizando una matriz homogénea. Por tanto, necesitamos hallar dicha matriz de transformación entre ejes.

La relación gráfica entre ambos ejes de coordenadas es la siguiente.

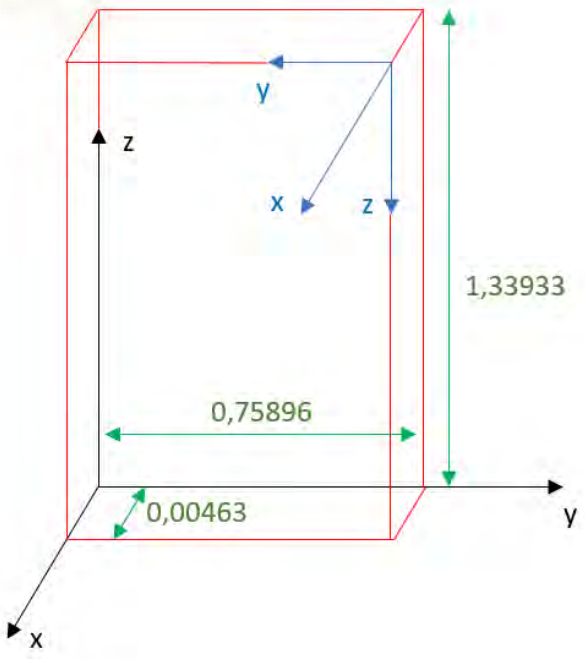


Imagen 136-Relación gráfica entre eje de la cámara y eje del mundo

En dicha representación gráfica, el eje negro representa el eje de coordenadas del mundo y el eje rojo representaría el eje de coordenadas de la cámara. Podemos sacar directamente el vector de translación del eje de la cámara respecto al eje del mundo.

$$t = (0.00463, 0.75896, 1.33933)$$

Con ese vector de translación y con la relación gráfica, podemos obtener gráficamente la matriz de transformación de coordenadas de la cámara a coordenadas del mundo, la cual veremos a continuación.

$$\begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0.00463 \\ 0 & -1 & 0 & 0.75896 \\ 0 & 0 & -1 & 1.33933 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$

De esta matriz homogénea podemos obtener las siguientes expresiones de transformación.

$$X_w = X_c + 0.00463$$

$$Y_w = 0.75896 - Y_c$$

$$Z_w = 1.33933 - Z_c = 0.804935$$

Con estas expresiones podemos encontrar las coordenadas respecto al eje del mundo desde unas coordenadas respectos a la cámara. Por tanto, ya hemos planteado todas las expresiones necesarias, que permiten transformar unas coordenadas de la imagen, en píxeles, a sus respectivas coordenadas respecto al eje del mundo.

Como decíamos anteriormente, este procedimiento lo vamos a encapsular también en una función, para poder realizar dicha transformación desde el lugar del código en el cual la necesitemos.

```
82  def convert_m2w(p_px):
83      Cx = 160
84      Cy = 120.5
85      Zc = 0.534395
86      fx = 199.8938206925
87      fy = 199.8938206925
88
89      x_c = ((p_px[0]-Cx)/fx)*Zc
90      y_c = ((p_px[1]-Cy)/fy)*Zc
91      x_w = x_c + 0.004627
92      y_w = 0.758958 - y_c
93
94      return x_w, y_w
```

Imagen 137-Función convert_m2w(p_px)

Esta función recibe como parámetro un vector, el cual contendrá las coordenadas *x* e *y* en píxeles, de una pieza de la escena. Dicha función nos devolverá esas mismas coordenadas, pero respecto al eje de coordenadas del mundo.

- **Desarrollo del *pick and place*, utilizando todas las funciones vistas hasta ahora y alguna sentencia más**

Ya hemos hablado de todas las funciones que se han definido en el programa, y que se han utilizado para mover el robot, en la aplicación de *pick and place*. Además, también hemos explicado el planteamiento de la función que convierte la posición de las piezas en coordenadas de la imagen a coordenadas del mundo. A partir de ahora se va a explicar el uso de esas funciones, combinado con más herramientas, para llevar a cabo la aplicación de *pick and place.*

Primero mostraremos el código de la función *main* del programa. Lo primero que haremos es crear un objeto de la clase *image_read*. Gracias a la creación de este objeto, lograremos que se ejecute el algoritmo de captación y procesamiento de la sección 5.2.1 y además también podremos acceder a las funciones de dicho algoritmo.

```
267     obj_img = image_read()  # Objeto de la clase image_read para acceder a sus funciones
268
```

Imagen 138-Creación objeto de la clase image_read

Una vez hecho esto, movemos el robot hasta la posición en la cual se procesarña la información de las imágenes, ya que desde esa posición es desde la cual se detectarán las posiciones de las piezas.

Una vez ubicado el robot en esa posición, podemos utilizar el objeto que hemos creado en la imagen 138, para acceder a la función *rev_coord*, y así obtener tanto las coordenadas de las piezas como el número de piezas existentes de cada color; esto se realiza en la línea 299 del programa que podremos ver a continuación.

```
287     # Move to home position
288     home_pos()
289
290     # Move to the position where pictures are processed
291     pos_take_images()
292
293     current_pose = geometry_msgs.msg.Pose()
294     current_pose = move_group_interface_arm.get_current_pose()
295
296     t.sleep(4)
297     coord = []
298
299     n_b, n_g, n_r, coord = obj_img.rev_coord()  # Recibe las coordenadas de las piezas en la imagen
300
301     print("************************")
302     print("El numero de piezas de cada color es, rojo:{}, azul:{}, verde:{}".format(n_r, n_b, n_g))
303     print("La cordenadas de los objetos son: {}".format(coord))
304     print("************************")
```

Imagen 139-Obtención de las coordenadas de las piezas

La variable *coord* es un vector de vectores, es un vector que contiene exactamente tres vectores. La posición cero del vector, corresponde al vector que contiene las coordenadas de las piezas azules, la posición uno corresponde al vector de las piezas verdes y la posición dos al de los azules. Las variables *n_r, n_b, n_g*, contienen el número de piezas rojas, azules y verdes respectivamente.

Una vez obtenidos esos datos, podemos pasar a mover el robot con el objetivo de realizar la acción de *pick and place* con las piezas. Para ello, desde el *main* ejecutaremos las siguientes funciones.

```
309     red_pieces(coord[2], n_r)
310     blue_pieces(coord[0], n_b)
311     green_pieces(coord[1], n_g)
```

Imagen 140-Funciones que realizan el pick and place

Cada una de estas funciones, se va a encargar de coger las piezas de un color concreto, y según el color de las piezas que esté recogiendo, las va a llevar su correspondiente vitrina de las tres que hay detrás del robot. Por ejemplo, la función *red_pieces*, se encargará de manipular todas las piezas rojas hasta que no quede ninguna en las mesas y las colocará en orden, en su correspondiente vitrina.

A cada una de las funciones, se les pasa el vector de coordenadas que le corresponde según el color, y el número de piezas que existen de ese color en concreto. Por ejemplo, a la función *blue_pieces*, se le pasará como parámetros *coord[0],* que contiene las posiciones de todas las piezas azules de la mesa y *n_b*, que contiene el número exacto de piezas azules de la mesa.

A continuación se va a mostrar la función *red_pieces(c, n).*

```
96  def red_pieces(c, n):
97      global current_pose, target_pose
98      global r
99      incr = 0
100     for i in range(n):
101         coord_red = []
102         coord_red.append(c[r])
103         coord_red.append(c[r+1])
104         x_r, y_r = convert_m2w(coord_red)
105
106         # Approx red piece
107         approximation(x_r, y_r)
108
109         # Open the gripper
110         open_gripper()
111
112         # Take red piece
113         take_piece()
114
115         # Close the gripper
116         close_gripper()
117
118         # Return to approximation pos
119         approximation(x_r, y_r)
120
121         # Move to home position
122         home_pos()
123
124         # Move the TCP over the red position
125         approx_goals = [-0.90239528235945 88, -0.7322973147618699, 1.2709732199546568, -2.115323358711743, -1.5688081113077317, -0.9020939621886495]
126         move_group_interface_arm.go(approx_goals, wait=True)
127         move_group_interface_arm.stop()
128
```

Imagen 141-Función red_pieces() parte 1

```
129         # Leave the piece
130         current_pose_2 = move_group_interface_arm.get_current_pose()
131         target_pose.position = current_pose_2.pose.position
132         target_pose.orientation = current_pose_2.pose.orientation
133         target_pose.position.x = 0.549634458556 - incr
134         move_group_interface_arm.set_pose_target(target_pose)
135         move_group_interface_arm.go(wait=True)
136         target_pose.position.z = 0.967
137         move_group_interface_arm.set_pose_target(target_pose)
138         move_group_interface_arm.go(wait=True)
139         move_group_interface_arm.stop()
140
141         open_gripper()
142
143         # Return over the red position
144         target_pose.position.z = 1.1
145         move_group_interface_arm.set_pose_target(target_pose)
146         move_group_interface_arm.go(wait=True)
147         move_group_interface_arm.stop()
148
149         pos_take_images()
150
151         r = r + 2
152         incr = incr + 0.1
```

Imagen 142-Función red_pieces() parte 2

Con el bucle *for* de la línea 100 y su condición, queda claro que se cogerán todas las piezas rojas que estén sobre la mesa. En la línea 104 se utiliza la función *convert_m2w*, la cual podemos ver en la imagen 137; con el uso de esta función conseguimos obtener las coordenadas de alguna pieza respecto al eje del mundo, para mover el robot hasta dichas coordenadas.

A continuación, desde la línea 107 hasta la 122, se utilizan las funciones que ya hemos visto, para comandar el robot hasta las coordenadas donde se ubique una de las piezas. Seguidamente desde la línea 130 a la 141, se puede ver el código que mueve el robot hasta la posición donde se deben dejar las piezas rojas y luego abre la pinza para dejarla.

Por último, podemos destacar la variable llamada *incr*, que sirve para dar un incremento en el eje *x*, cuando se dejen varias piezas sobre el mismo soporte, para que el robot no las deje unas encima de las otras.

Ahora vamos a ver una serie de imágenes para entender se mueve el robot, según se ejecutan las sentencias de la función *red_pieces.*

Desde la línea 107 a la línea 119, de la imagen 141, se mueve el robot para aproximarse a la pieza que se va a coger, para luego cogerla con las pinzas.

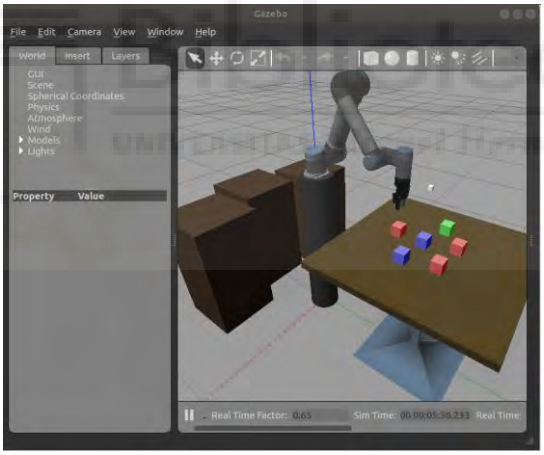


Imagen 143-Aproximación del robot a una pieza roja

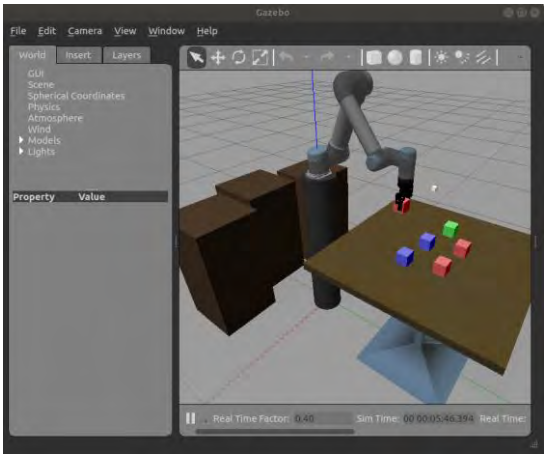


Imagen 144-Robot cogiendo pieza roja

Seguidamente, la línea 122 de la imagen 141, se lleva al robot, con la pieza sujeta entre las pinzas, a la posición *Home.*

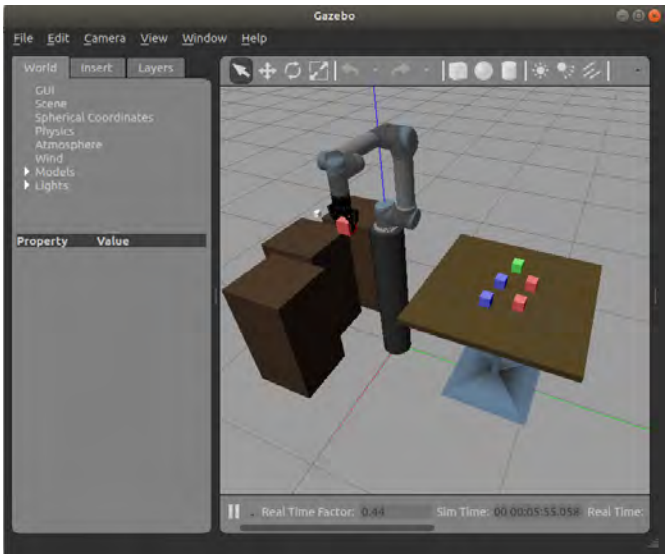


Imagen 145-Robot en posición Home con la pieza

Desde la línea 125 hasta la 147, de las imágenes 141 y 142, se mueve el robot hasta el soporte donde se deben dejar las piezas rojas, y la deja en dicho soporte.

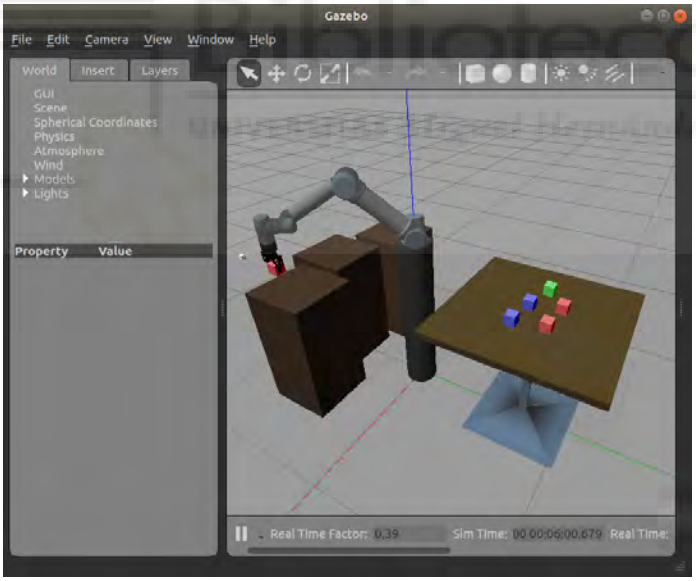


Imagen 146-Robot sobre soporte donde dejar la pieza roja

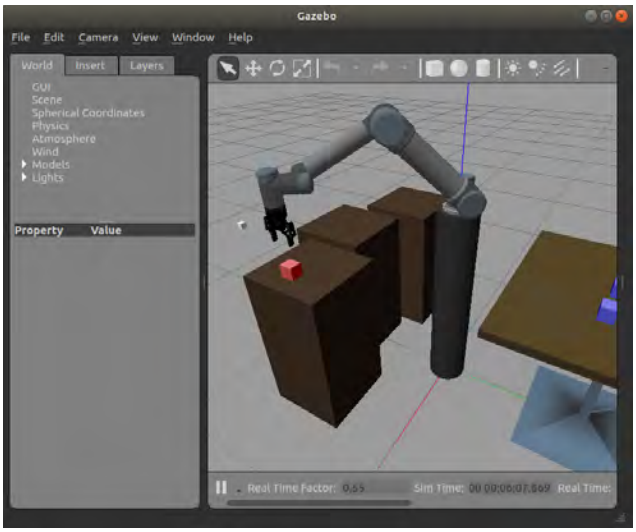


Imagen 147-Robot dejando pieza roja en su lugar

Por último, ejecutando la función de la línea 149, de la imagen 142, el robot se mueve hasta la posición de tomar imágenes preparándose para empezar el proceso de coger otra pieza.

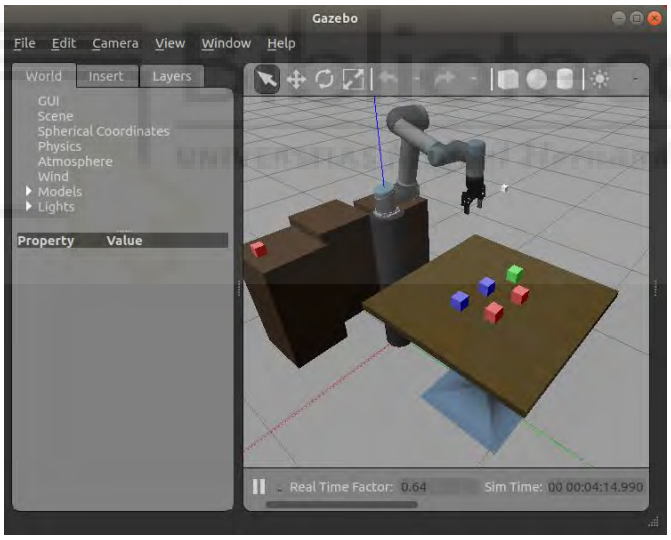


Imagen 148-Robot esperando a coger otra pieza

Las funciones *blue_pieces* y *green_pieces*, realizan el mismo proceso, pero sobre las piezas azules y verdes correspondientemente.

5.2.3.2. Mejora de la aplicación de pick and place

Al simular y probar el programa que se ha expuesto en el apartado 5.2.3.1, surgió una problemática. Este inconveniente suponía que, sí una pieza era movida mientras el programa se estaba ejecutando, ergo el robot estaba en movimiento, el programa no era capaz de reconocer ese movimiento y por tanto el robot nunca llegaba a coger esa pieza, ya que no reconocía el cambio de posición.

Este problema es causado porque durante la ejecución del programa, solo se obtienen las coordenadas de las piezas una única vez; solamente se obtiene las coordenadas al principio de la ejecución, como podemos ver en la imagen 139. Debido a esto, nunca se actualizan las coordenadas donde se ubican las piezas, lo que causa que, si se mueve una pieza de posición, se quita alguna pieza o se añade otra pieza, el robot realizará un mal *pick and place.*

Vamos a ver un ejemplo de ese error a continuación.

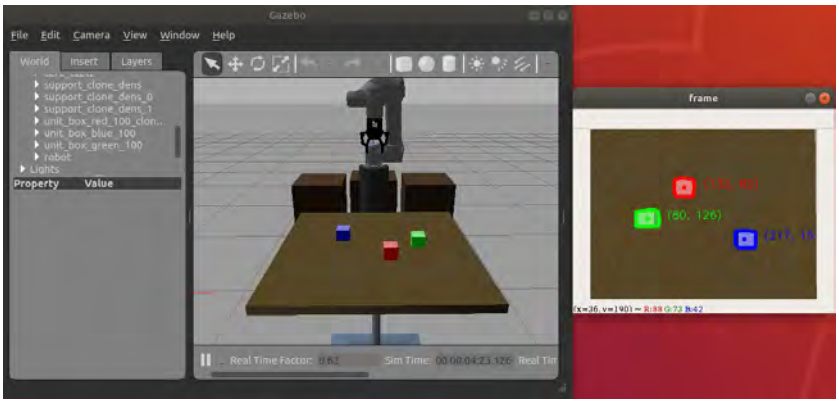


Imagen 149-Robot en posición de tomar imágenes

En esta imagen 149, podemos ver al robot que se ha colocado en la posición desde la cual va a tomar las coordenadas en las que se encuentran las piezas. Como podemos visualizar, en la mesa hay una pieza de cada color.

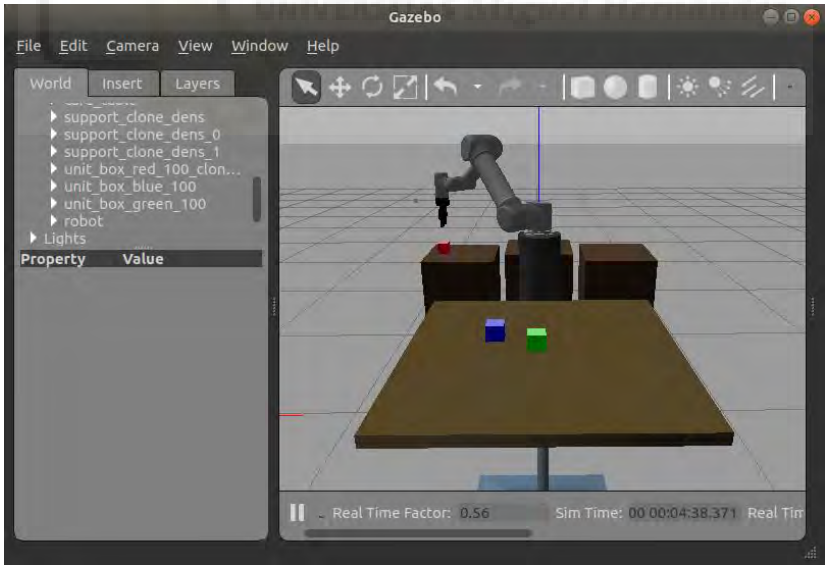


Imagen 150-Robot dejando pieza roja y pieza verde movida

En la imagen 150 podemos apreciar que mientras el robot manipulaba la pieza roja, la se ha modificado la posición de la pieza verde.

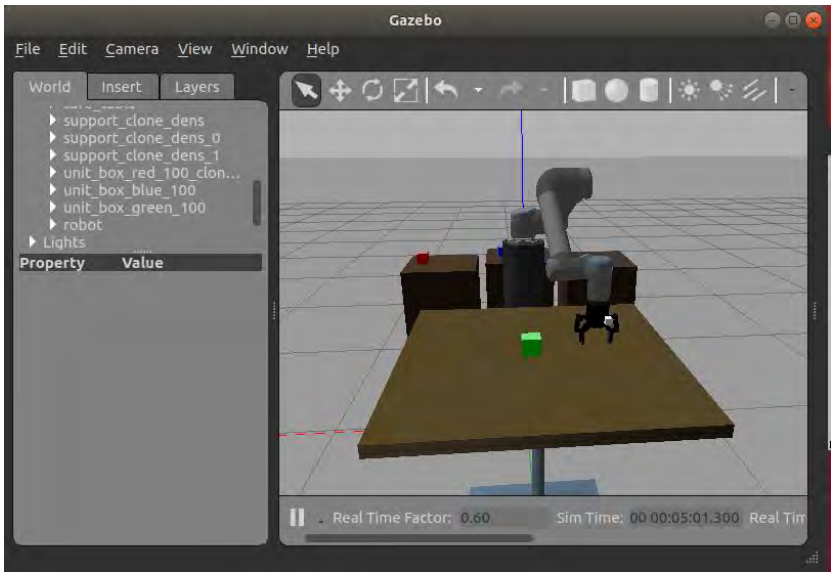


Imagen 151-Robot errando en el pick and place

En esta última imagen nos podemos dar cuenta de que, al cambiar la posición de la pieza verde, el robot falla al ir a recoger dichas piezas, debido a que el robot va se dirige a las coordenadas en las que estaba la pieza verde en la imagen 149, en vez de detectar el cambio de posición y actualizar las coordenadas.

Me propuse arreglar este error y conseguir que, si durante la ejecución del programa de *pick and place* momento se moviera una pieza durante la ejecución del *pick and place*, que el programa fuera capaz de reconocer ese movimiento y no equivocarse a la hora de coger la pieza movida. También se solucionaría el caso en que se eliminara una pieza de la mesa o el caso de que se añadiera otra pieza.

Ciertamente, hubo que hacer pocos cambios en el programa original de la aplicación de *pick and place.* Todos los cambios, se centraron en la ejecución de las funciones *red_pieces, blue_pieces* y *green_pieces*.

El primer cambio lo realizaremos en el *main* del programa. Concretamente, del *main* se modificó la parte contenida en la imagen 140. En dicha parte, debíamos de realizar las modificaciones que veremos a continuación.

```
294     nt = n_r+n_b+n_g
295     while(nt>0):
296         if(n_r>0):
297             red_pieces(coord[2])
298             incr_r = incr_r + 0.1
299         elif(n_b>0):
300             blue_pieces(coord[0])
301             incr_b = incr_b + 0.1
302         elif(n_g>0):
303             green_pieces(coord[1])
304             incr_g = incr_g + 0.1
305         t.sleep(2)
306         n_b, n_g, n_r, coord = obj_img.rev_coord()
307         nt = n_r+n_b+n_g
```

Imagen 152-Ejecución funciones pick and place mejorado

Como podemos ver, en este caso existe una variable nt, que contiene el valor total de piezas que quedan por recoger. Esta variable sirve de restricción en el bucle *while*, para que, mientras queden piezas en la mesa, se ejecuten las funciones de recogida de piezas.

Dentro de ese bucle, primero se analiza si quedan piezas rojas. En caso de quedar piezas rojas, se ejecutará la función *red_pieces*, y por tanto se recogerá una de las piezas rojas y se llevará a la posición de dejada, de forma idéntica a como se hacía en el programa original.

En caso de que no queden piezas rojas, se comprobará si quedan piezas azules, y en caso de que existan aún piezas, estas serán movidas hasta su respectiva vitrina. En caso de no quedar piezas azules, se pasará a analizar las piezas verdes de la misma forma que con los otros colores.

Como vemos en la línea 306, de la imagen 152, al final del bucle se actualizan las coordenadas y el número de piezas. Esto hace que cada vez que se mueva una pieza, antes de mover otra, se actualicen las coordenadas.

En este caso, las funciones *red_pieces, blue_pieces* y *green_pieces*, solo reciben como parámetro el vector de coordenadas. Esto es debido a que, cada vez que se ejecute una de las tres funciones, solamente se recogerá una pieza y seguidamente se actualizarán las coordenadas de dichas. Por tanto, habrá que ejecutar las funciones tantas veces como piezas se vayan a recoger. Esto lo podremos entender a continuación.

```
92  def red_pieces(c):
93      global current_pose, target_pose
94      global incr_r
95      coord_red = []
96      coord_red.append(c[0])
97      coord_red.append(c[1])
98      x_r, y_r = convert_m2w(coord_red)
99
100     # Approx red piece
101     approximation(x_r, y_r)
102
103     # Open the gripper
104     open_gripper()
105
106     # Take red piece
107     take_piece()
108
109     # Close the gripper
110     close_gripper()
111
112     # Return to approximation pos
113     approximation(x_r, y_r)
114
115     # Move to home position
116     home_pos()
117
118     # Move the TCP over the red position
119     approx_goals = [-0.9023952823594588, -0.7322973147618699, 1.2709732199546568, -2.115323358711743, -1.5688081113077317, -0.9020939621886495]
120     move_group_interface_arm.go(approx_goals, wait=True)
121     move_group_interface_arm.stop()
```

Imagen 153-Función red_pieces() aplicación mejorada parte 1

```
122
123     # Leave the piece
124     current_pose_2 = move_group_interface_arm.get_current_pose()
125     target_pose.position = current_pose_2.pose.position
126     target_pose.orientation = current_pose_2.pose.orientation
127     target_pose.position.x = 0.549634458556 - incr_r
128     move_group_interface_arm.set_pose_target(target_pose)
129     move_group_interface_arm.go(wait=True)
130     target_pose.position.z = 0.967
131     move_group_interface_arm.set_pose_target(target_pose)
132     move_group_interface_arm.go(wait=True)
133     move_group_interface_arm.stop()
134
135     open_gripper()
136
137     # Return over the red position
138     target_pose.position.z = 1.1
139     move_group_interface_arm.set_pose_target(target_pose)
140     move_group_interface_arm.go(wait=True)
141     move_group_interface_arm.stop()
142
143     pos_take_images()
```

Imagen 154- Función red_pieces() aplicación mejorada parte 2

En comparación con las imágenes 141 y 142, vemos que en este caso se ha eliminado el bucle *for*. Quitando ese bucle conseguimos que, al ejecutarse la función sólo se mueva una pieza. No se moverá otra pieza roja hasta que no se actualicen de nuevo las coordenadas de las piezas y se vuelva a ejecutar la función *red_pieces()*.

A continuación, se va a mostrar un ejemplo de ejecución de la aplicación de *pick and place* mejorada, en el que se va a ver con facilidad, la mejora que se ha implementado respecto a la aplicación original.

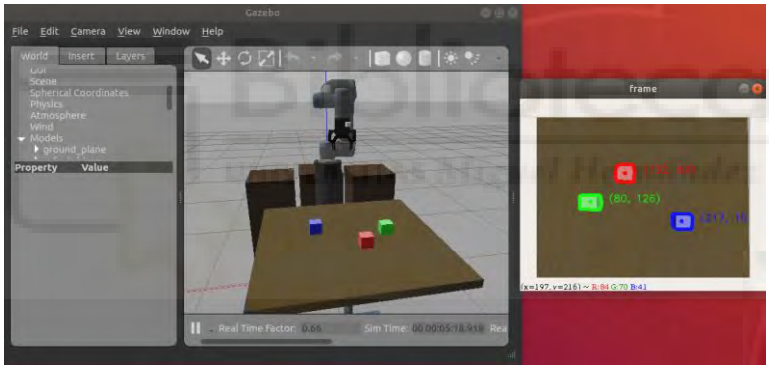


Imagen 155-Robot en posición de tomar imágenes

En esta primera imagen podemos ver, como el robot se posiciona en la posición en la que detecta las coordenadas de las tres piezas que hay en la mesa.

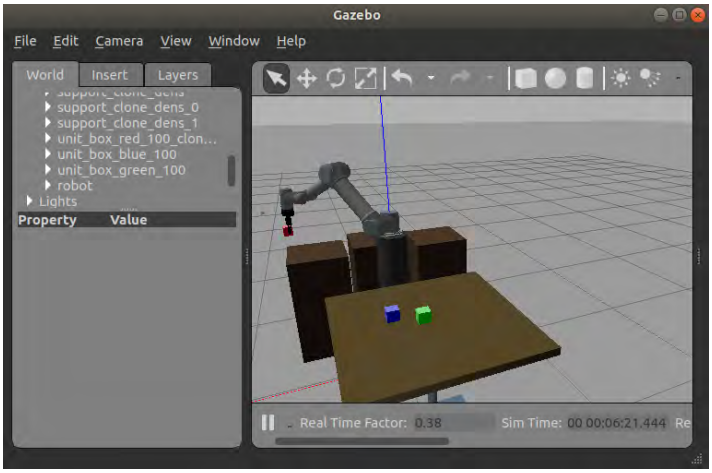


Imagen 156-Cambio posición pieza verde

En la imagen 156, se puede observar que la pieza verde ha sido movida de posición respecto a la posición en la que se ubicaba en la imagen 155.

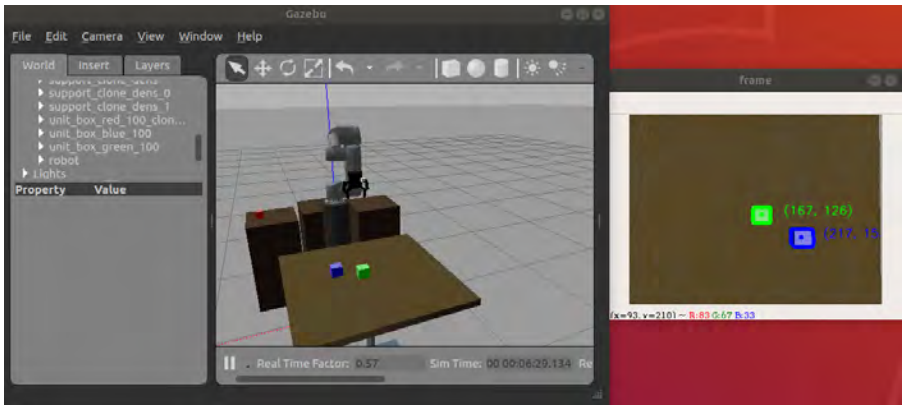


Imagen 157- Robot en posición de tomar imágenes

En el caso de la aplicación mejorada, como podemos ver en la imagen 157, después de dejar la pieza roja, el robot se mueve hasta la posición donde se detectan las piezas y se actualizan las coordenadas de dichas piezas.

Desde este momento, el robot ya ha detectado mediante la cámara, que las coordenadas de la pieza verde han cambiado.

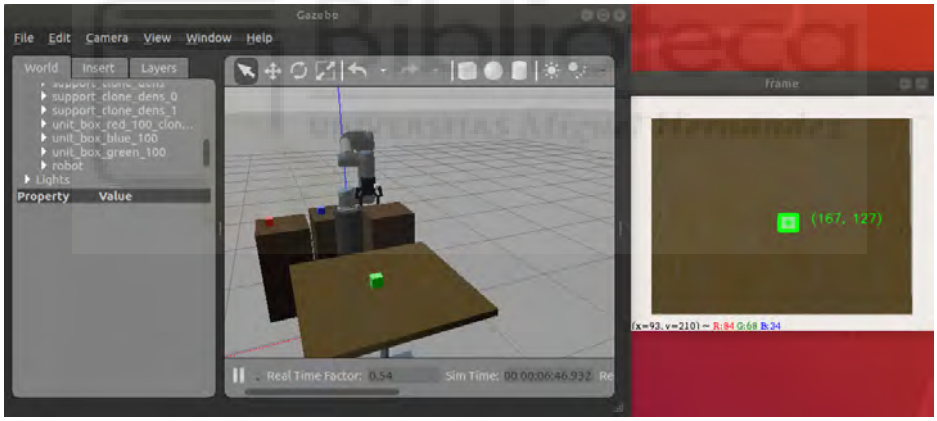


Imagen 158- Robot en posición de tomar imágenes

Atendiendo a la imagen 158, vemos que el robot después de dejar la pieza azul también se actualiza las coordenadas, por si la pieza verde ha sido desplazada de nuevo.

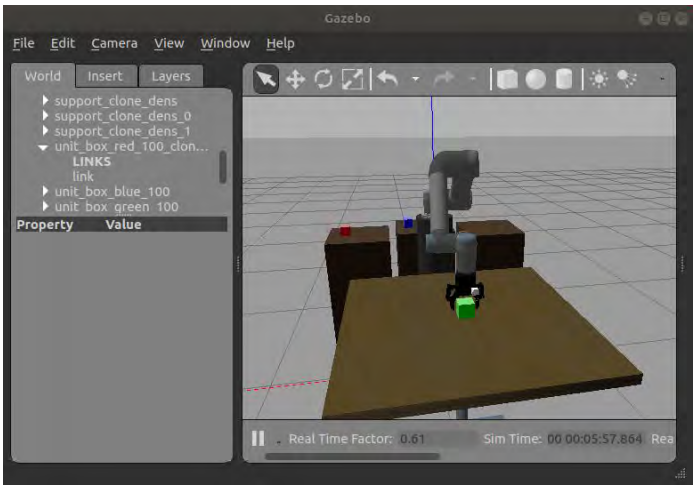


Imagen 159-Robot cogiendo pieza verde

En esta imagen se puede ver perfecto que a diferencia de la imagen 151, con la mejora implementada en el programa, el robot es capaz de coger las piezas que han sido desplazadas durante la ejecución del mismo.

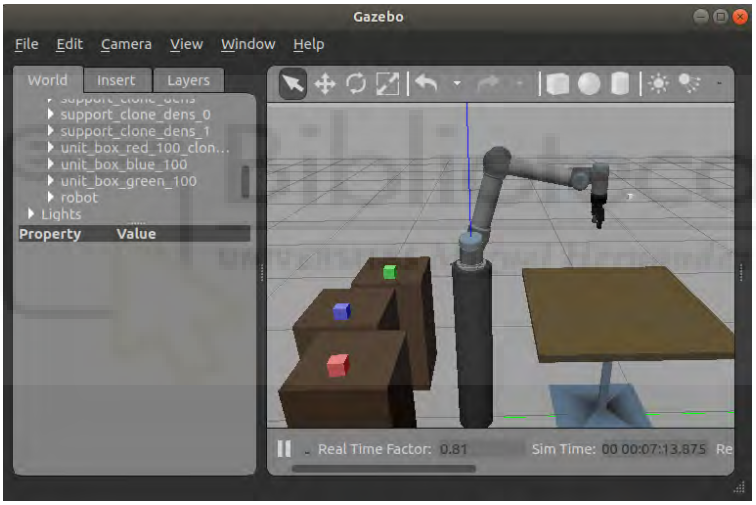


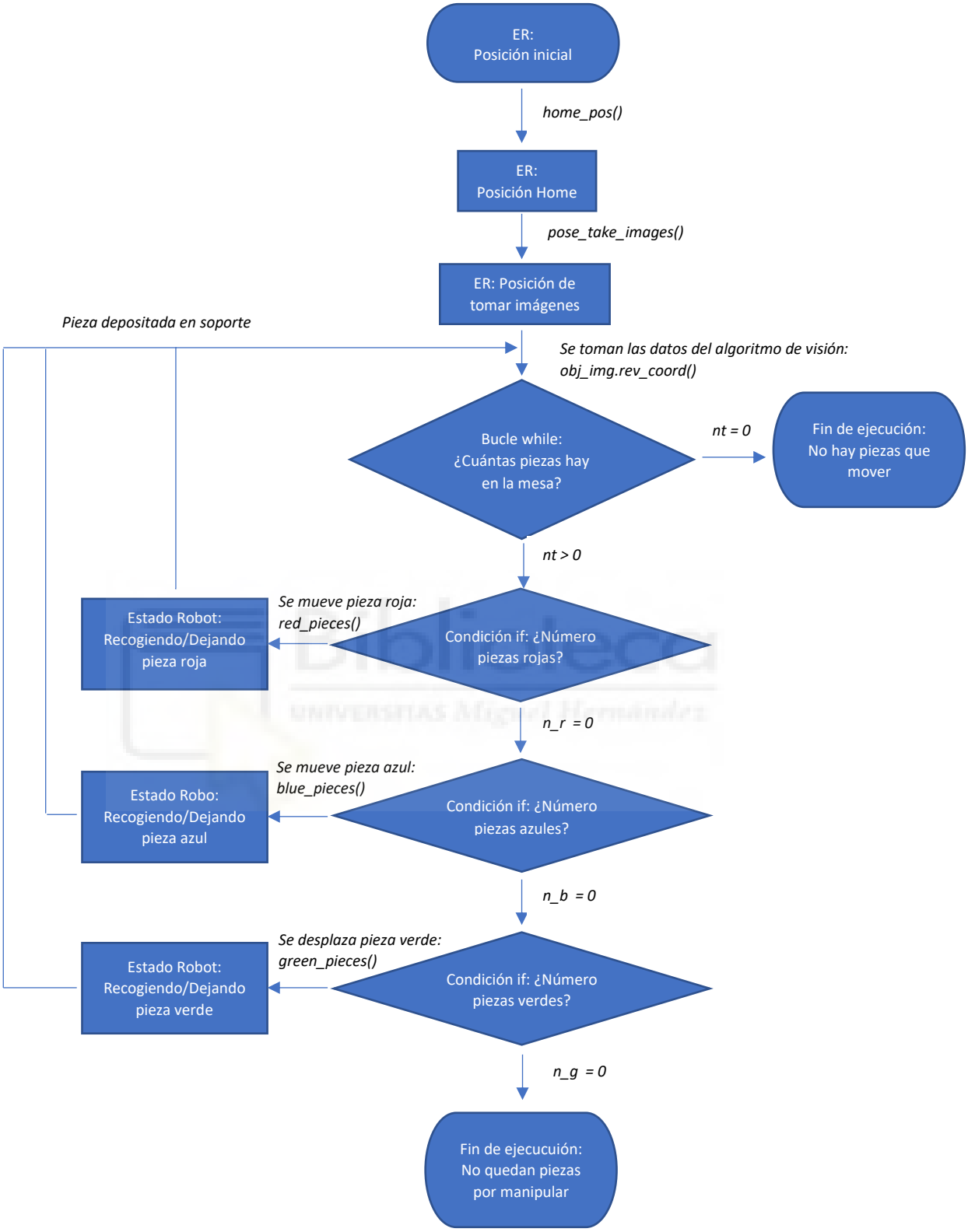
Imagen 160-Piezas en las vitrinas

Por último, en esta imagen podemos ver que, en la ejecución del programa mejorados, el robot deposita correctamente todas las piezas en sus respectivas vitrinas.

Si comparáramos este ejemplo de ejecución de la aplicación mejorada con el ejemplo de la aplicación de las imágenes 149, 150 y 151, nos daremos cuenta fácilmente de la mejora que se ha implementado en la aplicación de *pick and place*.

Con el objetivo de que se comprenda fácilmente, la lógica de programación que sigue la aplicación de *pick and place* final, la cual incluye también las mejoras que se han expuesto en el apartado 5.2.3.2, se va a mostrar un diagrama de flujo que recoge toda la lógica de dicha aplicación.

En el diagrama de flujo se va a utilizar la abreviación ER, que significará Estado del robot.


ER:
Posición inicial
home_pos()
ER:
Posición Home
pose_take_images()
ER: Posición de
tomar imágenes
Pieza depositada en soporte
Se toman las datos del algoritmo de visión:
obj_img.rev_coord()
Bucle while:
¿Cuántas piezas hay
en la mesa?
nt = 0
Fin de ejecución:
No hay piezas que
mover
nt > 0
Se mueve pieza roja:
red_pieces()
Condición if: ¿Número
piezas rojas?
Estado Robot:
Recogiendo/Dejando
pieza roja
n_r = 0
Se mueve pieza azul:
blue_pieces()
Condición if: ¿Número
piezas azules?
Estado Robo:
Recogiendo/Dejando
pieza azul
n_b = 0
Se desplaza pieza verde:
green_pieces()
Condición if: ¿Número
piezas verdes?
Estado Robot:
Recogiendo/Dejando
pieza verde
n_g = 0
Fin de ejecucuión:
No quedan piezas
por manipular

6. Simulaciones realizadas y resultados obtenidos

En el apartado 6 de esta memoria se van a mostrar los resultados, de simular las aplicaciones que se han desarrollado durante la realización del proyecto. Se van a mostrar simulaciones relacionadas con la aplicación de control manual del robot que se ha expuesto en el apartado 5.2.2 y la aplicación de *pick and place* que se ha mostrado en el apartado 5.2.3. Todas las simulaciones que veremos a continuación, se han realizado en el entorno de simulación de *Gazebo.*

6.1. Simulación aplicación de control manual

En este apartado vamos a mostrar los resultados de simulación obtenidos de la aplicación de control manual del robot.

- **Lanzamiento simulación**

Antes de empezar a lanzar las simulaciones de *Gazebo*, hemos de tener en cuenta que para poder trabajar con los paquetes de *ROS* que se han desarrollado dentro de nuestro entorno de trabajo, tenemos que realizar previamente una tarea. Teniendo en cuenta que nuestro entorno de trabajo se llama *new_catkin_*ws, podemos realizar la tarea de dos formas; la primera es ejecutar en todos los terminales en los que se va a trabajar, el comando que vamos a ver a continuación.

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

La segunda opción es colocar ese mismo comando, en el archivo *.bashrc* del sistema operativo. Al realizar alguna de estas dos opciones ya podremos acceder a los archivos y paquetes de nuestro entorno de trabajo.

Antes de mostrar el movimiento del robot al ejecutar esas aplicaciones, debemos ver específicamente primero, como inicializar el entorno de *ROS,* y luego como lanzar la simulación de *Gazebo* y como ejecutar el código desarrollado en *Python*.

Para inicializar el ecosistema de *ROS*, ejecutaremos por terminal el comando que veremos a continuación.

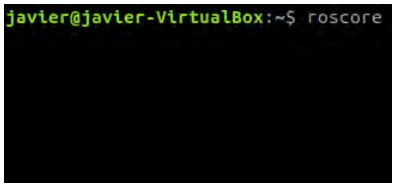


Imagen 161-Comando roscore

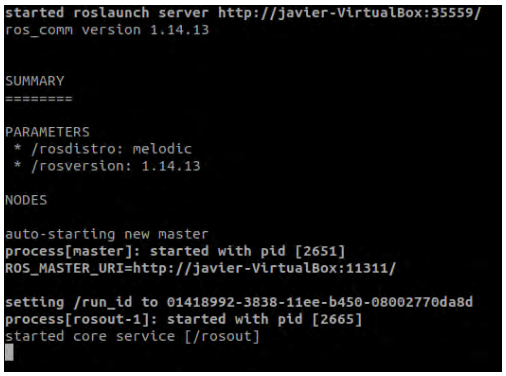


Imagen 162-roscore ejecutandose

La ejecución del comando *roscore* por el terminal, inicializa el *ROSMaster*, lo cual permite que los diferentes nodos de *ROS* puedan ser visibles y además nos habilita la comunicación con dichos nodos, utilizando mensajes. Además, también inicializa el *ROS Parameter Server* y el nodo *Rosout*.

Una vez tenemos inicializado el ecosistema de *ROS*, ya podemos lanzar las simulaciones desarrolladas en nuestro entorno de trabajo.

Podemos lanzar el modelo del robot que hemos desarrollado para *Gazebo*, desarrollo el cual podemos ver en el apartado 5.1 de la memoria, de la siguiente manera.

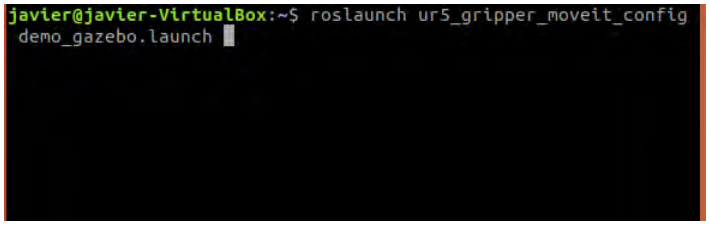


Imagen 163-Lanzamiento simulación Gazebo

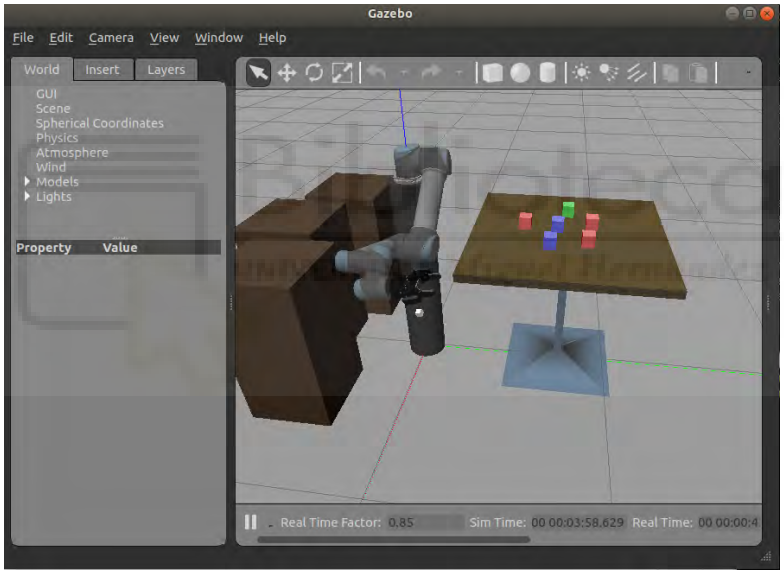


Imagen 164-Simulación del robot en Gazebo

El comando que aparece en la imagen 163 nos permite lanzar la simulación que vemos en la imagen 164.

El comando *roslaunch*, nos permite lanzar entornos de simulación desarrollados para *ROS*. a nosotros nos ha permitido lanzar los entornos de simulación del robot, en *Gazebo*. La segunda instancia del comando, *ur5_gripper_moveit_config*, es el nombre del paquete de *ROS* donde se ubican todos los archivos relacionados con la simulación que vemos en la imagen 164. Contiene por ejemplo el modelo URDF del robot y de la pinza, los archivos que permiten simular las físicas del robot etc.

Por último, *demo_gazebo.launch*, es el archivo que contiene la información necesaria, para lanzar los modelos URDF en *Gazebo* y para que se inicialice el *software*

MoveIt, de forma que se habilite la planificación de movimiento en el robot en la simulación.

Una vez tenemos lanzada la simulación, ya podemos pasar a ejecutar los programas realizados en el apartado 5.2. A continuación se va a mostrar como ejecutar el código de *Python*, desarrollado en el apartado 5.2.2, el cual se refiere a la programación de la aplicación manual. Para ejecutar dicha aplicación vamos a ejecutar el siguiente comando.

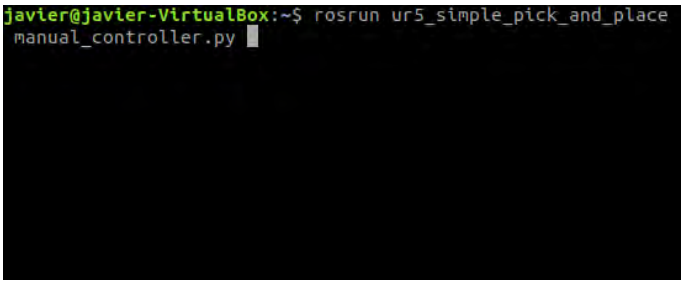


Imagen 165-Ejecuión aplicación manual

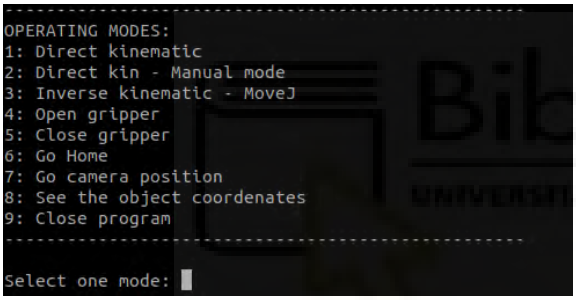


Imagen 166-Interfaz modo manual

Imagen 167-Imagen captada por la cámara

Como podemos observar, de la ejecución del comando de la imagen 165 por terminar, en ese mismo terminal surge la interfaz para utilizar la aplicación de control manual del robot, la cual podemos ver en la imagen 166. En la imagen 167 podemos ver la ventana que contiene la información que capta la cámara.

A continuación, vamos a observar las ventanas que tenemos en ejecución en el ordenador en este momento.

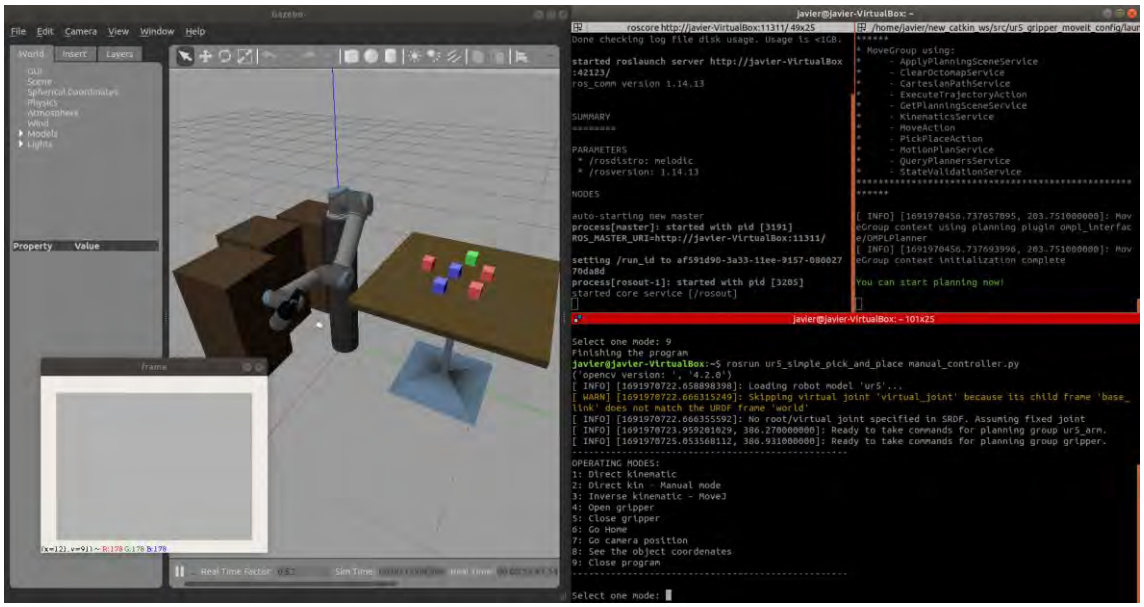


Imagen 168-Apariencia monitor ordenador

A la derecha de la imagen se encuentran los tres terminales que hemos utilizado para lanzar y ejecutar todos los archivos necesarios de la aplicación manual. A la izquierda, podemos ver la ventana de *Gazebo* en la que está la simulación del robot y también una ventana llamada *frame*, que contiene la imagen captada por la cámara y procesada.

Desde el terminal de abajo, accedemos a los diferentes modos de operación de la aplicación de control manual. Vamos a visualizar los efectos de cada modo de operación, sobre la simulación de *Gazebo.*

- **Modos 6 y 7**

Si seleccionamos los modos 6 o 7, llevaremos al robot a unas posiciones predefinidas.

```
-------------------------------------------------
OPERATING MODES:
1: Direct kinematic
2: Direct kin - Manual mode
3: Inverse kinematic - MoveJ
4: Open gripper
5: Close gripper
6: Go Home
7: Go camera position
8: See the object coordenates
9: Close program
-------------------------------------------------

Select one mode: 6
MOVING TO HOME POSITION
```

Imagen 169-Selección modo 6

Imagen 170-Simulación Gazebo

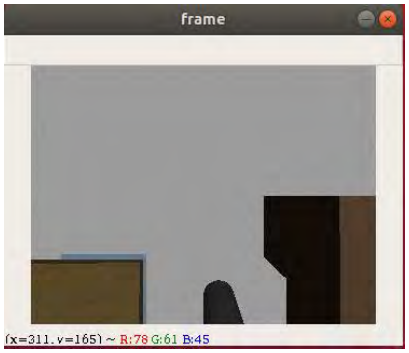


Imagen 171-Imagen captada por la cámara

```
------------------------------------------------
OPERATING MODES:
1: Direct kinematic
2: Direct kin - Manual mode
3: Inverse kinematic - MoveJ
4: Open gripper
5: Close gripper
6: Go Home
7: Go camera position
8: See the object coordenates
9: Close program
------------------------------------------------

Select one mode: 7
MOVING TO THE POSITION WHERE WE PROCESS PICTURES
```

Imagen 172-Selección modo 7

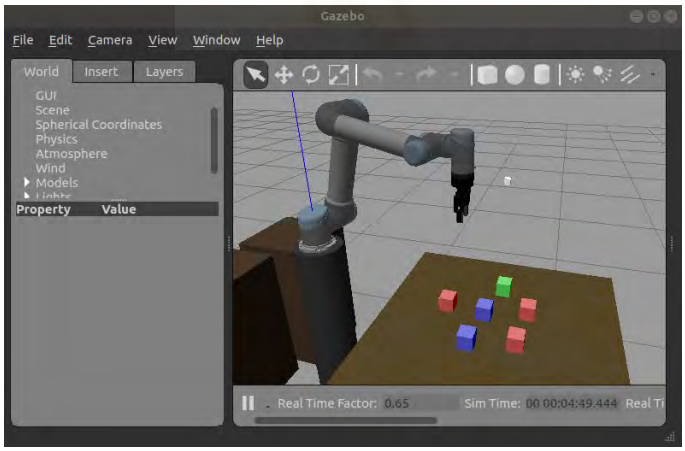


Imagen 173-Simulación Gazebo

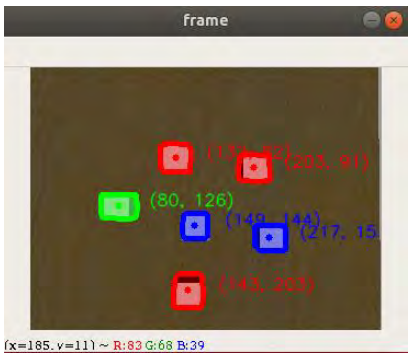


Imagen 174-Imagen captada por la cámara

Cuando se selecciona el modo 7, podemos ver en las imágenes 173 y 174 que el robot se encuentra en la posición en la cual se pueden detectar las coordenadas de las piezas. Si seleccionamos el modo 8, el programa nos dirá las coordenadas de dichas piezas.

```
------------------------------------------------
OPERATING MODES:
1: Direct kinematic
2: Direct kin - Manual mode
3: Inverse kinematic - MoveJ
4: Open gripper
5: Close gripper
6: Go Home
7: Go camera position
8: See the object coordenates
9: Close program
------------------------------------------------

Select one mode: 8
Coordenadas piezas azules: ([217, 155, 149, 145])
Coordendas piezas verdes: ([80, 126])
Coordenadas piezas rojas: ([143, 203, 203, 91, 132, 82])
```

Imagen 175-Selección modo 8

- **Modos 4 y 5**

La ejecución de los modos 4 y 5, sirven para abrir y cerrar la pinza respectivamente.

```
------------------------------------------------
OPERATING MODES:
1: Direct kinematic
2: Direct kin - Manual mode
3: Inverse kinematic - MoveJ
4: Open gripper
5: Close gripper
6: Go Home
7: Go camera position
8: See the object coordenates
9: Close program
------------------------------------------------

Select one mode: 4
OPENING THE GRIPPER
```

Imagen 176-Selección modo 4

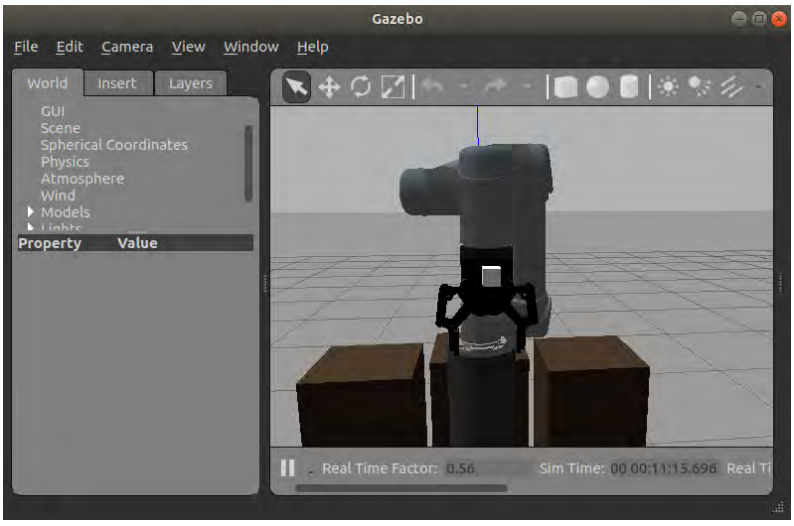


Imagen 177-Simulación Gazebo

```
------------------------------------------------
OPERATING MODES:
1: Direct kinematic
2: Direct kin - Manual mode
3: Inverse kinematic - MoveJ
4: Open gripper
5: Close gripper
6: Go Home
7: Go camera position
8: See the object coordenates
9: Close program
------------------------------------------------

Select one mode: 5
CLOSING THE GRIPPER
```

Imagen 178-Selección modo 5

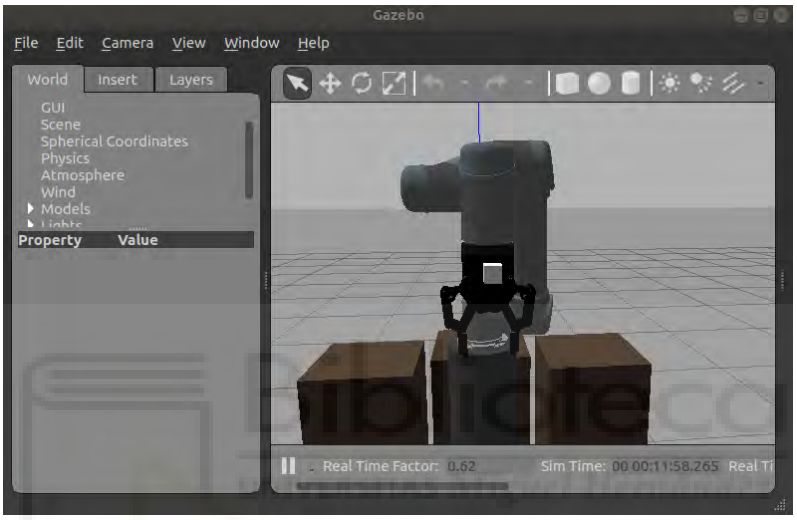


Imagen 179-Simulación Gazebo

- **Modo 1**

El modo de operación 1, permite mover el robot utilizando cinemática directa. Al seleccionar este modo de operación, el programa solicitará al usuario que introduzca los valores de las posiciones angulares que tiene que alcanzar cada articulación.

```
Select one mode: 1

DIRECT KINEMATIC
Current joint values:
[1.380757826479102, -1.2882345249857359, 1.457043449245301, -1.8217120295381841, -1.5717244462215305,
 -0.22258943156177668]

Enter new values:

Enter joint 0 value: 0
Enter joint 1 value: -pi/2
Enter joint 2 value: -0.4
Enter joint 3 value: -pi/4
Enter joint 4 value: 0
Enter joint 5 value: 0.4
```

Imagen 180-Selección modo 1

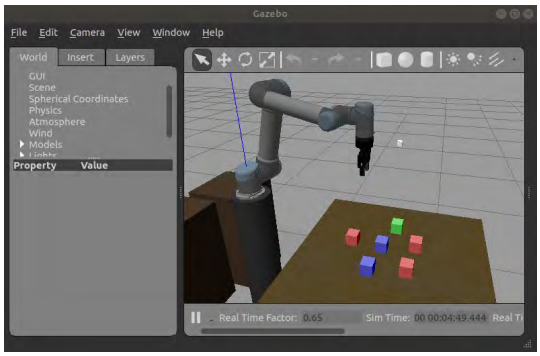


Imagen 181-Posición origen robot

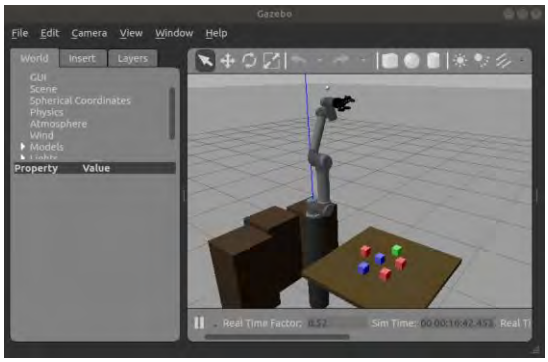


Imagen 182-Posición al ejecutar modo 1

- **Modo 3**

Con el modo de operación 3, podemos comandar el robot utilizando la cinemática inversa. En este caso, al seleccionar el modo 3, el programa pedirá al usuario la posición, (x, y, z), a la que se tiene que llevar el extremo del robot. Seguidamente también se consultará al usuario si se quiere modificar la orientación del extremo del robot. En caso de que no se quiera modificar, se conservará la orientación previa. En el caso de que el usuario quiera modificar la orientación del efector final, se deberá introducir el cuaternión que defina dicha orientación deseada.

En el siguiente ejemplo se va a mover el extremo del robot a la posición (0.1, 0.55, 1.3) y no se va a modificar la orientación de dicho extremo.

```
Select one mode: 3

INVERSE KINEMATIC
Current values of end effector pose:
header:
  seq: 0
  stamp:
    secs: 497
    nsecs: 661000000
  frame_id: "world"
pose:
  position:
    x: 0.498258964602
    y: 0.108260711555
    z: 1.43199392471
  orientation:
    x: -0.00323751955783
    y: 0.704278786059
    z: -0.00304874917361
    w: 0.709909582344

Enter new end effector values:

Enter position x: 0.1
Enter position y: 0.55
Enter position z: 1.3
Do you want to modify the orientation using a quaternion?(Yes(1) or No(2)): 2
```

Imagen 183-Selección modo 3

Imagen 184-Posición origen robot

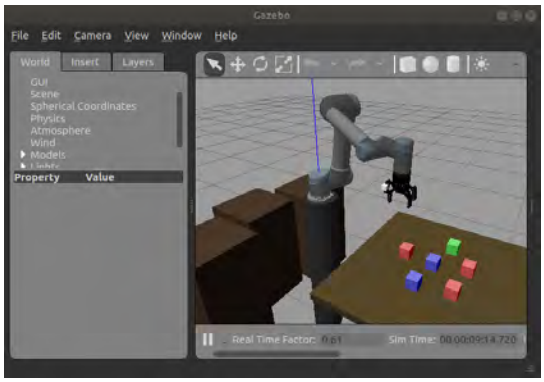


Imagen 185-Posición al ejecutar modo 3

Como se puede observar en la imagen 183, el programa pide las coordenadas (x, y, z) a las que se quiere llevar el robot. Luego, pregunta si se quiere modificar la orientación del extremo, que en este caso se ha introducido un 2 para indicar, que no se quiere modificar dicha orientación. Podemos darnos cuenta comparando las imágenes 184 y 185, que la orientación de la pinza es idéntica, ya que se le ha indicado al programa que no se modifique dicha orientación.

A continuación, se va a mostrar otro ejemplo del modo 3 en el que sí que se va a modificar la orientación.

```
Select one mode: 3

INVERSE KINEMATIC
Current values of end effector pose:
header:
  seq: 0
  stamp:
    secs: 950
    nsecs: 575000000
  frame_id: "world"
pose:
  position:
    x: 0.497943026448
    y: 0.108624742551
    z: 1.43114806102
  orientation:
    x: -0.00285734772539
    y: 0.707369534312
    z: -0.00316469034564
    w: 0.706831070502

Enter new end effector values:

Enter position x: 0
Enter position y: -0.4
Enter position z: 1.3
Do you want to modify the orientation using a quaternion?(Yes(1) or No(2)): 1
Enter value w: 0
Enter value x: 0
Enter value y: 1
Enter value z: 0
```

Imagen 186-Selección modo 3

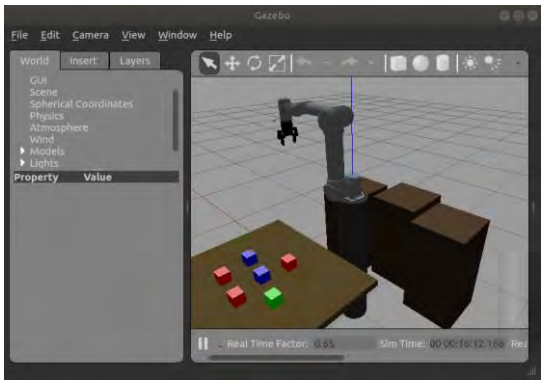


Imagen 187-Posición origen robot

Imagen 188-Posición al ejecutar modo 3

Como vemos en la imagen 186, en este caso se ha respondido con un 1 a la consulta de modificar la orientación, por tanto, seguidamente se ha pedido el cuaternión (x, y, z, w) que defina la orientación deseada del extremo del robot.

Se ha introducido el cuaternión (0, 1, 0, 0). Si comparamos la imagen 187 con la 188 podemos ver que la orientación de la pinza se ha modificado.

- **Ejemplo *pick and place* con la aplicación manual**

Acto seguido, se va a exponer un ejemplo de la posibilidad de usar el controlador manual para realizar un *pick and place* simple. Con este ejemplo se refleja la importancia que tuvo la construcción de la aplicación de control manual, para aprender controles que después nos sirvieron para el desarrollo de la aplicación final de *pick and place.*

Primero utilizamos el modo 6, para llevar al robot a la posición llamada *Home*.

```
-------------------------------------------------
OPERATING MODES:
1: Direct kinematic
2: Direct kin - Manual mode
3: Inverse kinematic - MoveJ
4: Open gripper
5: Close gripper
6: Go Home
7: Go camera position
8: See the object coordenates
9: Close program
-------------------------------------------------

Select one mode: 6
MOVING TO HOME POSITION
```

Imagen 189-Uso modo 6

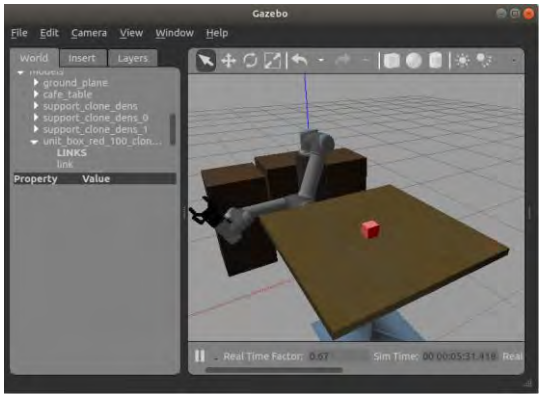


Imagen 190-Robot en posición inicial

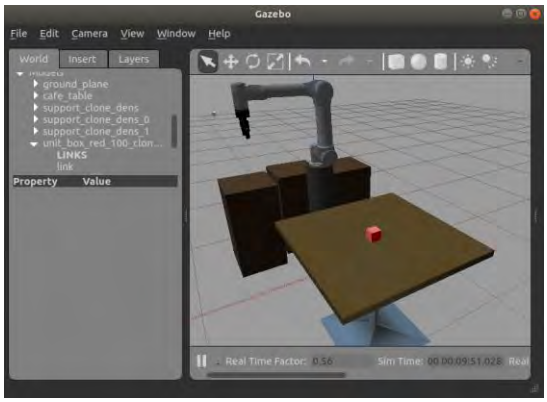


Imagen 191-Robot en posición Home

Y ahora con seleccionando el modo 7, mandaremos el robot a la posición desde la cual se toman imágenes. Aunque en este ejemplo no necesitamos las coordenadas de las piezas, se va a llevar el robot a esa posición para demostrar la utilidad de la aplicación manual como objetivo de familiarización con *MoveIt* para luego ser capaz de realizar todo el *pick and place*, en el que sí que se necesitará recoger mediante la cámara, las coordenadas de las piezas.

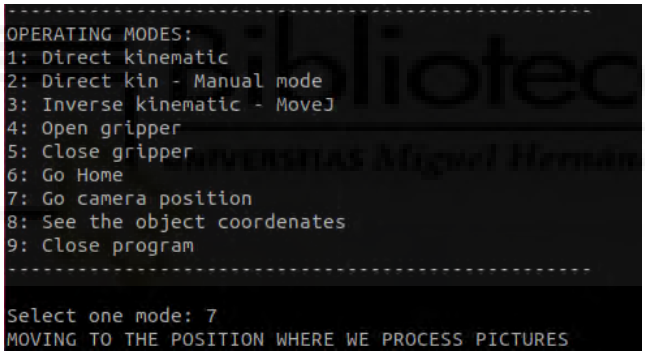


Imagen 192-Uso modo 7

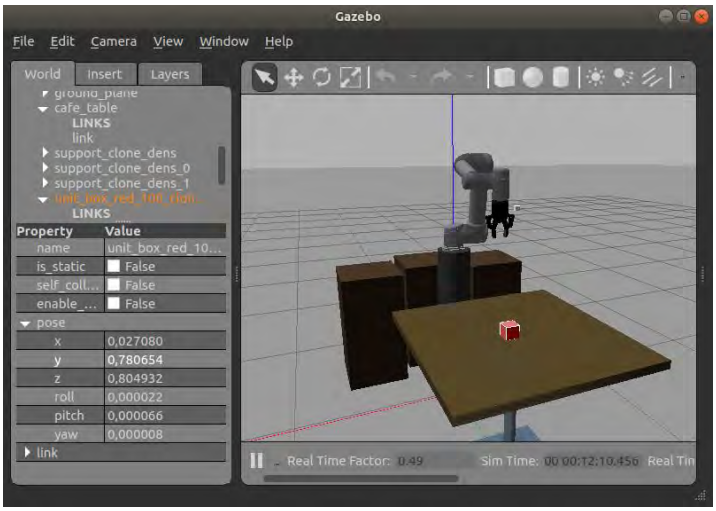


Imagen 193-Robot en posición de tomar imágenes

Posteriormente, utilizaremos el modo 3 para aproximar al robot a la pieza. Utilizando la cinemática inversa enviaremos al robot a la posición x=0.027080, y=0.780654, z=1.1; en esas coordenadas, x e y es son idénticas a la posición de la pieza roja y z es la altura a la que colocaremos la pinza como aproximación a la pieza.

```
Select one mode: 3

INVERSE KINEMATIC
Current values of end effector pose:
header:
  seq: 0
  stamp:
    secs: 832
    nsecs: 598000000
  frame_id: "world"
pose:
  position:
    x: -0.000269300634776
    y: 0.608279935471
    z: 1.35517332882
  orientation:
    x: -0.497423881671
    y: 0.493468650771
    z: 0.503684255141
    w: 0.50533191446

Enter new end effector values:

Enter position x: 0.027080
Enter position y: 0.780654
Enter position z: 1.1
Do you want to modify the orientation using a quaternion?(Yes(1) or No(2)): 2
```

Imagen 194-Aproximación a pieza con el modo 3

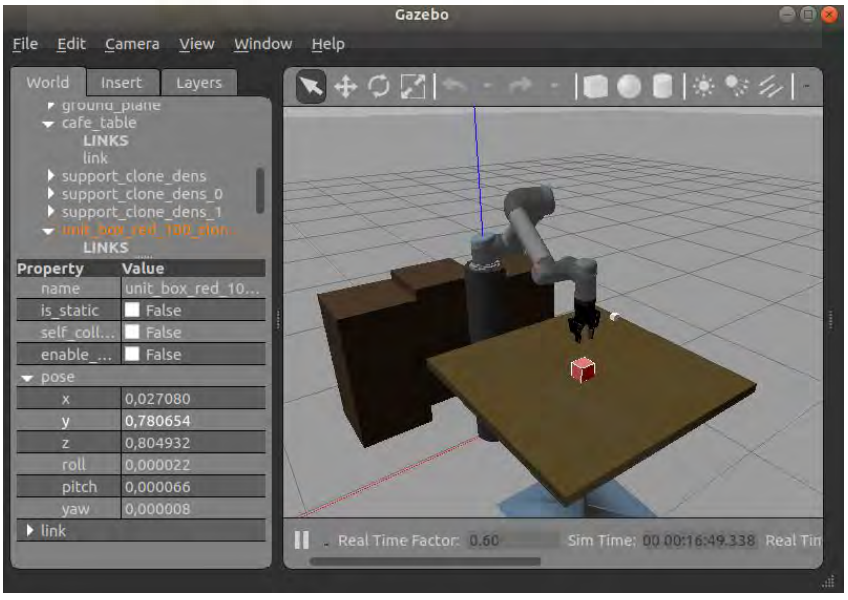


Imagen 195-Robot aproximándose a pieza roja

Ahora tenemos que abrir la pinza, y bajarla los suficiente, como para poder coger la pieza.

```
-------------------------------------------------
OPERATING MODES:
1: Direct kinematic
2: Direct kin - Manual mode
3: Inverse kinematic - MoveJ
4: Open gripper
5: Close gripper
6: Go Home
7: Go camera position
8: See the object coordenates
9: Close program
-------------------------------------------------

Select one mode: 4
OPENING THE GRIPPER
```

Imagen 196-Abriendo pinza

```
Select one mode: 3

INVERSE KINEMATIC
Current values of end effector pose:
header:
  seq: 0
  stamp:
    secs: 1092
    nsecs:  77000000
  frame_id: "world"
pose:
  position:
    x: 0.0269765098368
    y: 0.780522826469
    z: 1.09968136564
  orientation:
    x: -0.497527948033
    y: 0.49384349299
    z: 0.503904319395
    w: 0.504643420894

Enter new end effector values:

Enter position x: 0.0269765098368
Enter position y: 0.780522826469
Enter position z: 0.94
Do you want to modify the orientation using a quaternion?(Yes(1) or No(2)): 2
```


Imagen 197-Uso modo 3 para preparar la pinza a coger la pieza

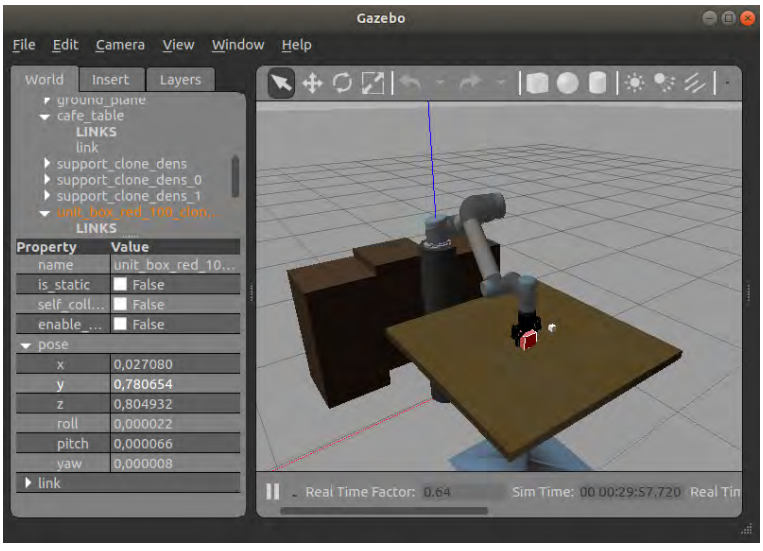


Imagen 198-Robot preparado para coger la pieza

Ahora cerramos la pinza, para recoger la pieza, y elevamos la pinza.

```
------------------------------------------------
OPERATING MODES:
1: Direct kinematic
2: Direct kin - Manual mode
3: Inverse kinematic - MoveJ
4: Open gripper
5: Close gripper
6: Go Home
7: Go camera position
8: See the object coordenates
9: Close program
------------------------------------------------

Select one mode: 5
CLOSING THE GRIPPER
```

Imagen 199-Cerrando pinza

```
Select one mode: 3

INVERSE KINEMATIC
Current values of end effector pose:
header:
  seq: 0
  stamp:
    secs: 2156
    nsecs: 573000000
  frame_id: "world"
pose:
  position:
    x: 0.0268962247218
    y: 0.780279845194
    z: 0.959130393539
  orientation:
    x: -0.510360306504
    y: 0.486027344237
    z: 0.507779596186
    w: 0.495448947821

Enter new end effector values:

Enter position x: 0.0268962247218
Enter position y: 0.780279845194
Enter position z: 1.4
Do you want to modify the orientation using a quaternion?(Yes(1) or No(2)): 2
```

Imagen 200-Elevación del extremo del robot con la pieza en la pinza

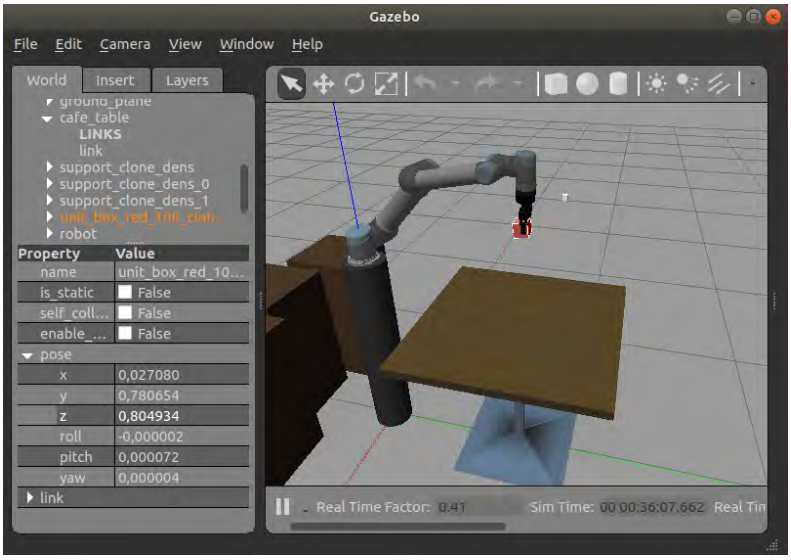


Imagen 201-Robot con la pieza entre las pinza

Para continuar, aproximaremos al robot al soporte en el cual vamos a dejar la pieza.

```
Select one mode: 3

INVERSE KINEMATIC
Current values of end effector pose:
header:
  seq: 0
  stamp:
    secs: 2215
    nsecs: 171000000
  frame_id: "world"
pose:
  position:
    x: 0.450388987087
    y: -0.49958482959
    z: 1.00060709444
  orientation:
    x: -0.50981006551
    y: 0.486904456231
    z: 0.507590171701
    w: 0.495348327139

Enter new end effector values:

Enter position x: 0.450388987087
Enter position y: -0.49958482959
Enter position z: 0.967
Do you want to modify the orientation using a quaternion?(Yes(1) or No(2)): 2
```

Imagen 202-Aproximación al soporte para dejar la pieza

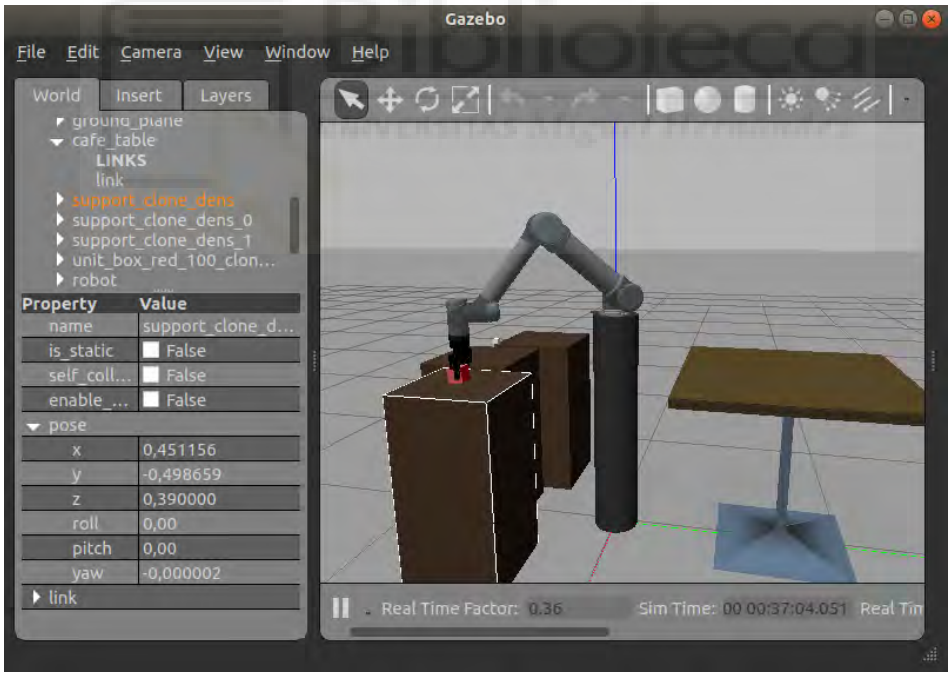


Imagen 203-Robot preparado para dejar la pieza

En este punto solo queda abrir la pinza y subir el el extremo del robot.

```
------------------------------------------------
OPERATING MODES:
1: Direct kinematic
2: Direct kin - Manual mode
3: Inverse kinematic - MoveJ
4: Open gripper
5: Close gripper
6: Go Home
7: Go camera position
8: See the object coordenates
9: Close program
------------------------------------------------

Select one mode: 4
OPENING THE GRIPPER
```

Imagen 204-Cerrando pinza con seleccionando el 4

```
Select one mode: 3

INVERSE KINEMATIC
Current values of end effector pose:
header:
  seq: 0
  stamp:
    secs: 2234
    nsecs: 682000000
  frame_id: "world"
pose:
  position:
    x: 0.449890088385
    y: -0.50057985086
    z: 0.967723312624
  orientation:
    x: -0.509748990453
    y: 0.488212897507
    z: 0.505413662365
    w: 0.496347824952

Enter new end effector values:

Enter position x: 0.449890088385
Enter position y: -0.50057985086
Enter position z: 1.1
Do you want to modify the orientation using a quaternion?(Yes(1) or No(2)): 2
```

Imagen 205-Subir el robot para liberar la pieza usando el modo 3

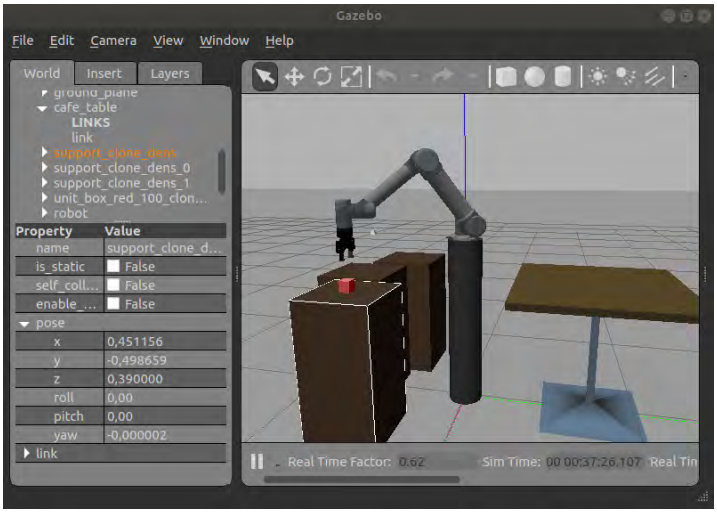


Imagen 206-Pieza depositada en soporte

- **Modo 2**

La última selección disponible dentro de la aplicación manual es el modo de operación 2. La funcionalidad de este modo es, que el usuario pueda mover el robot en tiempo real utilizando las teclas del teclado. Por tanto, cuando ejecutemos este modo de operación, podremos mover las articulaciones del robot utilizando unas teclas concretas del teclado.

Cuando seleccionamos el modo 2, por el terminal aparecerá una interfaz de usuario.

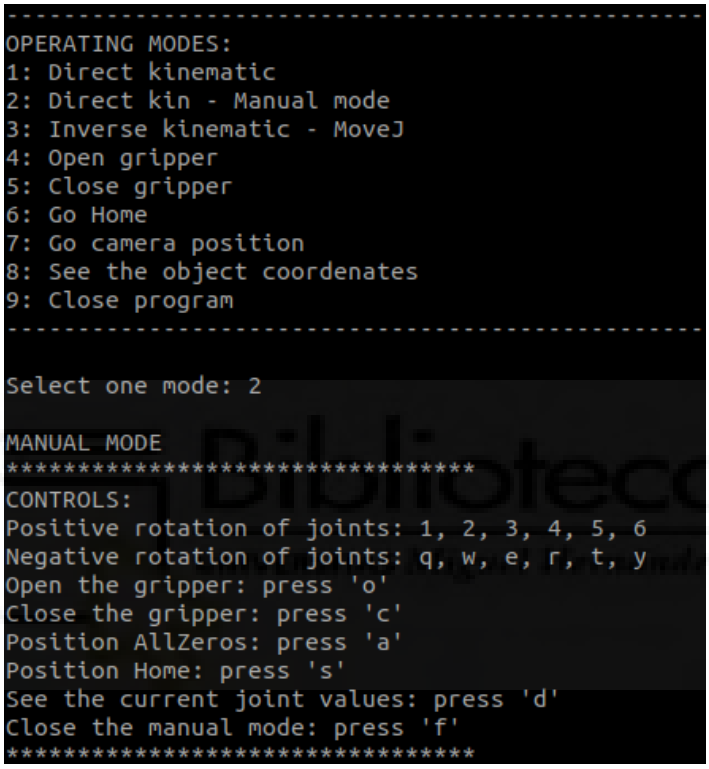


Imagen 207-Interfaz modo 2

Como seremos capaces de visualizar en la imagen 207, en el momento de seleccionar el modo 2, por terminal aparece una interfaz que nos indica las acciones que podemos realizar sobre el robot, al pulsar las distintas teclas del teclado.

Para entender la capacidad de este modo de operación, podemos decir que es como si manipuláramos al robot desde un mando de control remoto ya que, podemos manejar las articulaciones del robot en tiempo real con unas teclas específicas del teclado.

Debajo de *CONTROLS*, aparecen los controles disponibles desde el teclado. Si pulsamos la tecla *a*, el robot se desplazará hasta la posición llamada *AllZeros* y al pulsar la tecla *s*, el robot se desplazará hasta la posición *Home.*

A continuación, vamos a visualizar una imagen con el robot en posición inicial y debajo de dicha imagen, veremos al robot en la posición *AllZeros* y a la derecha el robot en *Home.*

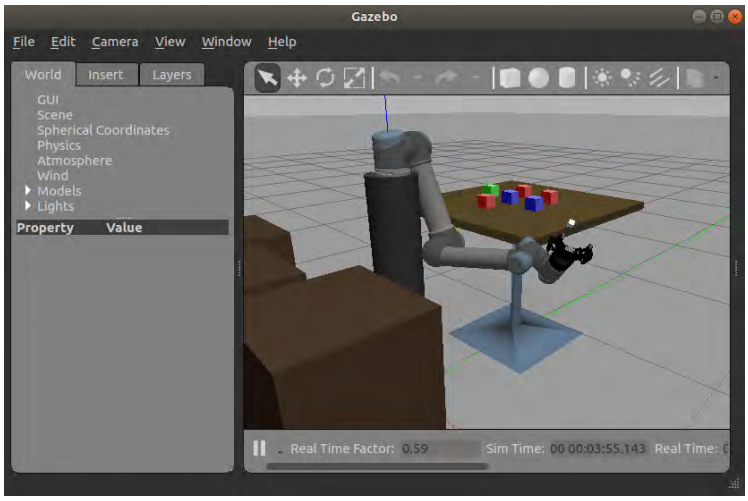


Imagen 208-Posición inicial modo 2

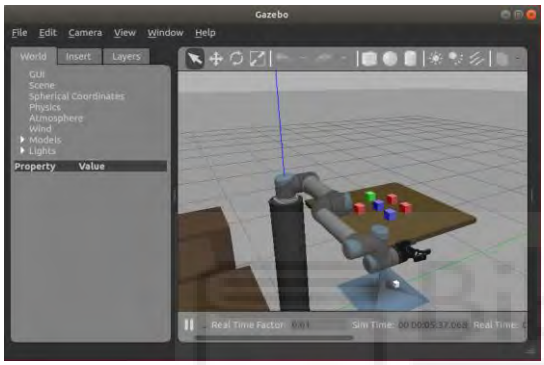


Imagen 209-Robot en posición AllZeros

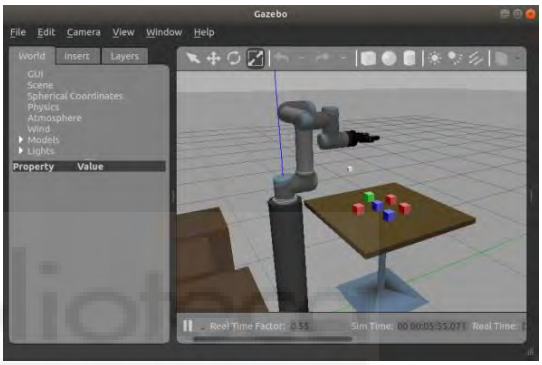


Imagen 210-Robot en posición Home

Además, pulsando las teclas *o* y *c*, podemos abrir y cerrar la pinza.

Pulsando o

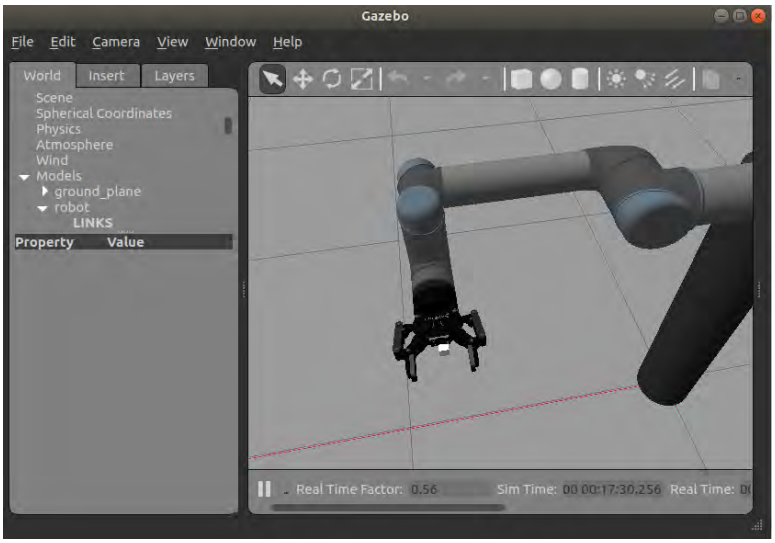


Imagen 211-Resultado modo 2 pulsando o

Pulsando c

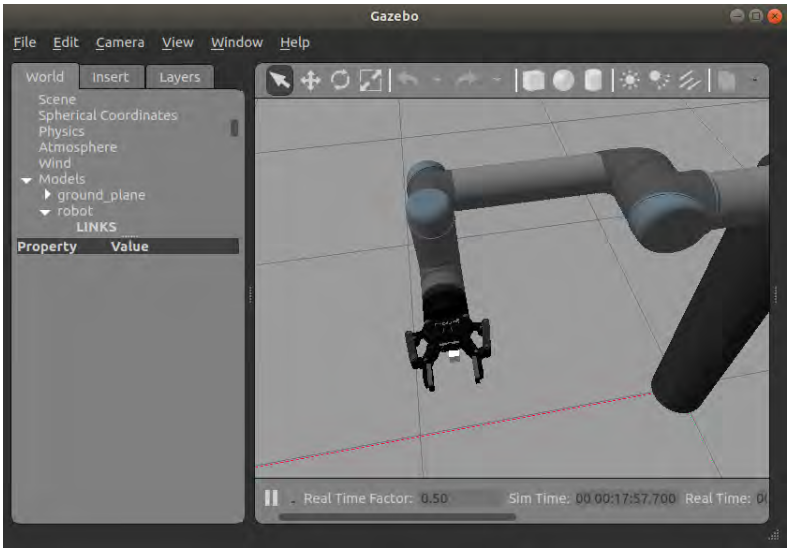


Imagen 212-Resultado modo 2 pulsando c

Al pulsar la tecla *d*, se mostrarán las posiciones angulares de las articulaciones, en ese mismo momento que se ha pulsado dicha tecla. Por tanto, al pulsar *d*, podremos ver los siguiente por terminal.

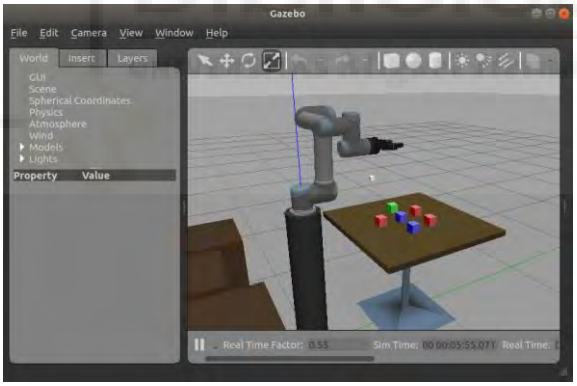


Imagen 213-Robot en posición Home

```
CONTROLS:
Positive rotation of joints: 1, 2, 3, 4, 5, 6
Negative rotation of joints: q, w, e, r, t, y
Open the gripper: press 'o'
Close the gripper: press 'c'
Position AllZeros: press 'a'
Position Home: press 's'
See the current joint values: press 'd'
Close the manual mode: press 'f'
********************************
Current joint values:
[3.91365895007211 86e-05, -1.4982020857473923, 1.5092184101918935, -0.0020038261235972854, -0.00039112
747569802764, 7.566895205979307e-05]
```

Imagen 214-Valores articulares del robot en posición Home

Entre los diferentes controles podemos ver que con las teclas 1, 2, 3, 4, 5 y 6, movemos las articulaciones del robot en el sentido positivo de dichas articulaciones. Con las teclas q, w, e, r, t e y, podemos comandar las articulaciones en el sentido negativo de giro.

A continuación, vamos a ver un ejemplo de movimiento del robot, utilizando las teclas que mueven las articulaciones del robot en sentido positivo.

Pulsamos 5 veces tecla 1

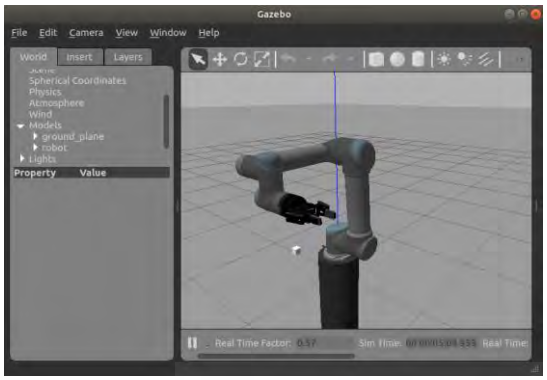


Imagen 215-Robot antes de pulsar tecla 1

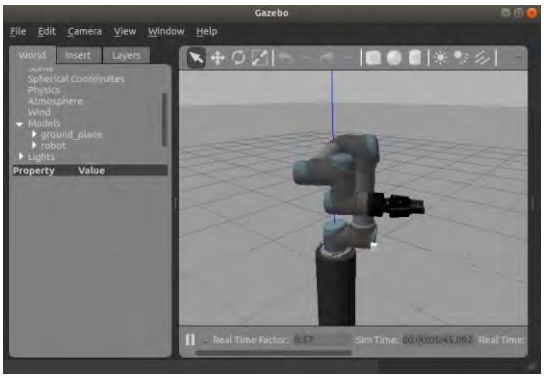


Imagen 216-Robot después de pulsar tecla 1

Podemos ver en la imagen 216, que pulsando la tecla 1 giramos la primera articulación en el sentido positivo de giro.

Pulsamos 5 veces la tecla 3

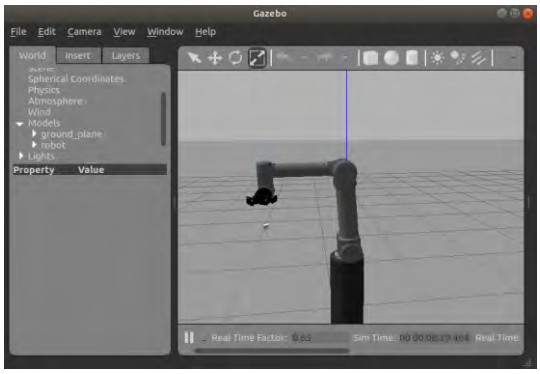


Imagen 217-Robot antes de pulsar tecla 3

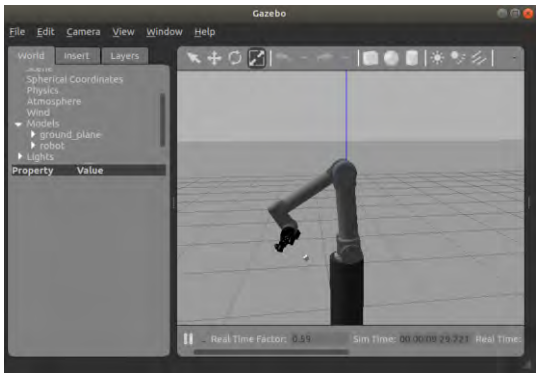


Imagen 218-Robot después de pulsar tecla 3

Observamos que al pulsar la tecla 3, se mueve la tercera articulación en la orientación positiva.

Pulsamos 5 veces la tecla 5

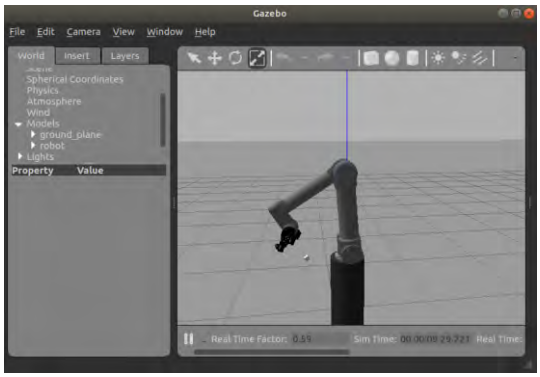


Imagen 219-Robot antes de pulsar tecla 5

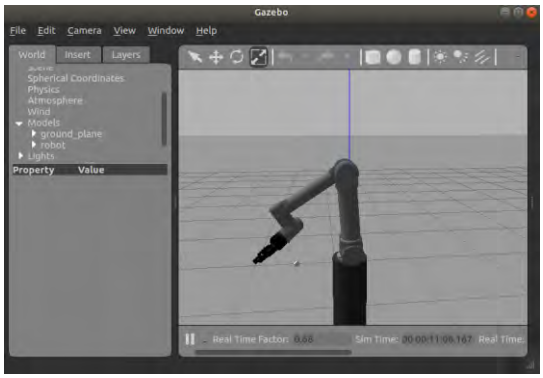


Imagen 220-Robot después de pulsar tecla 5

Con la tecla 5 movemos la quinta articulación en el sentido positivo de la articulación. Con las teclas 2, 4 y 6, también podemos mover las articulaciones segunda, cuarta y sexta, en el sentido positivo de giro.

A continuación, vamos a mostrar varios ejemplos de movimiento de las mismas articulaciones que hemos rotado hasta ahora, pero lo vamos a realizar en el sentido negativo de la articulación.

Pulsamos 5 veces la tecla q

Imagen 221-Robot antes de pulsar tecla q

Imagen 222-Robot después de pulsar tecla q

Según se puede constatar en las imágenes, se ha rotado la primera articulación en el sentido negativo de la articulación. Si comparamos estas, con las imágenes 215 y 216 nos daremos cuenta que tanto pulsando 1 como pulsando la letra q, movemos la misma articulación, pero en orientaciones contrarias.

Pulsamos 5 veces la tecla e

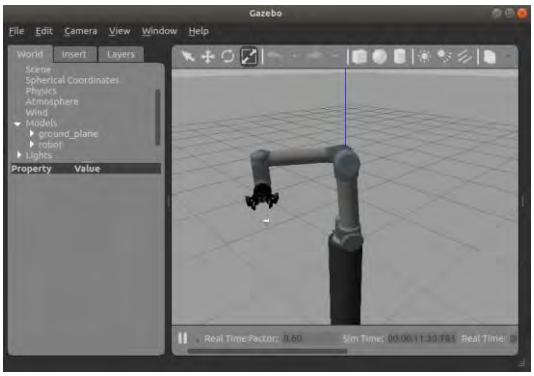


Imagen 223-Robot antes de pulsar tecla e

Imagen 224-Robot después de pulsar tecla e

Con la tecla e movemos la tercera articulación en el sentido negativo.

Pulsamos 5 veces la tecla t

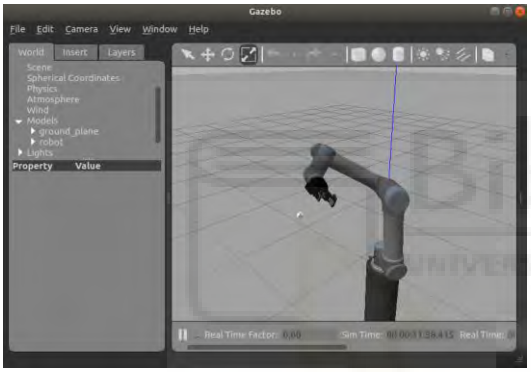


Imagen 225-Robot antes de pulsar tecla t

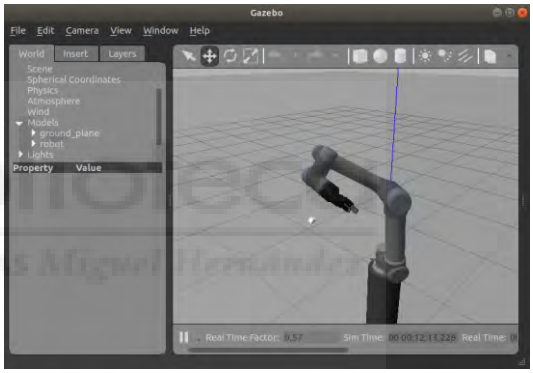


Imagen 226-Robot después de pulsar tecla t

Con la tecla t movemos la tercera articulación en el sentido negativo.

Debemos saber que pulsando las teclas w, r, e y, podemos rotar las articulaciones segunda, cuarta y sexta respectivamente, en su orientación negativa.

Con esto, ya hemos terminado de mostrar los resultados de la aplicación manual de control del robot. Hemos podido ver, que esta aplicación nos proporciona una serie de métodos, los cuales podemos seleccionar desde la interfaz de la imagen 166, con los cuales podemos mover el robot de la forma en que lo necesitemos.

6.2. Simulación aplicación de pick and place

En el apartado 6.2 de la memoria se van a mostrar los resultados de simulación del que se considera el objetivo de aplicación principal, la aplicación de *pick and place.*

Aunque en el apartado 5.2.3, se ha expuesto la aplicación de *pick and place* en dos partes, primero el desarrollo primario de la aplicación y en segundo lugar la mejora que se implementó dentro de dicha aplicación, en la exposición de los resultados se va a mostrar la ejecución de la aplicación final que incluye la mejora que se introdujo en el apartado 5.2.3.2.

En la extensión de este apartado se van a mostrar dos ejemplos de ejecución de la aplicación, uno en el que la escena inicial de las piezas no se va a modificar, es decir, ni se van a mover las piezas ni se van a quitar o añadir piezas mientras el robot esté ejecutándose. Y un segundo ejemplo en el que mientras el robot se esté moviendo, se van a mover y quitar piezas, para demostrar que la mejora implementada en el apartado 5.2.3.2 ha dado sus resultados.

Lo primero que se debe presentar es como se debe lanzar y ejecutar la aplicación de *pick and place* por el terminal. Para empezar, debemos ejecutar el comando *roscore*, para inicializar el ecosistema de *ROS* y utilizar sus funcionalidades. Esto también lo hicimos para ejecutar la aplicación manual, podemos verlo en las imágenes 161 y 162.

Seguidamente, debemos lanzar la simulación de *Gazebo* en la que se encuentra el robot *UR5* y los objetos necesarios para llevar a cabo el *pick and place*. Podemos lanzar dicho modelo de simulación con el siguiente comando ejecutado por terminal.

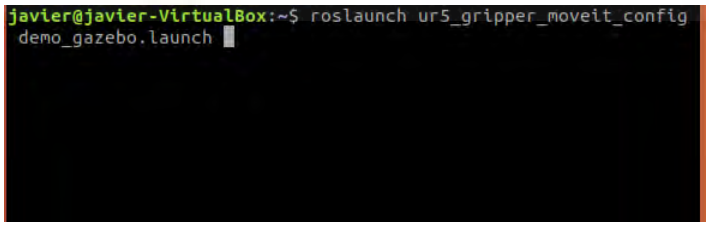


Imagen 227-Lanzamiento simulación Gazebo

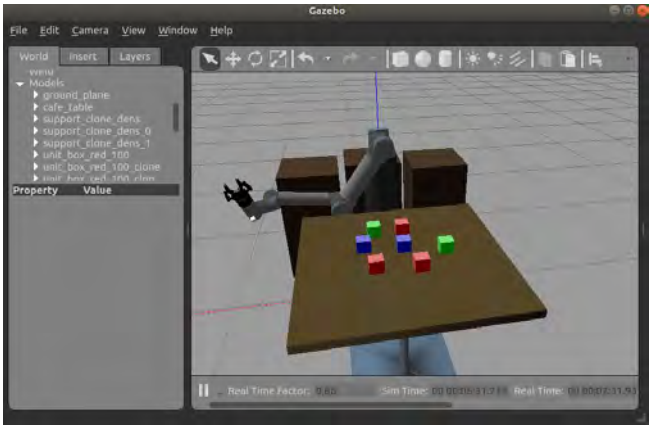


Imagen 228-Simulación del robot en Gazebo

Si lo comparamos con el modelo de simulación de la imagen 168, nos daremos cuenta de que se ha añadido un cubo verde más, esto se ha realizado con la intención de demostrar que se puede ejecutar la aplicación con más de un objeto por color.

Una vez ya tenemos el entorno de *Gazebo* lanzado, podemos ejecutar el programa que contiene la aplicación del *pick and place*. Concretamente se lanzará el programa que se ha desarrollado en el apartado 5.2.3. Para lanzar dicho programa, debemos ejecutar por terminal el siguiente comando.

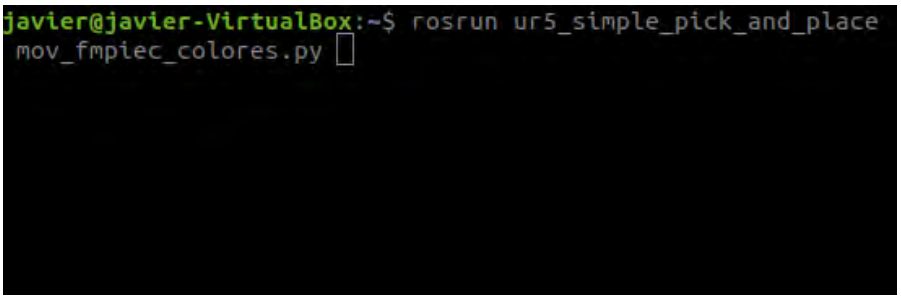


Imagen 229-Comando ejecución pick and place

Ya sabemos que *ur5_simple_pick_and_place* es el paquete donde se ubican los diferentes programas de aplicación. En este caso *mov_fmpiec_colores.py* es el archivo de *Pyhton* que contiene el programa desarrollado para comandar al robot en la realización del *pick and place.*

Al ejecutar dicho comando se iniciará, en la simulación de *Gazebo,* la ejecución de la aplicación. A continuación, vamos a ver imágenes de la ejecución de dicha aplicación.

6.2.1. Ejecución normal

Como ya hemos comentado anteriormente, se va a mostrar primero un ejemplo de ejecución de la aplicación de *pick and place*, en el que no se va a modificar el escenario de *Gazebo.* Es decir, el *pick and place* se va a ejecutar integro sin mover ninguna pieza durante el movimiento del robot, ni modificar el escenario de piezas de la imagen 228.

A continuación, se van a exponer una sucesión de capturas que muestran, el resultado de la simulación de la aplicación de *pick and place*, o lo que es lo mismo, el movimiento que se da en la simulación de la imagen 228 cuando se ejecuta el comando de la imagen 229.

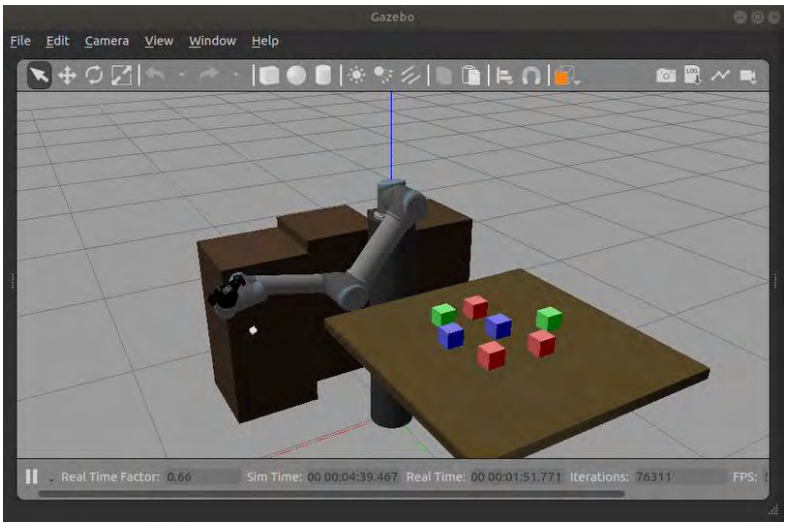


Imagen 230-Posición inicial pick and place

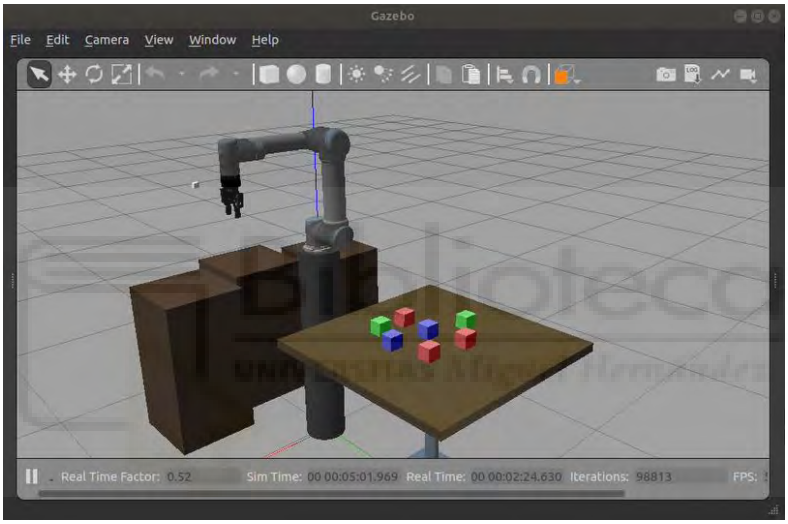


Imagen 231-Robot en Home antes de empezar el pick and place

Imagen 232-Robot en posición de tomar imágenes

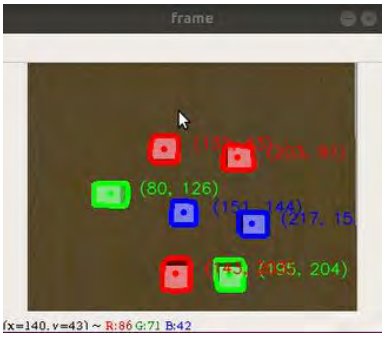


Imagen 233-Imagen la cámara

Al llevar al robot a esa posición, por terminal aparece la siguiente información relativa a las piezas de la mesa.

```
************************
El numero de piezas de cada color es, rojo:3, azul:2, verde:2
La cordenadas de los objetos son: [[217, 155, 150, 144], [195, 204, 80, 126], [143, 203, 203, 91, 132
, 83]]
************************
```

Imagen 234-Información de las piezas en el terminal

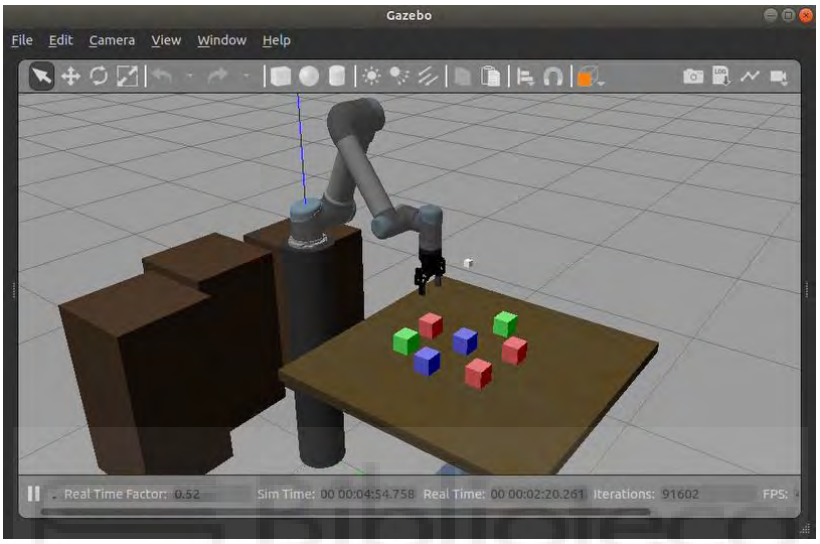


Imagen 235-Robot aproximándose a pieza roja

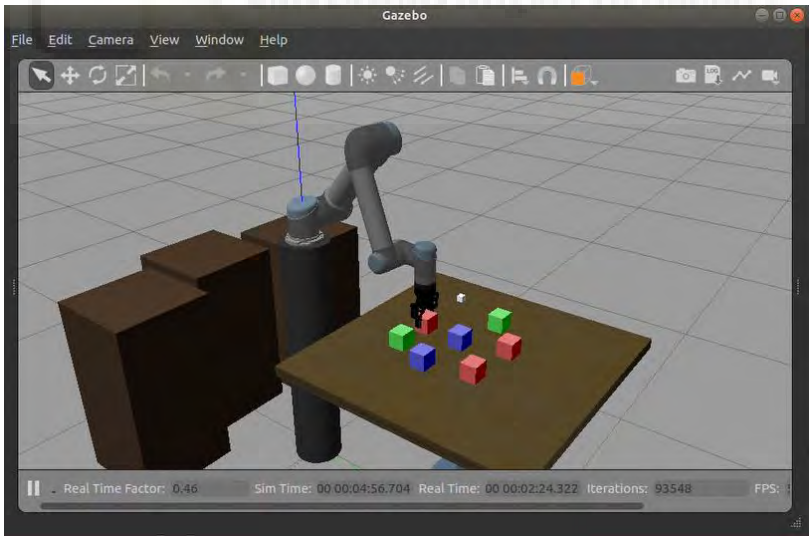


Imagen 236-Robot cogiendo pieza roja

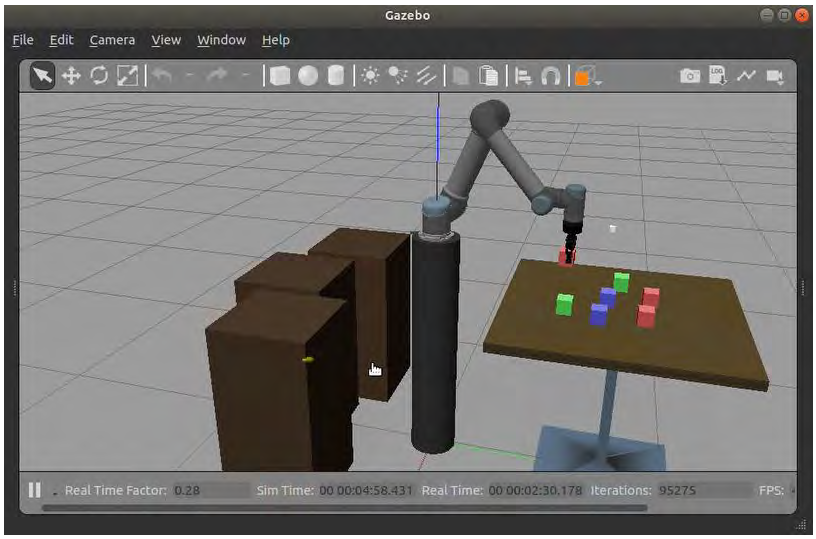


Imagen 237-Robot moviendo pieza roja

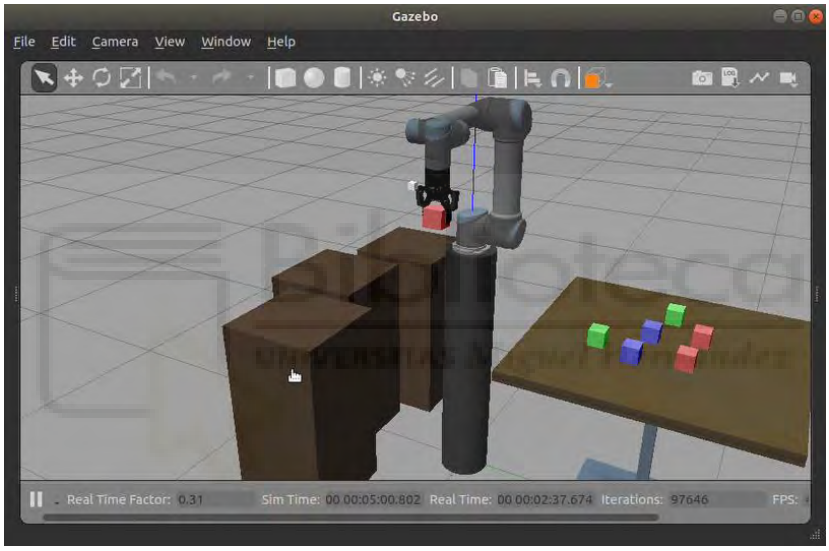


Imagen 238-Robot en Home antes de dejar la pieza

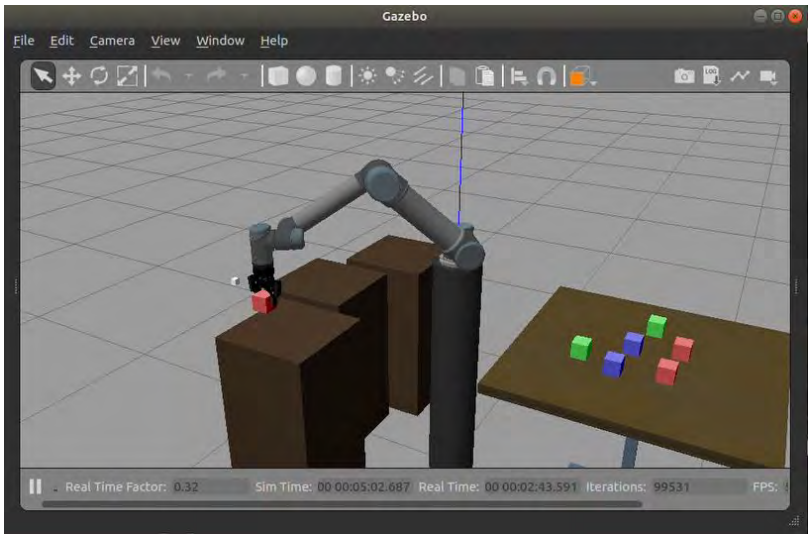


Imagen 239-Robto aproximándose a dejar pieza roja

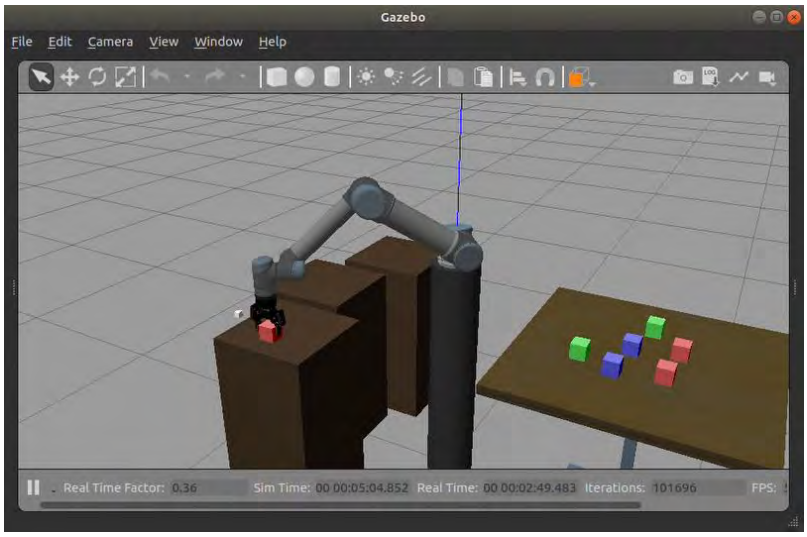


Imagen 240-Dejando pieza en soporte

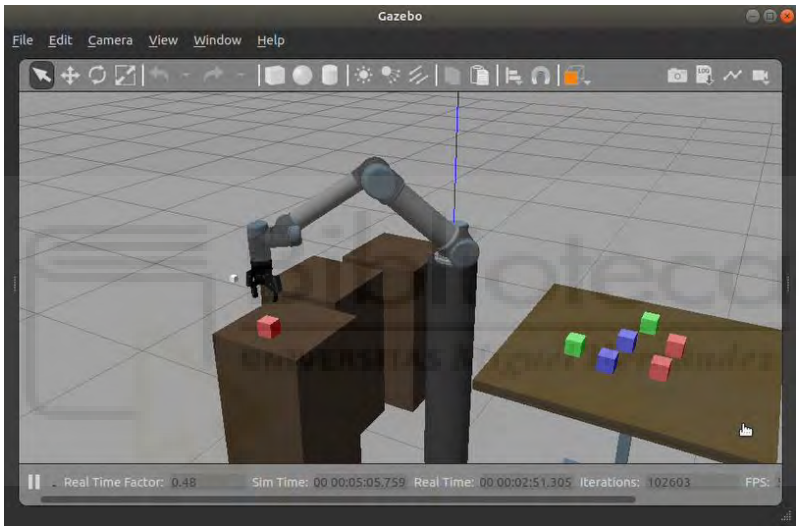


Imagen 241-Pieza depositada en soporte

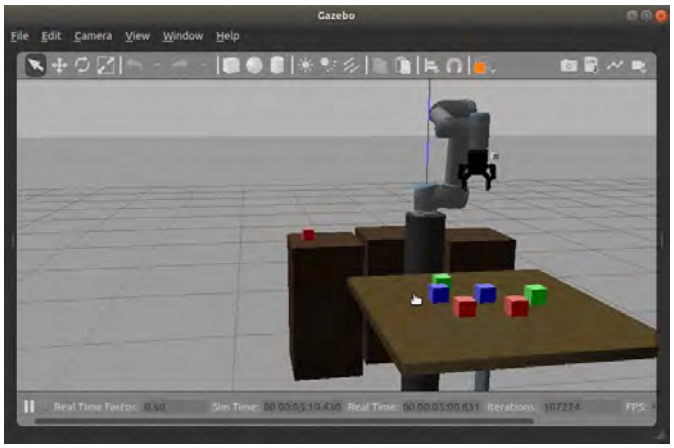


Imagen 242-Robot en posición de tomar imágenes

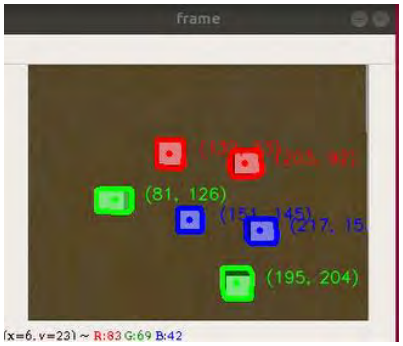


Imagen 243-Imagen la cámara

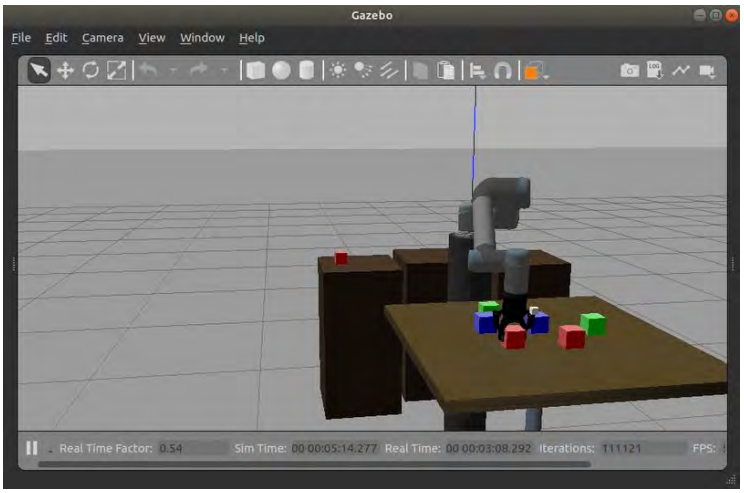


Imagen 244-Robot cogiendo segunda pieza roja

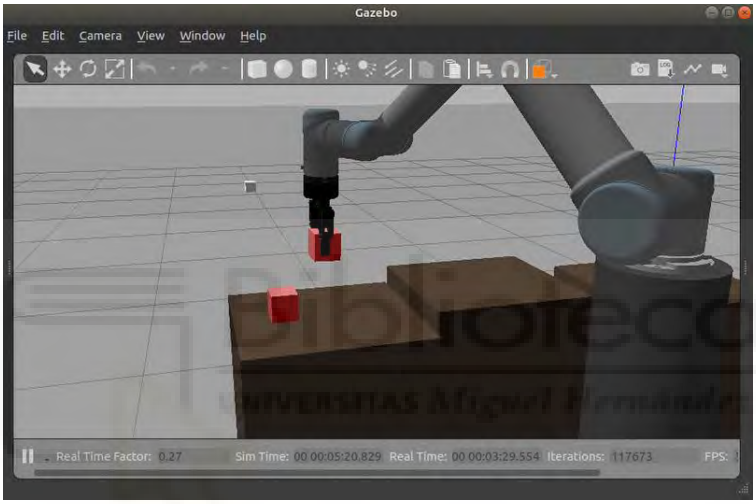


Imagen 245-Robot aproximándose a dejar segunda pieza

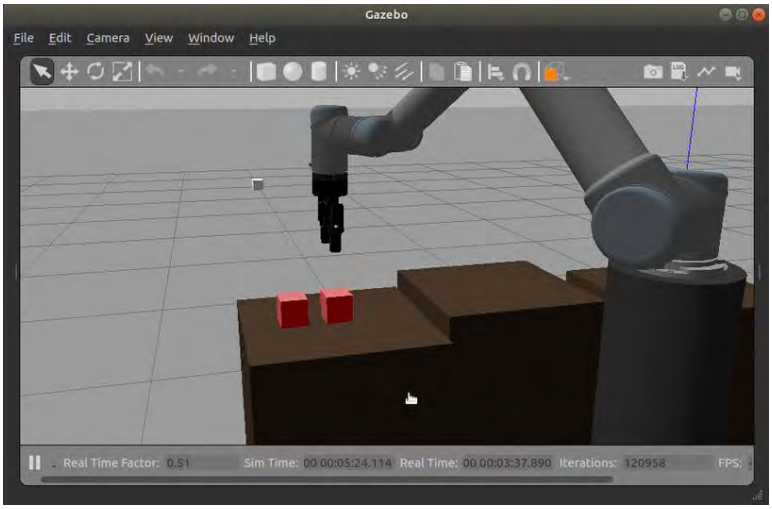


Imagen 246-Segunda pieza depositada

Imagen 247-Robot en posición de tomar imágenes

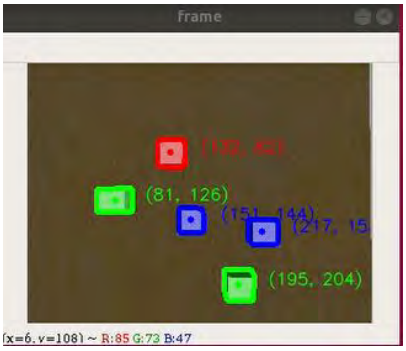


Imagen 248-Imagen la cámara

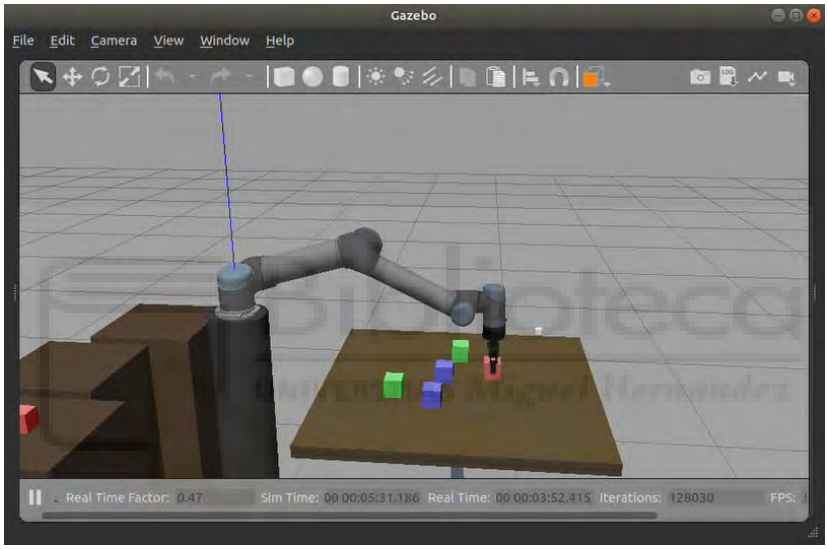


Imagen 249-Cogiendo tercera pieza roja

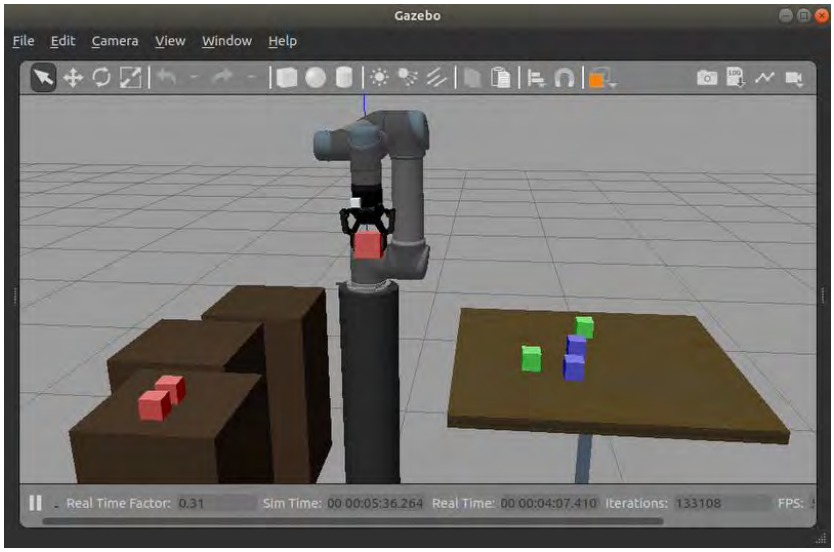


Imagen 250-Robot moviendo tercera pieza roja

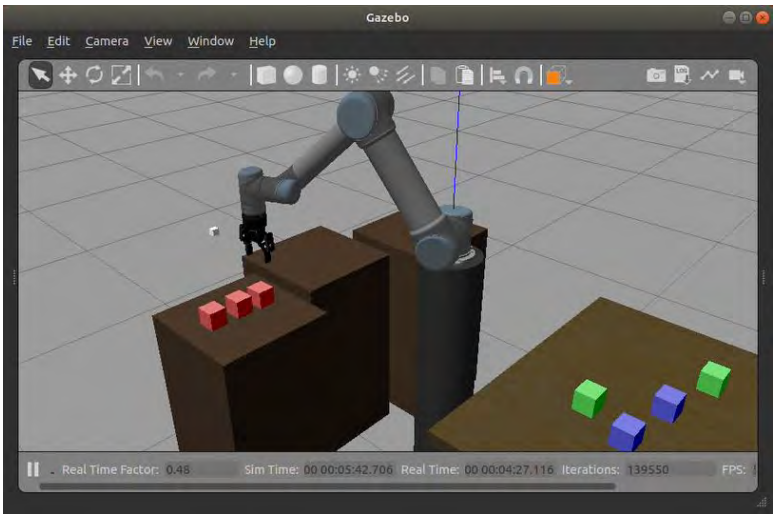


Imagen 251-Tercera pieza depositada

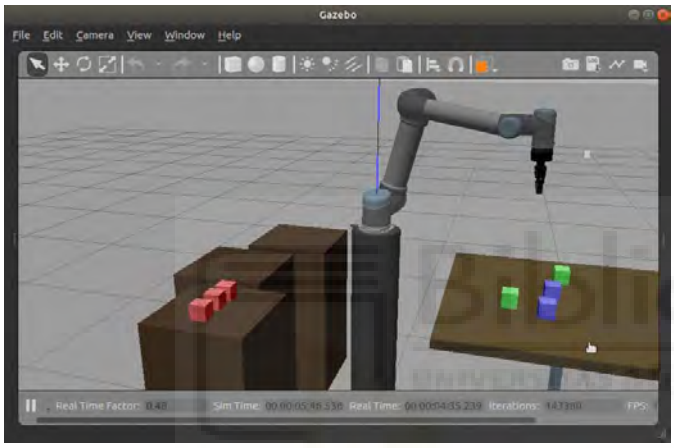


Imagen 252-Robot en posición de tomar imágenes

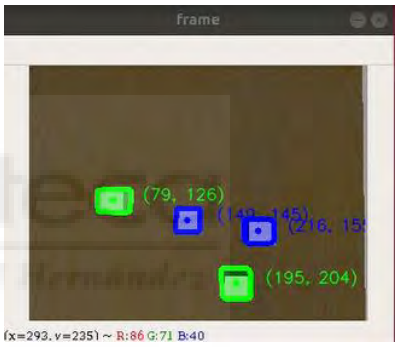


Imagen 253-Imagen la cámara

De ahora en adelante no se mostrará la imagen donde el robot está detectando las coordenadas, como por ejemplo en las imágenes 252 y 253. No se va a mostrar ese paso debido a que, al no mover ninguna pieza, las coordenadas no se van a actualizar.

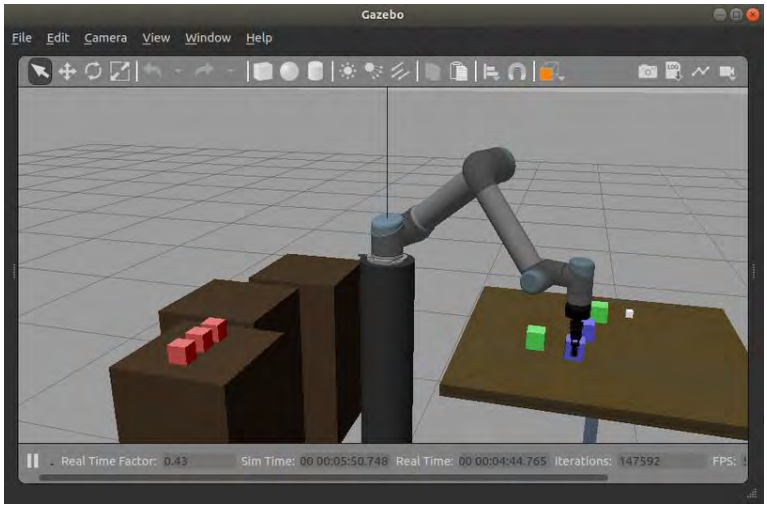


Imagen 254-Robot cogiendo pieza azul

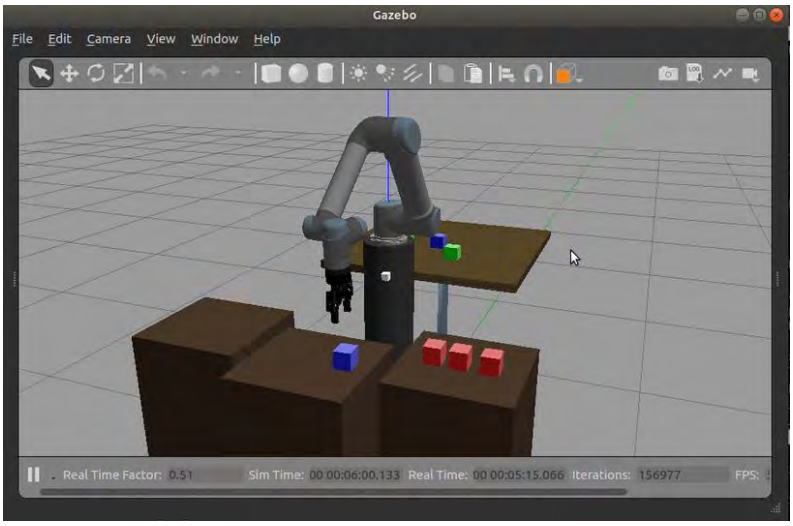


Imagen 255-Pieza azul depositada

Imagen 256-Robot cogiendo segunda pieza azul

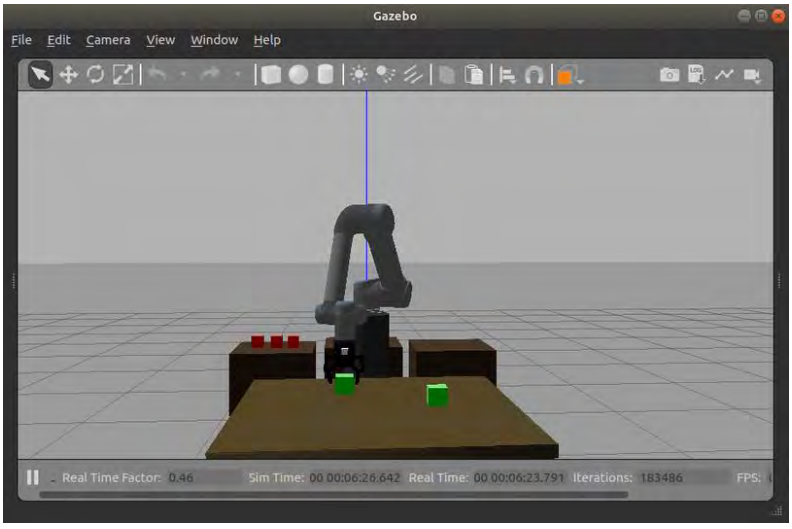


Imagen 257-Robot cogiendo pieza verde

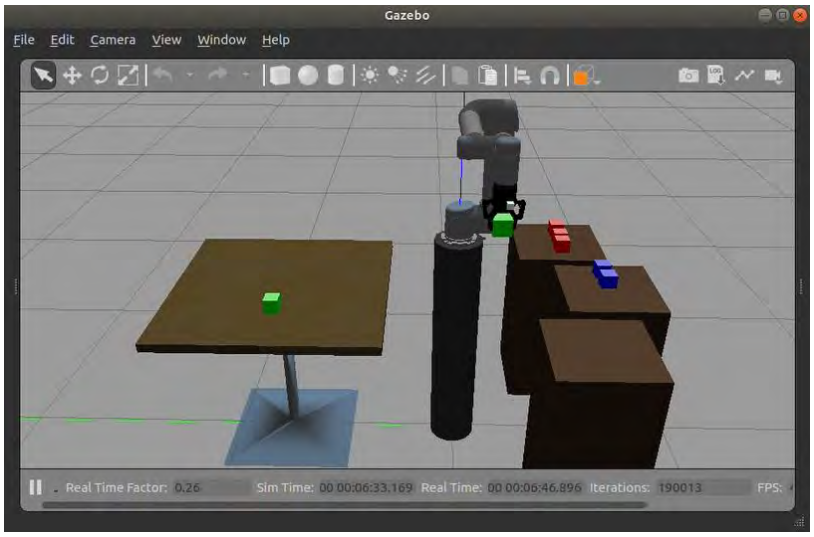


Imagen 258-Robot desplazando pieza verde

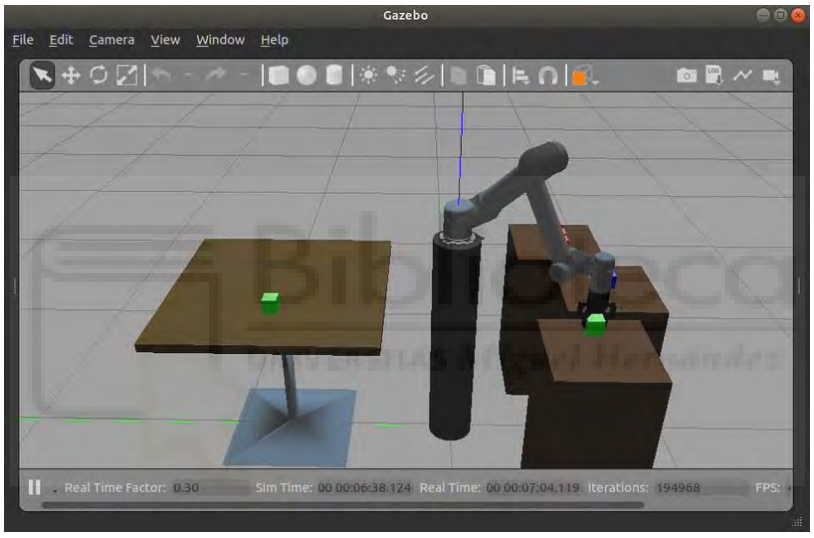


Imagen 259-Robot dejando pieza verde

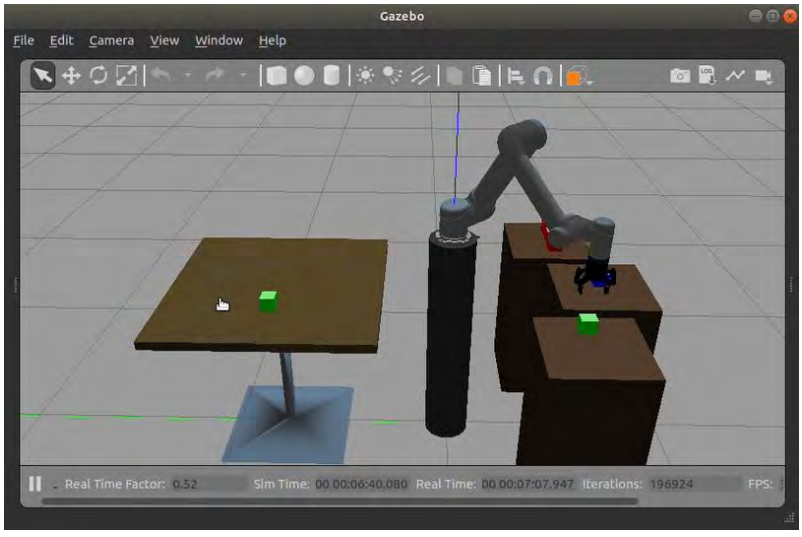


Imagen 260-Pieza verde depositada en soporte

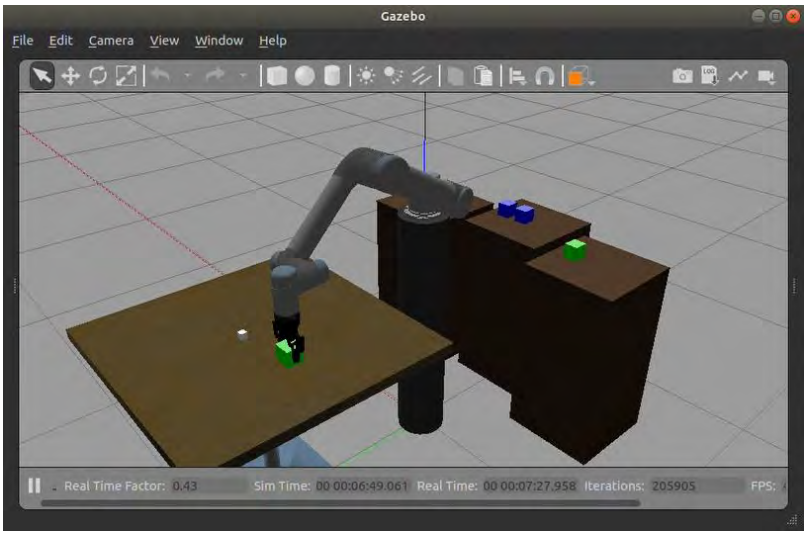


Imagen 261-Robot cogiendo segunda pieza verde

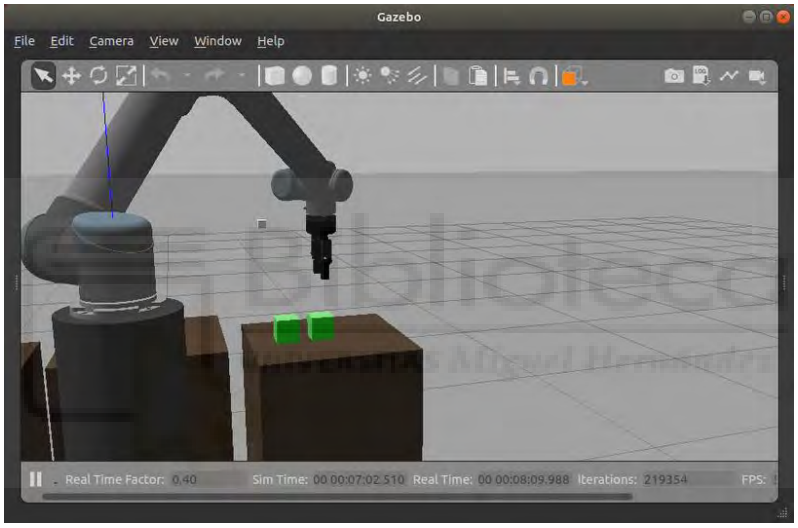


Imagen 262-Segunda pieza verde depositada

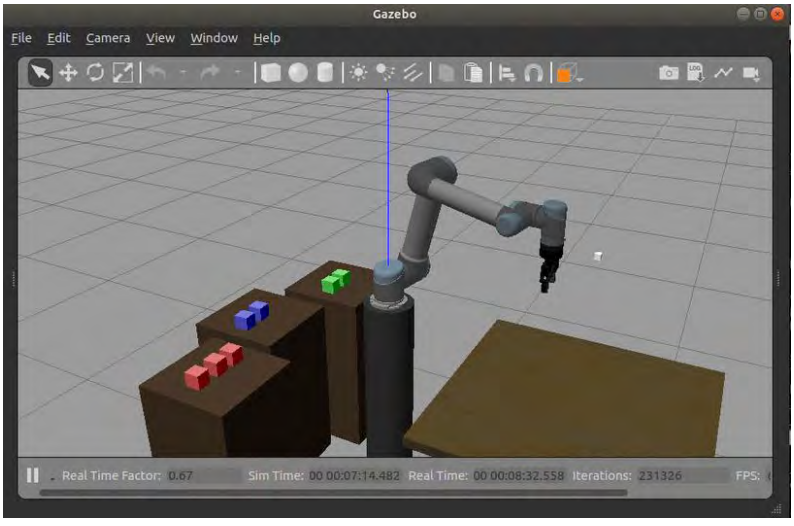


Imagen 263-Finalización pick and place

6.2.2. Ejecución modificando la escena mientras el robot se mueve

En este momento, se va a mostrar de nuevo una ejecución de el programa que lleva a cabo el *pick and place*, pero en esta sección, durante la ejecución del movimiento del robot, se van a cambiar de posición las piezas, se van a eliminar algunas de dichas piezas e incluso se añadirá una nueva pieza.

En este apartado no se van a poner las capturas tan en detalle del movimiento del robot como en el apartado 6.2.1, ya que lo que nos interesa en este caso es darnos cuenta de como reacciona el robot antes cambios en las coordenadas de las piezas.

A continuación, se van a mostrar una serie de capturas donde se ve el *pick and place* del robot *UR5*, pero en este caso las capturas se van a centrar más en los momentos en los que se modifica la escena de las piezas.

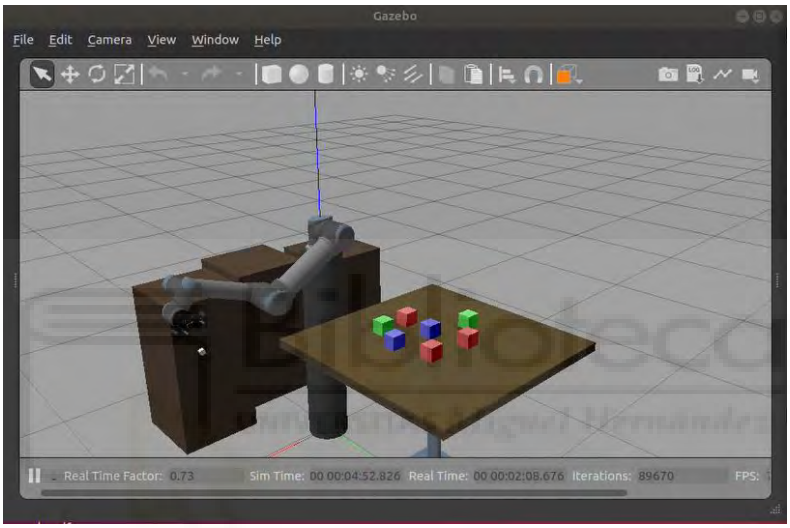


Imagen 264-Posición inicial robot

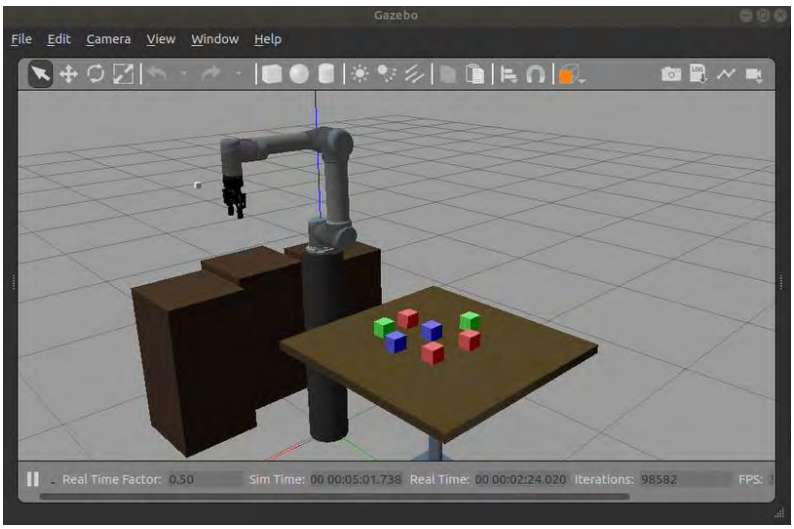


Imagen 265-Robot en posición Home

Ahora el robot se dirige a detectar las coordenadas de las piezas en su disposición inicial.

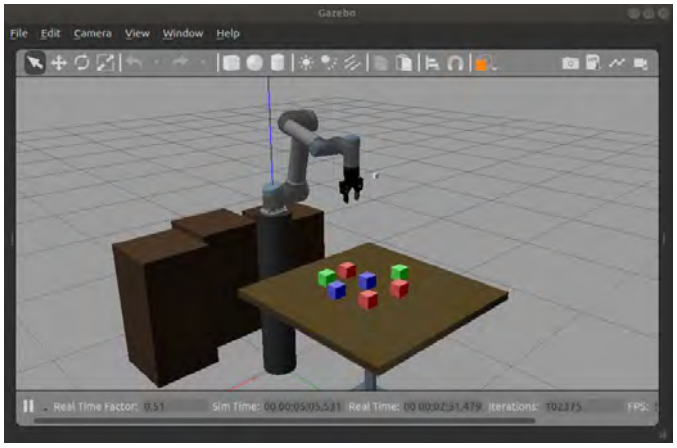


Imagen 266-Robot en posición de tomar imágenes

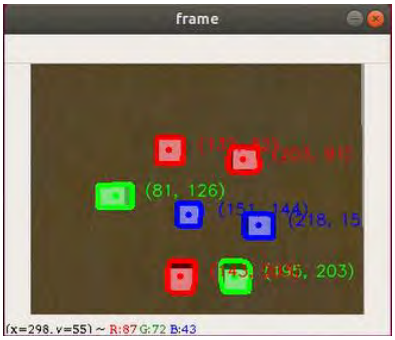


Imagen 267-Imagen la cámara

```
************************
El numero de piezas de cada color es, rojo:3, azul:2, verde:2
La cordenadas de los objetos son: [[218, 154, 151, 144], [195, 203, 81, 126], [143, 203, 203, 91, 132
, 82]]
************************
```

Imagen 268-Información de las piezas en el terminal

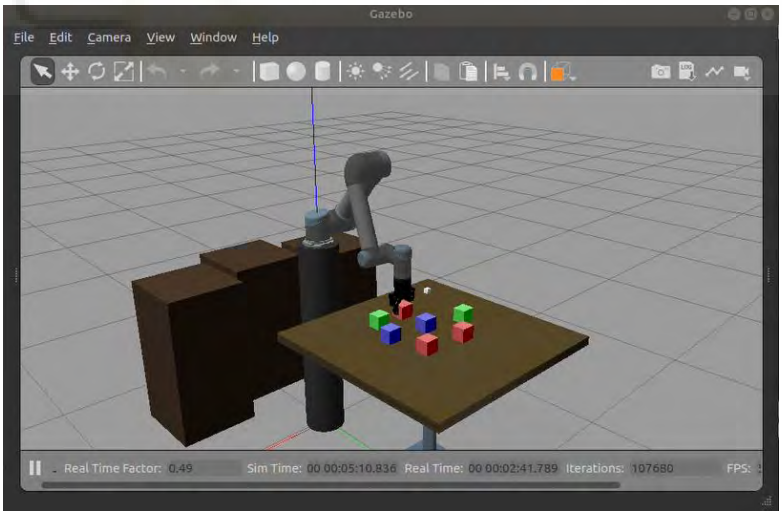


Imagen 269-Robot cogiendo pieza roja

Mientras el robot se dirige a depositar la pieza roja en su soporte, nos disponemos a mover una de las piezas rojas que hay en la mesa.

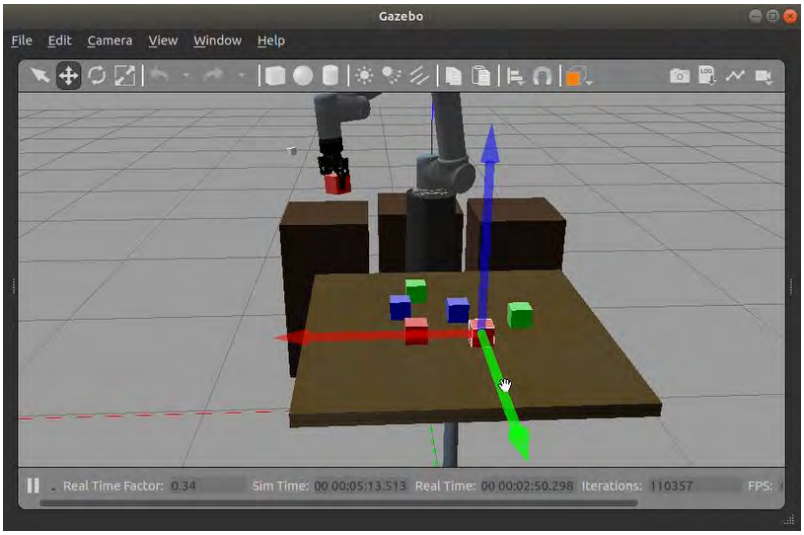


Imagen 270-Moviendo pieza roja

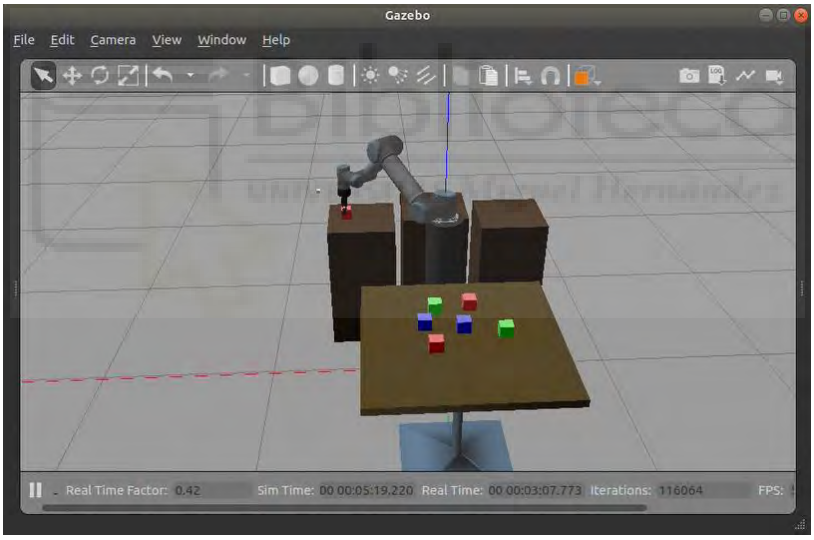


Imagen 271-Pieza roja desplazada mientras otra pieza roja es depositada

En este punto, ya hemos desplazado una de las piezas rojas, mientras el robot estaba dirigiéndose a dejar otra de las piezas roja. Ahora vamos a ver si el robot reconoce la pieza desplazada o si, por otro lado, realiza el *pick and place* de forma errónea

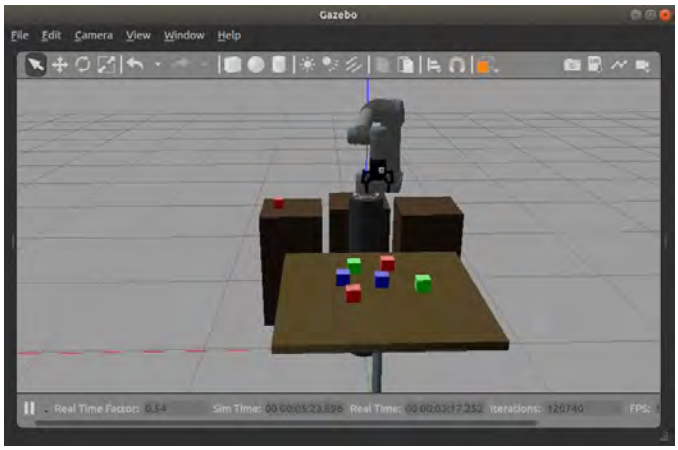


Imagen 272-Robot en posición de tomar imágenes

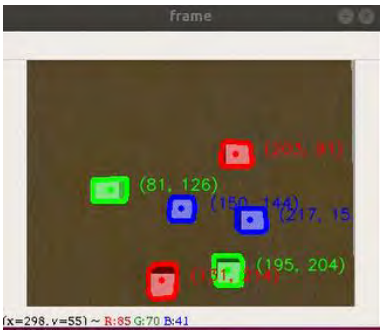


Imagen 273-Imagen la cámara

En este punto, el robot está en la posición desde la cual se toman las imágenes, por tanto, en esta posición se deben actualizar las coordenadas de las piezas. Lo cual implica, que a partir de ahora el robot debe seguir realizando el *pick and place,* pero teniendo en cuenta las nuevas coordenadas de la pieza roja que hemos desplazado.

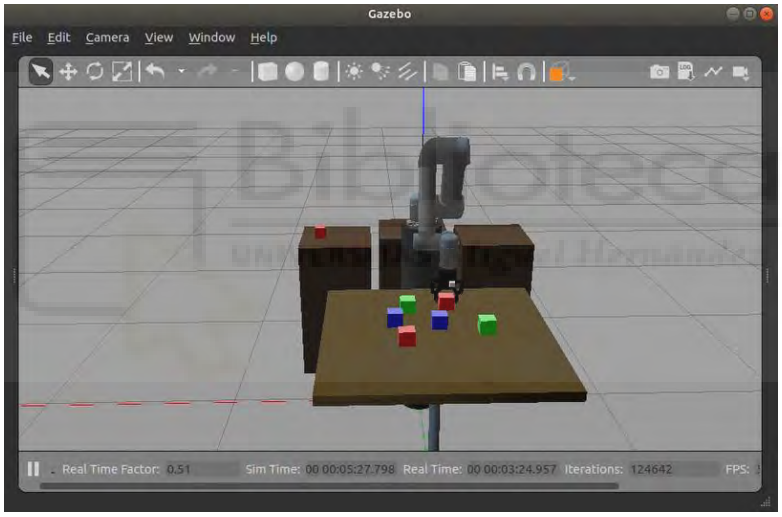


Imagen 274-Robot cogiendo la pieza roja que se había desplazado

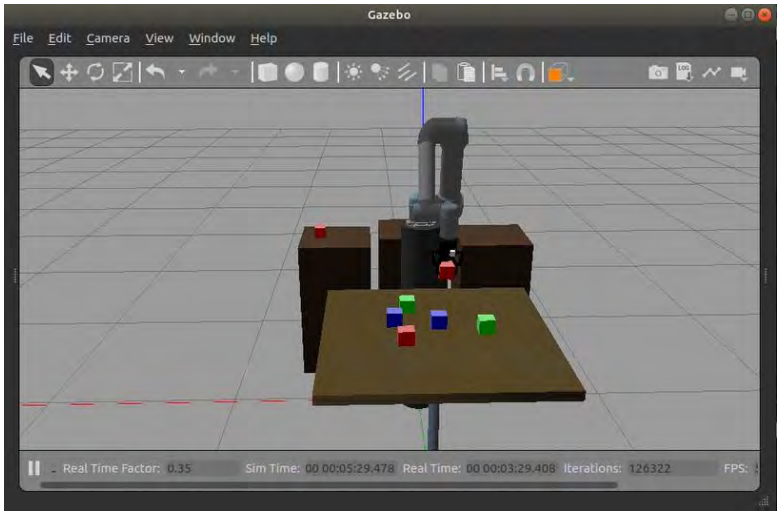


Imagen 275-Robot desplazando segunda pieza roja

Con las imágenes 274 y 275 nos podemos percatar de que el robot ha respondido exitosamente ante el movimiento de una de las piezas, actualizando sus coordenadas sin problemas.

Prosiguiendo con el *pick and place*, mientras el robot se dirige a dejar la segunda pieza roja, se va a proceder a eliminar la última pieza roja que queda sobre la mesa.

Imagen 276-Eliminando tercera pieza roja

Imagen 277-Tercera pieza roja eliminada

Después de eliminar dicha pieza, se comprobará si el robot responde exitosamente y se dirige directamente a recoger piezas azules o, por el contrario, el robot erra y se dirige a recoger una pieza inexistente.

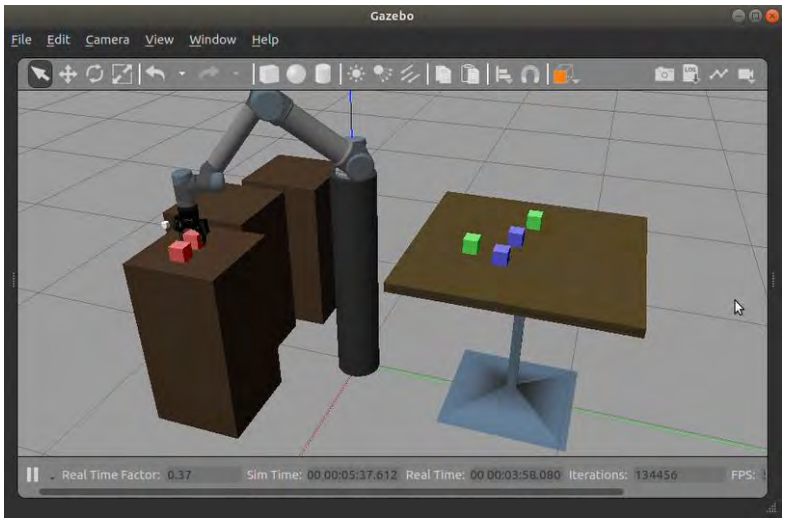


Imagen 278-Dejando pieza roja

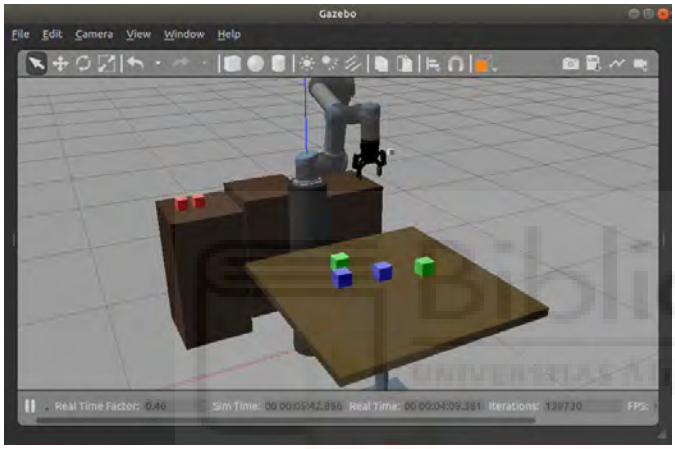


Imagen 279-Robot en posición de tomar imágenes

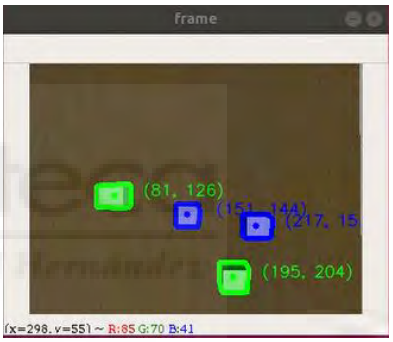


Imagen 280-Imagen la cámara

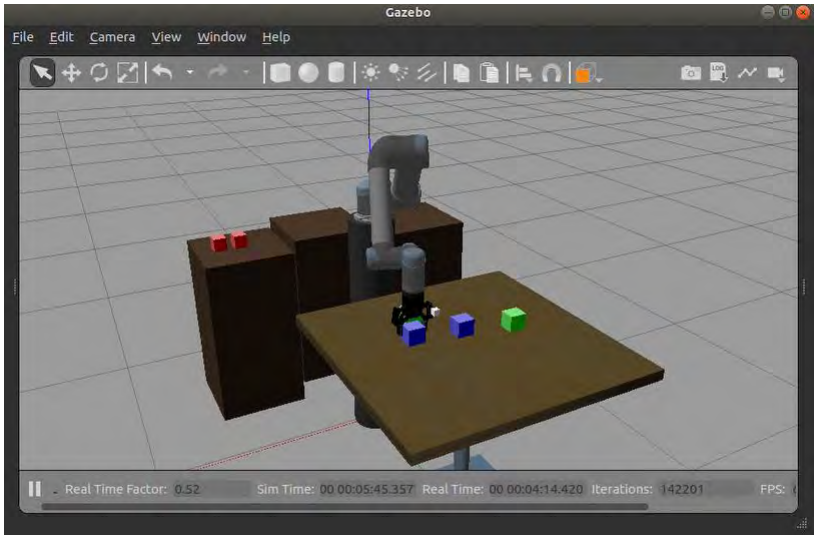


Imagen 281-Robot cogiendo pieza azul

Viendo la imagen 281, podemos notar que, si eliminamos una pieza de la escena, el robot sigue funcionando correctamente y simplemente, pasa a coger la siguiente pieza, ya que se actualizan las coordenadas de dichas piezas.

Seguidamente, vamos a añadir una pieza más a la mesa para saber si el robot responde correctamente ante la adición de alguna pieza más.

Imagen 282-Añadiendo otra pieza azul

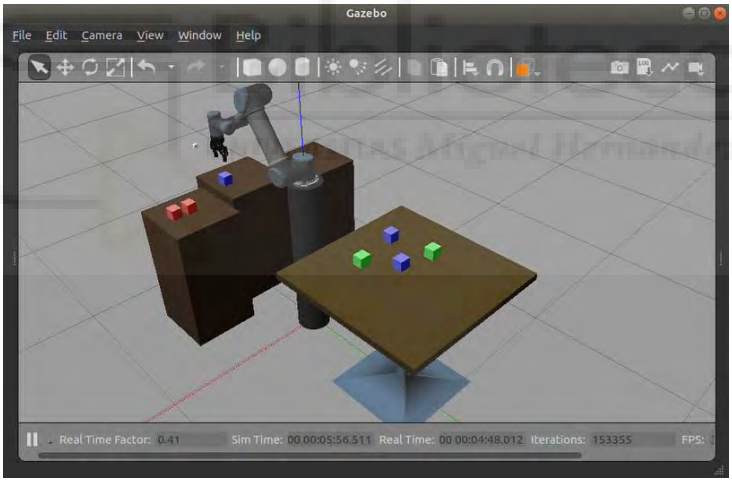


Imagen 283-Pieza azul añadida a la simulación

Imagen 284-Robot en posición de tomar imágenes

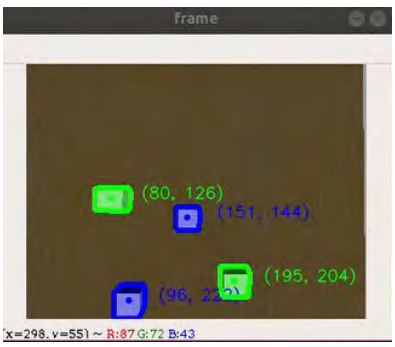


Imagen 285-Imagen la cámara

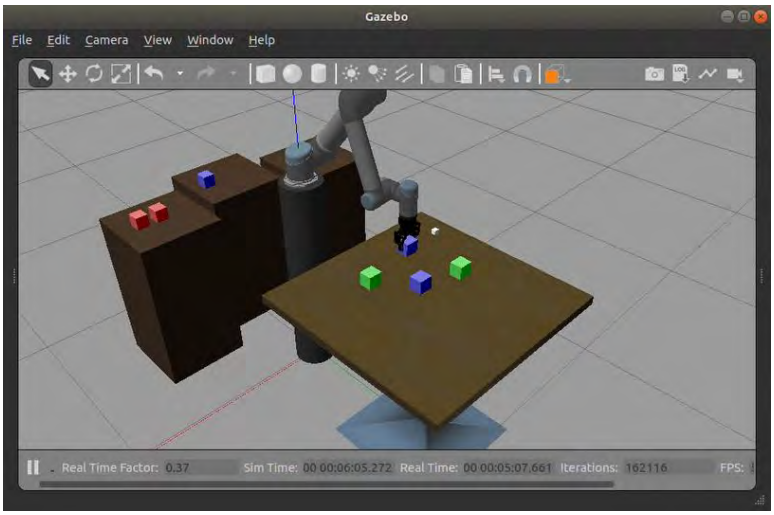


Imagen 286-Robot cogiendo pieza azul

Con esta imagen 286, queda claro que, si añadimos otra pieza a la mesa, el *pick and place* sigue ejecutándose satisfactoriamente. Vamos a seguir viendo el resto de la ejecución.

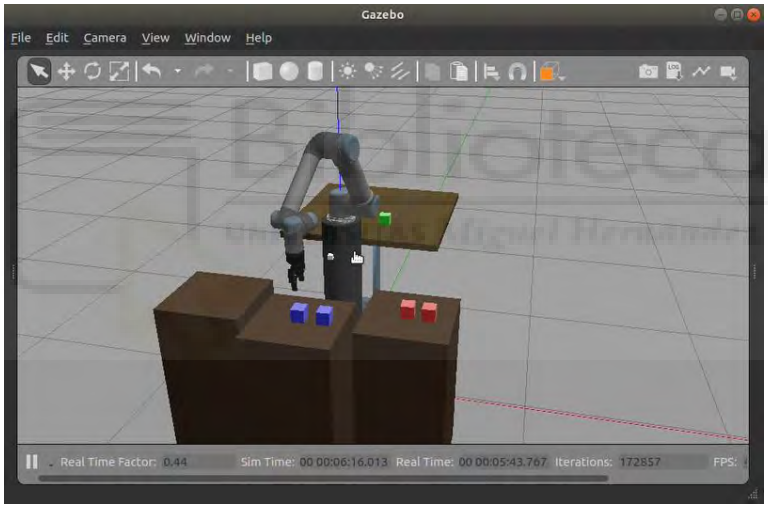


Imagen 287-Robot dejando segunda pieza azul

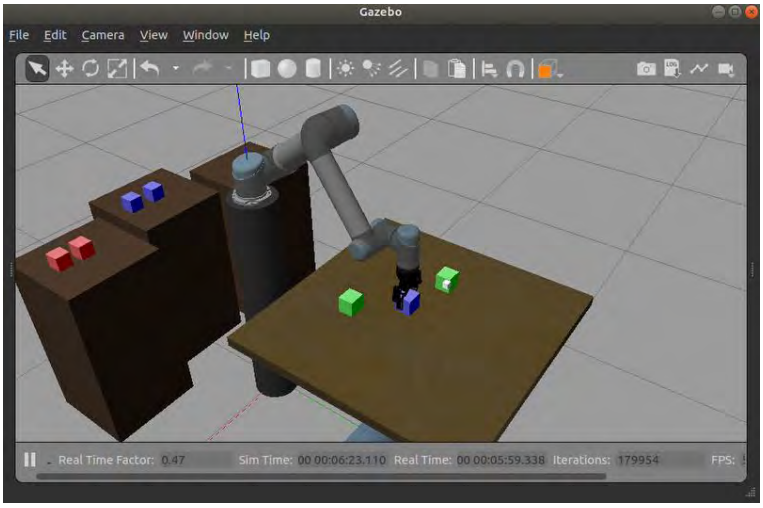


Imagen 288-Robot cogiendo tercera pieza azul

Por último, mientras el robot desplaza la tercera pieza azul, una de las piezas verdes va a ser eliminada y la otra se va a mover a otro lugar.

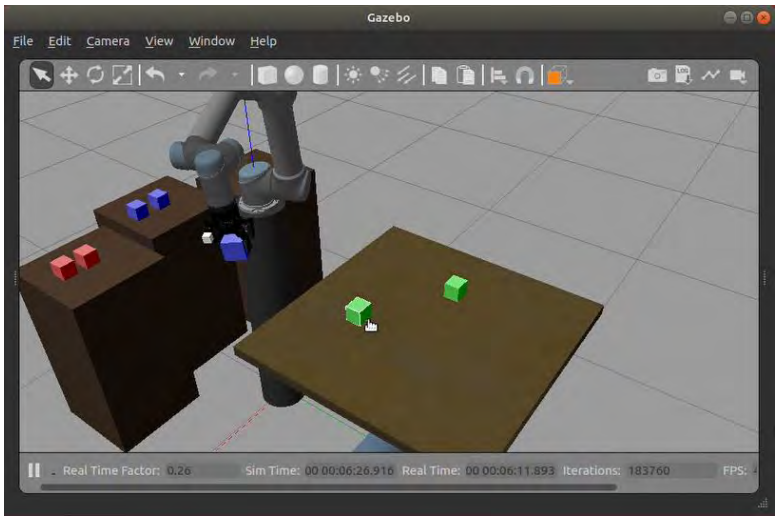


Imagen 289-Eliminando una pieza verde

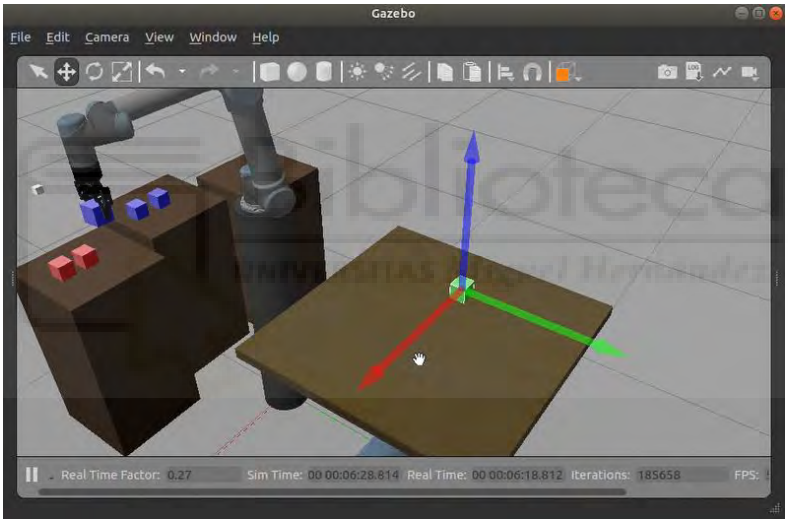


Imagen 290-Una pieza verde eliminada y desplazando la otra pieza verde

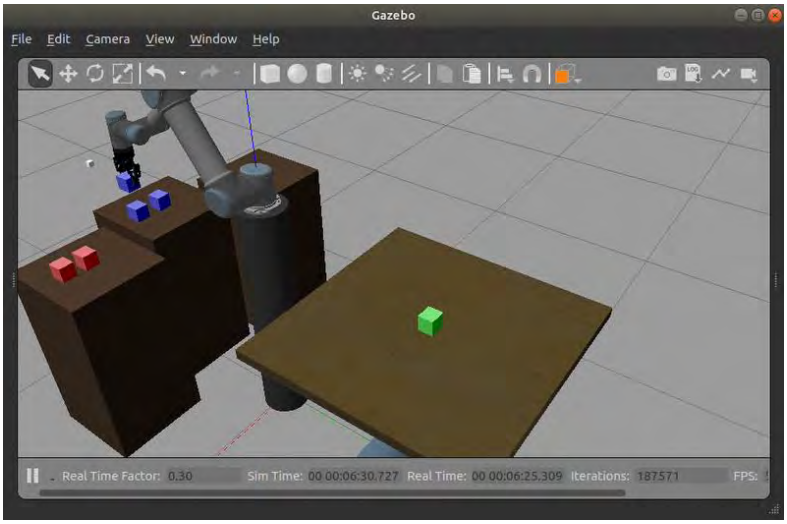


Imagen 291-Pieza verde desplazada

Imagen 292-Robot en posición de tomar imágenes

Imagen 293-Imagen la cámara

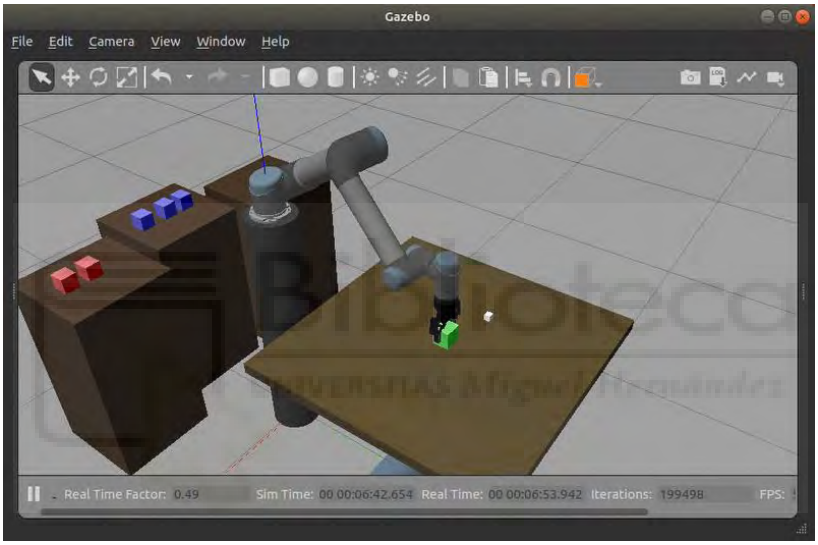


Imagen 294-Robot cogiendo pieza verde

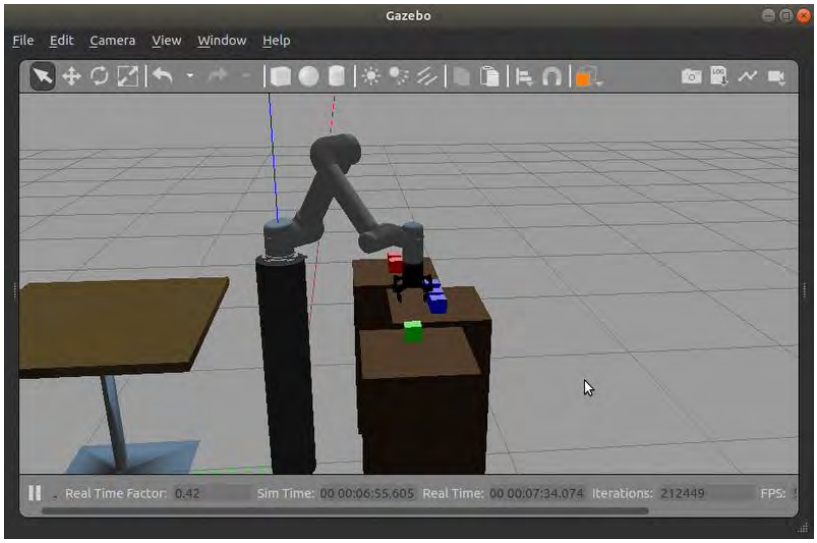


Imagen 295-Pieza verde depositada en soporte

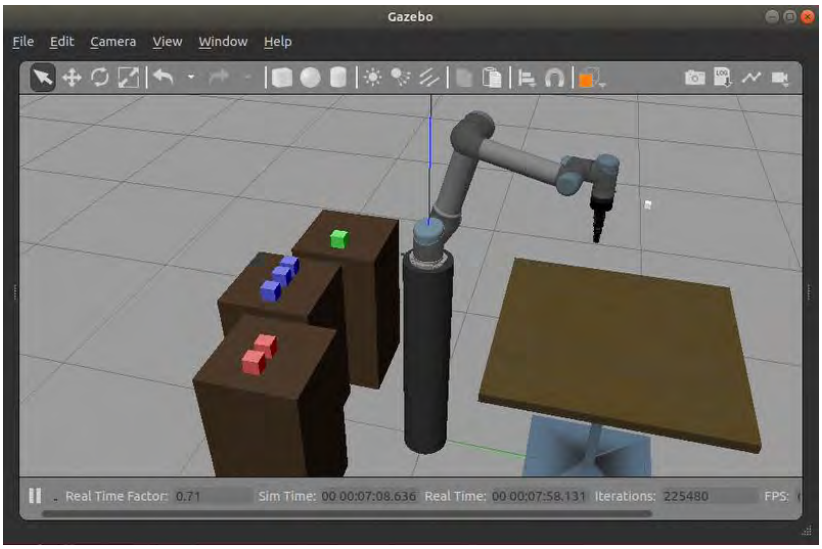


Imagen 296-Fin de ejecución

Como hemos visto, en el proceso de ejecución de este apartado 6.2.2, el programa desarrollado para llevar a cabo la aplicación de *pick and place*, está dotado con capacidades suficiente para hacer frente a movimientos, adiciones o eliminaciones en las diferentes piezas que debe recoger el robot.

7. Conclusiones

7.1. Objetivos alcanzados

A lo largo de todo el desarrollo del proyecto, se han cumplido los objetivos que se van a mencionar a continuación.

- Se han obtenido nociones sobre el funcionamiento de algunas aplicaciones de la robótica industrial.
- Se ha logrado un conocimiento sobre la utilidad de la robótica colaborativa y sus ventajas frente a la robótica industrial convencional
- Se ha alcanzado un nivel solvente de desempeño, con el sistema operativo *Ubuntu* y con el uso de las funcionalidades del terminal y sus diferentes comandos.
- Se ha obtenido una familiarización con el *software ROS* y se ha aprendido a manejar su sistema de comunicaciones. Además, se han adquirido diversas técnicas de *ROS* relacionadas con la robótica.
- Se han obtenido las capacidades suficientes para entender la estructura de los modelos robóticos en lenguaje *URDF* y su proceso de creación, modificación y uso.
- Se ha adquirido la solvencia suficiente con *ROS,* para aprovechar la capacidad de *path planning* de su herramienta llamada, *MoveIt.* También se ha logrado aprovechar dichas capacidades, utilizando la interfaz que nos permite utilizar *MoveIt* con lenguaje *Python.*
- Se han obtenido las destrezas suficientes para simular modelos robóticos en el espacio 3D de *Gazebo.*
- Se ha conseguido obtener una simulación del robot *UR5* en el entorno 3D de *Gazebo*.
- Se ha sido capaz de adherir una pinza a la simulación del robot *UR5*.
- Se ha podido mover el robot *UR5* en simulación, tanto por cinemática directa como inversa, utilizando la herramienta *MoveIt*.
- Se ha obtenido la habilidad de abrir y cerrar la pinza utilizando *MoveIt*.
- Se ha logrado incluir una cámara en la simulación del robot.
- Se ha logrado capturar la imagen que recibe la cámara en simulación.
- Se ha podido preparar la imagen capturada para ser procesada mediante algoritmos de visión por computador.
- Se ha construido una simulación de *Gazebo* en la que se encuentra el robot con la pinza y la cámara, y además todos los objetos necesarios para llevar a cabo la aplicación de *pick and place*.
- Se ha logrado simular en el entorno de *Gazebo* los programas que se han desarrollado 5.2.
- Hemos realizado una aplicación manual de manejo del robot, que engloba todas las funcionalidades que necesitamos utilizar del paquete *moveit_commander*,

con la finalidad de familiarizarnos con las distintas formas de mover el robot, previo a la realización de la aplicación final de *pick and place*.

- Se ha programado y simulado una aplicación de *pick and place*, utilizando el robot *UR5*, integrando técnicas de visión por computador mediante la ayuda de una cámara ubicada sobre el último eslabón del robot. Dichas técnicas nos han permitido integrar en la aplicación, un algoritmo de visión que consigue detectar las piezas del escenario y clasificarlas según el color, para conseguir que el robot mueva las piezas y las deje en un lugar u otro según su color.
- Se ha mejorado la aplicación de *pick and place*, con el fin de que, si durante la ejecución de la aplicación, una de las piezas de la escena es movida, eliminada u otra pieza es añadida a la escena, el nuevo programa sea capaz de detectar dichos cambios y manipular a pesar de esto, todas las piezas correctamente, sin cometer errores de posición causados por esas modificaciones de la escena.

7.2. Trabajos futuros

En lo relativo a posibles ramas de trabajo que pueden derivar de este proyecto, podemos comentar varias.

En lo relativo a expandir el aprendizaje que se ha extraído de la realización del proyecto, se podría expandir el conocimiento sobre las múltiples aplicaciones de *ROS.* Con eso lograríamos desarrollar una mayor soltura en lo que se refiere a interconectar dispositivos con ROS, ya sean *software* o *hardware.*

Dentro de *ROS,* centrándonos en las aplicaciones robótica y las simulaciones con *Gazebo*, podríamos realizar simulaciones con otros robots. La cantidad de robots que se pueden controlar con *ROS* es muy elevada, ya solo en el repositorio de *ROS-Industrial* se pueden simular robots de *ABB, KUKA* y *Universal Robots* entre otros.

Podría ser interesante pensar como adaptar nuestra aplicación a otro tipo de robots, porque en un proyecto real, no siempre podremos utilizar el robot que queramos.

En cuanto a lo relativo a posibles modificaciones o mejoras en nuestra aplicación, podríamos hacer otro algoritmo de visión que clasificara las piezas por tamaño o incluso por forma. Si hiciéramos estos nuevos algoritmos de visión, se podría preguntar al usuario de qué forma quiere clasificar los objetos, y según la elección se podría usar un algoritmo u otro. O incluso realizar un algoritmo que clasificara por color, tamaño y forma, con lo que tendríamos muchos tipos de objetos distintos.

Se podría sustituir la mesa que sostiene las piezas, por una cinta transportadora que nos surtiera las piezas. También podríamos investigar sobre la existencia de algún modelo *URDF* de una ventosa, que fuera compatible con nuestro robot, ya que las ventosas son muy utilizadas en la industria debido a que son más fáciles de usar que las pinzas.

Todo eso se resume en, intentar reforzar los conocimientos adquiridos sobre las herramientas de *software,* como puede ser *ROS, Gazebo* y *MoveIt*, y, además, ampliar la gama de aplicaciones que se podrían realizar con aplicaciones de visión parecidas a la que se ha realizado, o disposiciones de la simulación distintas a la que tenemos. Todo esto nos servirá para expandir nuestros conocimientos en el tema de la robótica y, además, nos servirá para ampliar la gama de aplicaciones robóticas con las que hemos sido capaces de trabajar.

8. Referencias

[1] Antonio Barrientos, Luís Felipe Peñín, Carlos Balaguer y Rafael Aracil, "*Fundamentos de robótica", Madrid, McGray-Hill 1997.*

[2] Arturo Gil Aparicio, *"Apuntes de Robótica",* área de Ingeniería de Sistemas y Automática, Universidad Miguel Hernández de Elche.

[3] Luis Miguel Jiménez García, *"Apuntes visión por computador",* área de Ingeniería de Sistemas y Automática, Universidad Miguel Hernández de Elche.

[4] "*Manual de usuario robot UR5/CB3"*, Universal Robots, https://www.universal-robots.com/download/manuals-cb-series/user/ur5/33/user-manual-ur5-cb-series-sw33-spanish/

[5] Z. Zhang, *"A flexible new technique for camera calibration",* in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 22, no. 11, pp. 1330-1334, Nov. 2000, doi: 10.1109/34.888718.

[6] R. Suárez, J. Rosell, M. Vinagre, F. Cortes, A. Ansuategui, I. Maurtua, D. Martin, A. Guash, J. Azpiazu, D. Serrano y N. García, *"Robot Operating System",* Grupo de Trabajo de Innovación de la Asociación Española de Robótica y Automatización (AER), https://www.aer-automation.com/wp-content/uploads/2022/03/ROS_articuloAER.pdf.

9. Anejos

Anejo I: Código nodo de captación y procesamiento de imagen

```python
#!/usr/bin/env python
"""
Python node for reading camera data
"""

import sys, time
import numpy as np
import cv2 as cv
import roslib
import rospy
from sensor_msgs.msg import CompressedImage  # ROS messages

print('opencv version: ', cv.__version__)

class image_read:
    def __init__(self):
        # Define the subscriber topic
        self.subscriber = rospy.Subscriber(("/myur5/camera1/image_raw/compressed"),
                                           CompressedImage, self.callback, queue_size=1)

    def callback(self, ros_data):
        global coordenates, n_b, n_g, n_r
        coordenates = []
        def dibujar(mask, color):
            n = 0
            coord = []
            contours, hiterachy = cv.findContours(mask, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_NONE)

            for c in contours:
                area = cv.contourArea(c)
                if area > 100:
                    n = n + 1
                    m = cv.moments(c)
                    if m["m00"] == 0:
                        m["m00"] = 1
                    x = int(m["m10"] / m["m00"])
                    y = int(m["m01"] / m["m00"])
                    cv.circle(frame, (x, y), 3, color, -1)
                    font = cv.FONT_HERSHEY_SIMPLEX
                    cv.putText(frame, "(" + str(x) + ", " + str(y) + ")", (x + 28, y), font, 0.5, color, 1, cv.LINE_AA)
                    convexhull = cv.convexHull(c)
                    cv.drawContours(frame, [convexhull], 0, color, 3)
                    coord.append(x)
                    coord.append(y)
            return n, coord


        """Here images are read and processed"""
        # ROJO
        redBajo1 = np.array([0, 100, 20], np.uint8)
        redAlto1 = np.array([8, 255, 255], np.uint8)
        redBajo2 = np.array([175, 100, 20], np.uint8)
        redAlto2 = np.array([179, 255, 255], np.uint8)

        # AZUL
        blueBajo = np.array([100, 100, 20], np.uint8)
        blueAlto = np.array([125, 255, 255], np.uint8)

        # VERDE
        greenBajo = np.array([45, 100, 20], np.uint8)
        greenAlto = np.array([95, 255, 255], np.uint8)

        np_arr = np.fromstring(ros_data.data, np.uint8)
        frame = cv.imdecode(np_arr, cv.IMREAD_COLOR)

        hsv = cv.cvtColor(frame, cv.COLOR_BGR2HSV)
        maskRed1 = cv.inRange(hsv, redBajo1, redAlto1)
        maskRed2 = cv.inRange(hsv, redBajo2, redAlto2)
        maskRed = cv.bitwise_or(maskRed1, maskRed2)
        maskBlue = cv.inRange(hsv, blueBajo, blueAlto)
        maskGreen = cv.inRange(hsv, greenBajo, greenAlto)
```

```
72
73              piec_blue = []
74              piec_green = []  # Vectores que almacenan la posicion en pixels de cada una de las piezas
75              piec_red = []
76
77              n_b, piec_blue = dibujar(maskBlue, (255, 0, 0))
78              coordenates.append(piec_blue)
79              n_g, piec_green = dibujar(maskGreen, (0, 255, 0))
80              coordenates.append(piec_green)
81              n_r, piec_red = dibujar(maskRed, (0, 0, 255))
82              coordenates.append(piec_red)
83
84              cv.imshow('frame', frame)
85              cv.waitKey(2)
86
87          def rev_coord(self):
88              global coordenates, n_b, n_g, n_r
89              return n_b, n_g, n_r, coordenates
90
91
92      def main(args):
93          """Initializes and clearup ros node"""
94          ic = image_read()
95          rospy.init_node('image_read', anonymous=True)
96          try:
97              rospy.spin()
98          except KeyboardInterrupt:
99              print('Shutting down the ROS Image Reader Node')
100             cv.destroyAllWindows()
101
102     if __name__ == '__main__':
103         main(sys.argv)
```

Anejo II: Código *Python* de la aplicación manual

```python
#!/usr/bin/env python

import sys
import copy
import rospy
import numpy as np
import math as m
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
from math import pi
from std_msgs.msg import String
from moveit_commander.conversions import pose_to_list
from moveit_msgs.msg import DisplayTrajectory
import cv2 as cv
from pynput import keyboard
from read_camera_image_mult import image_read

def direct_kin():
    current_joint_values = move_group_interface_arm.get_current_joint_values()
    print('Current joint values:')
    print(current_joint_values)
    print('\nEnter new values:\n')
    try:
        joint_goals = [float(input("Enter joint " + str(i) + " value: ")) for i in range(6)]
    except:
        print("Error: bad joint_goals")
        exit()
    move_group_interface_arm.go(joint_goals, wait = True)
    print("New goals for the robot: " + str(joint_goals))
    move_group_interface_arm.stop
```

```python
def on_press(key):
    global new_values
    try:
        current_joint_values = move_group_interface_arm.get_current_joint_values()
        # print(current_joint_values)
        if key.char == "1":
            new_values[0] += 0.1121997376
            new_values[1:] = current_joint_values[1:]
        elif key.char == "2":
            new_values[0] = current_joint_values[0]
            new_values[1] += 0.1121997376
            new_values[2:] = current_joint_values[2:]
        elif key.char == "3":
            new_values[:1] = current_joint_values[:1]
            new_values[2] += 0.1121997376
            new_values[3:] = current_joint_values[3:]
        elif key.char == "4":
            new_values[:2] = current_joint_values[:2]
            new_values[3] += 0.1121997376
            new_values[4:] = current_joint_values[4:]
        elif key.char == "5":
            new_values[:3] = current_joint_values[:3]
            new_values[4] += 0.1121997376
            new_values[5] = current_joint_values[5]
        elif key.char == "6":
            new_values[:4] = current_joint_values[:4]
            new_values[5] += 0.1121997376
        elif key.char == "q":
            new_values[0] -= 0.1121997376
            new_values[1:] = current_joint_values[1:]
        elif key.char == "w":
            new_values[0] = current_joint_values[0]
            new_values[1] -= 0.1121997376
            new_values[2:] = current_joint_values[2:]
        elif key.char == "e":
            new_values[:1] = current_joint_values[:1]
            new_values[2] -= 0.1121997376
            new_values[3:] = current_joint_values[3:]
        elif key.char == "r":
            new_values[:2] = current_joint_values[:2]
            new_values[3] -= 0.1121997376
            new_values[4:] = current_joint_values[4:]
```

```
        elif key.char == "t":
            new_values[:3] = current_joint_values[:3]
            new_values[4] -= 0.1121997376
            new_values[5] = current_joint_values[5]
        elif key.char == "y":
            new_values[:4] = current_joint_values[:4]
            new_values[5] -= 0.1121997376
        elif key.char == "a":
            new_values = [0, 0, 0, 0, 0, 0]
        elif key.char == "s":
            new_values = [0, -1.4981833546922498, 1.5076458045811982, -0.0019684415915168785, 0, 0]
        elif key.char == "d":
            print('Current joint values:')
            print(current_joint_values)
        elif key.char == "o":
            move_group_interface_gripper.go(gripper_open, wait=True)
            move_group_interface_gripper.stop()
        elif key.char == "c":
            move_group_interface_gripper.go(gripper_close, wait=True)
            move_group_interface_gripper.stop()
        elif key.char == "f":
            sys.exit()
        move_group_interface_arm.go(new_values, wait = True)
        move_group_interface_arm.stop
    except AttributeError:
        print("You shouldn't press special characters")

def inverse_kin():
    current_pose = move_group_interface_arm.get_current_pose()
    print("Current values of end effector pose:")
    print(current_pose)
    print('\nEnter new end effector values:\n')

    pose_target = geometry_msgs.msg.Pose()
    pose_target.position.x = float(input('Enter position x: '))
    pose_target.position.y = float(input('Enter position y: '))
    pose_target.position.z = float(input('Enter position z: '))
    z = int(input("Do you want to modify the orientation using a quaternion?(Yes(1) or No(2)): "))
    if z == 1:
        pose_target.orientation.w = float(input('Enter value w: '))
        pose_target.orientation.x = float(input('Enter value x: '))
        pose_target.orientation.y = float(input('Enter value y: '))
        pose_target.orientation.z = float(input('Enter value z: '))
    elif z == 2:
        pose_target.orientation = current_pose.pose.orientation

    move_group_interface_arm.set_pose_target(pose_target)

    move_group_interface_arm.go(wait=True)

    move_group_interface_arm.stop()
    move_group_interface_arm.clear_pose_targets()

gripper_open = [0.005]
gripper_close = [0.24]

if __name__ == '__main__':
    obj_img = image_read()

    new_values = []
    # initialize moveit_comander and rospy node
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('move_group_python_interface', anonymous=True)

    # instantiate a RobotCommander object
    robot = moveit_commander.robot.RobotCommander()

    # create a MoveGroupCommander object
    # we will move the arm or the gripper with these objects
    group_name_1 = "ur5_arm"
    group_name_2 = "gripper"
    move_group_interface_arm = moveit_commander.move_group.MoveGroupCommander(group_name_1)
    move_group_interface_gripper = moveit_commander.move_group.MoveGroupCommander(group_name_2)

    # Create DisplayTrajectory ROS publisher which is used to
    # display trajectoris in Rviz
    display_trajectory_publisher = rospy.Publisher('/move_group/display_planned_path', DisplayTrajectory, queue_size=20)

    r = 0

    # print("End effector: " + str(move_group_interface_arm.get_end_effector_link()))
```

```
157     while True:
158         print('-------------------------------------------------')
159         print('OPERATING MODES:')
160         print('1: Direct kinematic')
161         print('2: Direct kin - Manual mode')
162         print('3: Inverse kinematic - MoveJ')
163         print('4: Open gripper')
164         print('5: Close gripper')
165         print('6: Go Home')
166         print('7: Go camera position')
167         print('8: See the object coordenates')
168         print('9: Close program')
169         print('-------------------------------------------------')
170         try:
171             num = int(input('\nSelect one mode: '))
172         except:
173             print(rospy.logerr('Bad mode selection'))
174             exit()
175 
176         if num == 1:
177             print("\nDIRECT KINEMATIC")
178             direct_kin()
179         elif num == 2:
180             print("\nMANUAL MODE")
181             print("*********************************")
182             print("CONTROLS:")
183             print("Positive rotation of joints: 1, 2, 3, 4, 5, 6")
184             print("Negative rotation of joints: q, w, e, r, t, y")
185             print("Open the gripper: press 'o'")
186             print("Close the gripper: press 'c'")
187             print("Position AllZeros: press 'a'")
188             print("Position Home: press 's'")
189             print("See the current joint values: press 'd'")
190             print("Close the manual mode: press 'f'")
191             print("*********************************")
192             current_joint_values = move_group_interface_arm.get_current_joint_values()
193             new_values = current_joint_values
194             with keyboard.Listener(
195                     on_press=on_press) as listener:
196                 listener.join()
197             print("Manual mode closed\n")
```

```
198         elif num == 3:
199             print('\nINVERSE KINEMATIC')
200             inverse_kin()
201         elif num == 4:
202             print("OPENING THE GRIPPER")
203             move_group_interface_gripper.go(gripper_open, wait=True)
204             move_group_interface_gripper.stop()
205         elif num == 5:
206             print("CLOSING THE GRIPPER")
207             move_group_interface_gripper.go(gripper_close, wait=True)
208             move_group_interface_gripper.stop()
209         elif num == 6:
210             print("MOVING TO HOME POSITION")
211             joint_goals = [0.0, -1.5447, 1.5447, -1.5794, -1.5794, 0.0]
212             move_group_interface_arm.go(joint_goals, wait=True)
213             move_group_interface_arm.stop()
214         elif num == 7:
215             print("MOVING TO THE POSITION WHERE WE PROCESS PICTURES")
216             joint_goals = [1.3909180384024973, -1.2970095837866964, 1.4559604820238743, -1.75, -1.573950195686849,
217                            -0.1819244873697139]
218             move_group_interface_arm.go(joint_goals, wait = True)
219             move_group_interface_arm.stop()
220         elif num == 8:
221             coord = []
222             _, _, _, coord = obj_img.rev_coord()
223             print("Coordenadas piezas azules: ({})\nCoordendas piezas verdes: ({})\nCoordenadas piezas rojas: ({})".format(
224                 coord[0], coord[1], coord[2]))
225         elif num == 9:
226             print('Finishing the program')
227             exit()
228         else:
229             print(rospy.logerr('Bad mode selection'))
230             exit()
```

Anejo III: Código *Python* de la aplicación de *pick and place*

```python
#!/usr/bin/env python

import sys
import copy
import rospy
import numpy as np
import math as m
import moveit_commander
import moveit_msgs.msg
import geometry_msgs.msg
import shape_msgs.msg
from math import pi
from std_msgs.msg import String
from moveit_commander.conversions import pose_to_list
from moveit_msgs.msg import DisplayTrajectory
import cv2 as cv
from pynput import keyboard
from read_camera_image_mult import image_read
import time as t

global current_pose, target_pose_correct_orien
global comp_orien
comp_orien = 1

def pos_take_images():
    global target_pose_correct_orien
    print("MOVING TO CAMERA POSITION...")
    images_goals = [1.3909180384024973, -1.2970095837866964, 1.4559604820238743, -1.75, -1.573950195686849,
                    -0.1819244873697139]
    move_group_interface_arm.go(images_goals, wait=True)
    move_group_interface_arm.stop()
    target_pose_correct_orien = geometry_msgs.msg.Pose()
    target_pose_correct_orien = move_group_interface_arm.get_current_pose()

def home_pos():
    print("MOVING TO HOME POSITION...")
    joint_goals = [0.0, -1.5447, 1.5447, -1.5794, -1.5794, 0.0]
    move_group_interface_arm.go(joint_goals, wait=True)
    move_group_interface_arm.stop()

def home_pos_inv():
    print("MOVING TO HOME POSITION...")
    joint_goals = [-3.14, -1.5447, 1.5447, -1.5794, -1.5794, 0.0]
    move_group_interface_arm.go(joint_goals, wait=True)
    move_group_interface_arm.stop()

def close_gripper():
    print("CLOSING THE GRIPPER...")
    move_group_interface_gripper.go(gripper_close, wait=True)
    move_group_interface_gripper.stop()

def open_gripper():
    print("OPENING THE GRIPPER...")
    move_group_interface_gripper.go(gripper_open, wait=True)
    move_group_interface_gripper.stop()

# Approximate the gripper over the piece
def approximation(x, y):
    global target_pose, comp_orien, target_pose_correct_orien
    target_pose.position.x = x
    target_pose.position.y = y
    target_pose.position.z = 1.1
    if comp_orien > 2:
        target_pose.orientation = target_pose_correct_orien.pose.orientation
    move_group_interface_arm.set_pose_target(target_pose)
    move_group_interface_arm.go(wait=True)
    move_group_interface_arm.stop()
    comp_orien = comp_orien + 1

# Take the piece with the gripper after approximation
def take_piece():
    global target_pose, comp_orien
    target_pose.position.z = 0.96
    move_group_interface_arm.set_pose_target(target_pose)
    move_group_interface_arm.go(wait=True)
    move_group_interface_arm.stop()

```

```python
# Transform from image coordenates to world coordenates
def convert_m2w(p_px):
    Cx = 160
    Cy = 120.5
    Zc = 0.534395
    fx = 199.8938206925
    fy = 199.8938206925

    x_c = ((p_px[0]-Cx)/fx)*Zc
    y_c = ((p_px[1]-Cy)/fy)*Zc
    x_w = x_c + 0.004627
    y_w = 0.758958 - y_c

    return x_w, y_w

def red_pieces(c):
    global current_pose, target_pose
    global incr_r
    coord_red = []
    coord_red.append(c[0])
    coord_red.append(c[1])
    x_r, y_r = convert_m2w(coord_red)

    # Approx red piece
    approximation(x_r, y_r)

    # Open the gripper
    open_gripper()

    # Take red piece
    take_piece()

    # Close the gripper
    close_gripper()

    # Return to approximation pos
    approximation(x_r, y_r)

    # Move to home position
    home_pos()

    # Move the TCP over the red position
    approx_goals = [-0.9023952823594588, -0.7322973147618699, 1.2709732199546568, -2.115323358711743, -1.5688081113077317,
                    -0.9020939621886495]
    move_group_interface_arm.go(approx_goals, wait=True)
    move_group_interface_arm.stop()

    # Leave the piece
    current_pose_2 = move_group_interface_arm.get_current_pose()
    target_pose.position = current_pose_2.pose.position
    target_pose.orientation = current_pose_2.pose.orientation
    target_pose.position.x = 0.549634458556 - incr_r
    move_group_interface_arm.set_pose_target(target_pose)
    move_group_interface_arm.go(wait=True)
    target_pose.position.z = 0.967
    move_group_interface_arm.set_pose_target(target_pose)
    move_group_interface_arm.go(wait=True)
    move_group_interface_arm.stop()

    open_gripper()

    # Return over the red position
    target_pose.position.z = 1.1
    move_group_interface_arm.set_pose_target(target_pose)
    move_group_interface_arm.go(wait=True)
    move_group_interface_arm.stop()

    pos_take_images()
```

```python
def blue_pieces(c):
    global current_pose, target_pose
    global incr_b

    coord_blue = []
    coord_blue.append(c[0])
    coord_blue.append(c[1])

    x_b, y_b = convert_m2w(coord_blue)

    # Aprox blue piece
    approximation(x_b, y_b)

    # Take blue piece
    take_piece()

    close_gripper()

    approximation(x_b, y_b)

    home_pos()

    # Move the TCP above the blue position
    approx_goals = [-1.5860453154773948, -0.9473863999013306, 1.6479834864306415, -2.2721118993163927, -1.5642330045305393,
                    -1.5840395145339077]
    move_group_interface_arm.go(approx_goals, wait=True)
    move_group_interface_arm.stop()

    # Leave blue piece
    current_pose_2 = move_group_interface_arm.get_current_pose()
    target_pose.position = current_pose_2.pose.position
    target_pose.orientation = current_pose_2.pose.orientation
    target_pose.position.x = 0.0998218605971 - incr_b
    move_group_interface_arm.set_pose_target(target_pose)
    move_group_interface_arm.go(wait=True)
    target_pose.position.z = 0.967
    move_group_interface_arm.set_pose_target(target_pose)
    move_group_interface_arm.go(wait=True)
    move_group_interface_arm.stop()
```

```python

    open_gripper()

    target_pose.position.z = 1.1
    move_group_interface_arm.set_pose_target(target_pose)
    move_group_interface_arm.go(wait=True)
    move_group_interface_arm.stop()

    pos_take_images()

```

```python
def green_pieces(c):
    global current_pose, target_pose
    global incr_g

    coord_green = []
    coord_green.append(c[0])
    coord_green.append(c[1])

    x_g, y_g = convert_m2w(coord_green)

    # Aprox green piece
    approximation(x_g, y_g)

    # Take blue piece
    take_piece()

    close_gripper()

    approximation(x_g, y_g)

    home_pos_inv()

    # Move the TCP above the green position
    approx_goals = [-2.3516030599181112, -1.0095631474481657, 1.7579949487353765, -2.31427905629861, -1.56649968952454,
                    -2.3444216722389184]
    move_group_interface_arm.go(approx_goals, wait=True)
    move_group_interface_arm.stop()

    # Leave the green piece
    current_pose_2 = move_group_interface_arm.get_current_pose()
    target_pose.position = current_pose_2.pose.position
    target_pose.orientation = current_pose_2.pose.orientation
    target_pose.position.x = -0.350061671499 - incr_g
    move_group_interface_arm.set_pose_target(target_pose)
    move_group_interface_arm.go(wait=True)
    target_pose.position.z = 0.967
    move_group_interface_arm.set_pose_target(target_pose)
    move_group_interface_arm.go(wait=True)
    move_group_interface_arm.stop()

    open_gripper()

```

```
    target_pose.position.z = 1.1
    move_group_interface_arm.set_pose_target(target_pose)
    move_group_interface_arm.go(wait=True)
    move_group_interface_arm.stop()

    pos_take_images()


gripper_open = [0.005]
gripper_close = [0.24]

if __name__=='__main__':
    global target_pose, incr_r, incr_b, incr_g
    obj_img = image_read()  # Objeto de la clase image_read para acceder a sus funciones
    incr_r = 0
    incr_b = 0
    incr_g = 0

    # initialize moveit_comander and rospy node
    moveit_commander.roscpp_initialize(sys.argv)
    rospy.init_node('move_group_python_interface', anonymous=True)

    # create a RobotCommander object
    robot = moveit_commander.robot.RobotCommander()

    # create a MoveGroupCommander object
    # we will move the arm or the gripper with these objects
    group_name_1 = "ur5_arm"
    group_name_2 = "gripper"
    move_group_interface_arm = moveit_commander.move_group.MoveGroupCommander(group_name_1)
    move_group_interface_gripper = moveit_commander.move_group.MoveGroupCommander(group_name_2)

    print("----------------------------------")
    print("Starting the pick and place program")
    print("----------------------------------")

    # Move to home position
    home_pos()

    # Move to take images position
    pos_take_images()

    current_pose = geometry_msgs.msg.Pose()
    current_pose = move_group_interface_arm.get_current_pose()

    t.sleep(2)
    coord = []

    n_b, n_g, n_r, coord = obj_img.rev_coord()  # Recibe las coordenadas de las piezas en la imagen
    print("************************")
    print("El numero de piezas de cada color es, rojo:{}, azul:{}, verde:{}".format(n_r, n_b, n_g))
    print("La cordenadas de los objetos son: {}".format(coord))
    print("************************")

    target_pose = geometry_msgs.msg.Pose()
    target_pose.orientation = current_pose.pose.orientation

    nt = n_r+n_b+n_g
    while(nt>0):
        if(n_r>0):
            red_pieces(coord[2])
            incr_r = incr_r + 0.1
        elif(n_b>0):
            blue_pieces(coord[0])
            incr_b = incr_b + 0.1
        elif(n_g>0):
            green_pieces(coord[1])
            incr_g = incr_g + 0.1
        t.sleep(2)
        n_b, n_g, n_r, coord = obj_img.rev_coord()
        nt = n_r+n_b+n_g

    cv.destroyAllWindows()
```